University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Łukasz Białek
Student no. 277555

Actions in information-rich environments: a paraconsistent approach

PhD dissertation
in COMPUTER SCIENCE

Supervisor:
Prof. dr hab. Andrzej Szałas
Faculty of Mathematics, Informatics and Mechanics
University of Warsaw

May 2020

Supervisor's statement

Hereby I confirm that the presented thesis was prepared under my supervision and that it fulfils the requirements for the degree of PhD of Computer Science.

Date Supervisor's signature

Author's statement

Hereby I declare that the presented thesis was prepared by me and none of its contents was obtained by means that are against the law.

The thesis has never before been a subject of any procedure of obtaining an academic degree.

Moreover, I declare that the present version of the thesis is identical to the attached electronic version.

Date Author's signature

Contents

Abstract

The dissertation presents both theoretical and practical research on action planning under inconsistent and incomplete information. The work is firmly rooted in a related research area of beliefs, belief change and action planning. The proposed solution is based on 4QL, a four-valued query language. Its initial syntax and semantics are extended by new operations on belief bases and actual planning machinery. A unique feature of the 4QL language is the presence of truth values $\mathbf{t}$ (*true*), $\mathbf{f}$ (*false*), $\mathbf{i}$ (*inconsistent*), $\mathbf{u}$ (*unknown*) as well as the unrestricted use of negation in both conclusions and premises of rules while retaining intuitive results and tractable query evaluation.

Before introducing the actual research results, the theoretical basics of 4QL, belief bases and belief structures are reminded. Then a $4QL^{Bel}$ language is presented. It is designed to perform doxastic reasoning on belief bases and belief structures. It extends the 4QL language by introducing a $\mathrm{Bel}()$ operator which allows to ask queries individually to each element of the belief base and constructing results as least upper bounds of output sets with respect to 4QL's information ordering. Using this operator one can query a belief base for global beliefs shared among all world representations.

Then belief bases are further expanded with rigid and flexible constraints which allow for maintaining the desired states of knowledge during planning. Violating constraints of a belief base results in the base always responding with the truth value $\mathbf{u}$ for all queries. Moreover, a novel operation on belief bases, belief shadowing, is introduced. The operation allows for transient change of an agent's beliefs using the ones from another belief bases. What makes it unique is, among others, that flexible constraints get altered as well which allows for expressing not only a belief change but also a constraints change.

Finally, an ACTLOG language is introduced, with 4QL's syntax expanded with the ADL-inspired action schema where ADL is a well known Action Definition Language. Due to the presence of non-classical truth values, the language is capable of specifying planning problems under incomplete and possibly inconsistent information. In particular, inconsistent beliefs can be used in both actions preconditions and planning goals to be achieved. This feature is not commonly found in other planning engines. The language also allows for conditional effects of actions as well as for using belief operators inside action specifications. Apart from that, composite actions are also introduced utilizing parallel, sequential and conditional operators. Composite actions not only increase expressiveness of the language but also can be seen as action templates which can significantly reduce planning complexity.

All theoretical results presented in the thesis have been implemented as a part of the inter4QL software being an open-source interpreter for the 4QL language. As part of the implementation the language has been extended with $4QL^{Bel}$ constructs introduced in the thesis. Moreover, an experimental ACTLOG planner supporting both normal and composite actions has been introduced. The implementation details and discussion of proposed solutions are provided. The dissertation uses the implementation to verify a set of examples with accents put on strengths of the proposed solutions.

Streszczenie

Rozprawa prezentuje zarówno teoretyczne jak i praktyczne wyniki badań w dziedzinie planowania akcji z wykorzystaniem niepełnej i sprzecznej informacji. Cała praca jest silnie osadzona w pokrewnej dziedzinie przekonań, ich zmian oraz planowania akcji. Zaproponowane rozwiązania opierają się na czterowartościowym języku zapytań 4QL. Jego unikalną cechą jest obecność czterech wartości logicznych $\mathfrak{t}$ (*true - prawda*), $\mathfrak{f}$ (*false - fałsz*), $\mathfrak{i}$ (*inconsistent - sprzeczność*), $\mathfrak{u}$ (*unknown - niewiedza*) oraz brak ograniczeń w użyciu negacji zarówno w ciałach reguł jak i ich konkluzjach. Wspomniane cechy nie wpływają jednocześnie na intuicyjność otrzymywanych wyników oraz efektywnie obliczalną ewaluację zapytań.

Przed przedstawieniem pierwszych wyników badań, rozprawa przypomina teoretyczne podstawy samego języka 4QL oraz baz i struktur przekonań. Następnie wprowadzony zostaje język $4QL^{Bel}$ zaprojektowany do wsparcia wnioskowania na bazach i strukturach przekonań. Rozszerza on język 4QL poprzez wprowadzenie operatora $\mathrm{Bel}()$ opierającego swoją zasadę działania na zadawaniu indywidualnych zapytań do poszczególnych elementów bazy przekonań a następnie na konstruowaniu ostatecznego wyniku jako kresu górnego zbiorów odpowiedzi (względem porządku informacyjnego wartości logicznych języka 4QL). Używając tego operatora możliwe jest odpytywanie wspomnianych struktur o globalne przekonania współdzielone pomiędzy wszystkimi reprezentacjami świata.

Bazy przekonań zostają następnie rozszerzone o twarde i elastyczne więzy pozwalające na zachowywanie określonych stanów wiedzy podczas planowania akcji. Złamanie więzów bazy przekonań skutkuje przełączeniem bazy w tryb odpowiadania wartością logiczną $\mathfrak{u}$ na wszelkie zapytania do niej kierowane. Co więcej, wprowadzona zostaje tu również operacja przysłaniania przekonań działająca na bazach przekonań. Operacja ta pozwala na potencjalnie krótkotrwałą zmianę przekonań agenta na inne, zdefiniowane w alternatywnej bazie przekonań. Unikalną cechą tego rozwiązania jest, między innymi, wsparcie dla przysłaniania również elastycznej części więzów co pozwala na wyrażanie zmian nie tylko przekonań ale i więzów.

Następnie rozprawa przedstawia język ACTLOG będący rozszerzeniem języka 4QL o składnię definiowania akcji inspirowaną rodziną języków akcji, w skład której wchodzi język ADL (Action Definition Language). Z racji użycia logiki zawierającej nieklasyczne wartości logiczne, ACTLOG pozwala na definiowanie problemów planowania opierających się na niepełnych lub też sprzecznych informacjach. W szczególności, sprzeczne przekonania mogą być używane zarówno w warunkach wstępnych akcji jak i w formule określającej cel planowania. Język ten pozwala również na warunkowe efekty akcji oraz na użycie operatorów przekonań podczas specyfikowania akcji. Ponadto, rozprawa przedstawia akcje złożone oparte na operatorach sekwencyjnego, równoległego oraz warunkowego złożenia akcji. Tego typu akcje nie tylko zwiększają ekspresywność języka ale mogą również być postrzegane jako szablony planów pozwalające na znaczne ograniczenie złożoności obliczeniowej samego planowania.

Wszystkie wyniki teoretyczne zawarte w tej rozprawie zostały zaimplementowane w oprogramowaniu inter4QL będącym otwartym interpreterem języka 4QL. W ramach implementacji język ten został rozszerzony o konstrukcje z języka $4QL^{Bel}$. W skład interpretera wchodzi również planner dostarczający eksprymentalne wsparcie dla języka ACTLOG. Szczegóły implementacyjne oraz omówienie zastosowanych rozwiązań są częścią tej rozprawy. Z drugiej strony, rozprawa używa dostarczonej implementacji w celach weryfikacji przygotowanych przykładów skupiając się jednocześnie na mocnych stronach zaproponowanych rozwiązań.

Keywords

Rule Languages, Reasoning about Knowledge and Beliefs, Tractable Languages, Action Languages, Paraconsistent and Paracomplete Reasoning, Belief Bases, Planning

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Informatics, Computer Science

Subject classification

Computing methodologies →
- Artificial intelligence →
  - Knowledge representation and reasoning →
    - Reasoning about belief and knowledge
    - Nonmonotonic, default reasoning and belief revision
  - Planning and scheduling →
    - Multi-agent planning
    - Planning for deterministic actions
    - Planning under uncertainty

Tytuł pracy w języku polskim

Akcje w środowiskach złożonych informacyjnie: podejście parakonsystentne

Publications

Publications of major results from the thesis

- Łukasz Sebastian Białek, Barbara Dunin-Kęplicz i Andrzej Szałas, Towards a Paraconsistent Approach to Actions in Distributed Information-Rich Environments, in: Intelligent Distributed Computing XI, Springer International Publishing, 2017, s. 49–60.
- Łukasz Sebastian Białek, Barbara Dunin-Kęplicz i Andrzej Szałas, Rule-Based Reasoning with Belief Structures, in: Foundations of Intelligent Systems. 23rd International Symposium, ISMIS 2017, Warsaw, Poland, June 26-29, 2017, Proceedings, Springer, Cham 2017, s. 229–239.
- Łukasz Sebastian Białek, Barbara Dunin-Kęplicz i Andrzej Szałas, Belief Shadowing, in: EMAS 2018: Engineering Multi-Agent Systems, Springer, Cham 2019, s. 158–180.
- Łukasz Sebastian Białek, Barbara Dunin-Kęplicz i Andrzej Szałas, A paraconsistent approach to actions in informationally complex environments, in: Annals Of Mathematics And Artificial Intelligence 86 (2019), s. 231–255.

Other related publications

- Jacek Szklarski, Łukasz Sebastian Białek i Andrzej Szałas, Paraconsistent Reasoning in Cops and Robber Game with Uncertain Information: A Simulation-Based Analysis, in: International Journal Of Uncertainty Fuzziness And Knowledge-based Systems 27 (3) 2019, s. 429–455.
- Łukasz Sebastian Białek i Andrzej Szałas, Lightweight Reasoning with Incomplete and Inconsistent Information: a Case Study, in: 2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT), Vol. 3, IEEE, Los Alamitos, California 2014, s. 325–332.

Please note that extensive parts of above papers are used in the thesis directly or in a slightly rephrased form.

Acknowledgments

I would like to express my highest gratitude to my supervisor and advisor, Prof. dr hab. Andrzej Szałas. The priceless guidance through the world of multi-valued logics, beliefs and action planning, a series of discussions about the ideas and work progress and finally a patient and professional support during the whole research process undoubtedly allowed to significantly improve the obtained results.

Throughout the years of my PhD studies I also had a great pleasure to work with Prof. dr hab. Barbara Dunin-Kęplicz to whom I would like to thank for sharing the vast knowledge and experience both during PhD lectures and beyond.

I would like to thank my beloved family - Wife Magdalena, Daughter Weronika, Mom Maria, Dad Zygmunt, Brother Jakub and Grandmother Józefa - for their constant and endless support and encouragement.

Working on all of the components of this PhD thesis was taking a significant amount time. This is why I would like to give my special thanks to my Daughter and Wife for the understanding and additional support to provide me with the time needed to achieve this important milestone. I appreciate this support a lot.

Finally, last but not least, this research was partially supported by the Polish National Science Center grant 2015/19/B/ST6/02589.

Chapter 1

Introduction and related work

1.1. Beliefs in dynamic environments

Reasoning about the beliefs has a long history. In particular is has been intensively studied and applied in Knowledge Representation and Reasoning sub-area of AI [106]. One of the popular implementations of Beliefs in AI is a BDI model [32, 47, 93]. An undeniable feature of BDI-based systems is their ability of replanning during the actual plan execution to avoid the outdated reasoning results due to the environment change that happened before or during the execution. The feature makes these systems very well-suited for dynamic environments where reacting to changes and adjusting the plan is crucial. At the same time such systems are very fast which makes them appropriate for real-time or close to real-time applications.

Beliefs and their modifications are intensively tackled in many contexts. A fundamental issue is the definition of different kinds of beliefs [47, 50, 60, 70, 98, 135] together with sophisticated structures like belief sets and belief bases [61, 62, 118]. An alternative framework, belief merging, is addressed in many sources (for an overview see [78]). The authors study merging of a several belief bases in the presence of integrity constraints. The presented solutions do not allow for inconsistent belief bases which forces the authors to look for consistency preserving belief merging operators. Also, the complexity of belief merging is typically high (see [76]).

Generally speaking, when agents act in dynamic environments, belief change/revision/update/merging is inevitable, creating a multitude of problems of theoretical and applied nature [24, 78, 105]. In such cases belief update and revision are in the mainstream of the area. For representative approaches see also [66, 82, 84, 85, 92, 105] and references there. Importantly, belief revision has been found as one of the most fundamental research topics [57, 63] aiming at consistent and deterministic solutions. Among others, the well known AGM [4] model was developed as a theoretical framework for adequate belief modification practices. It inspired a large body of work over many years. For surveys see [51, 52] and references there. A significant amount of AGM extensions and improvements have been proposed [7, 34, 51, 55] , including paraconsistent ones [111, 127, 128]. Apart from undeniable profits, belief modifications can be computationally expensive, and create some other issues, like underdetermination (inability to determine rules to be defeated). In the case of group beliefs, like in teamwork, the situation becomes even more complex [47].

When dealing with dynamic environments, it is good to have some kind of protection against unwanted belief states - especially when belief updates are automatically generated without any supervision from a human operator. Here integrity constraints appear especially important. Actually, the idea of constraints is not new in information systems (see, e.g., [33, 69, 77, 109, 115]), where a distinction between hard and soft constraints might be desirable [97]. Hard constraints cannot be

violated while soft ones are flexible and often considered as preferences whose violation should be avoided as long as possible.

In real-world applications, beliefs are contextual, and affected socially, psychologically and emotionally. Some, like "do not harm", are hardly mutable but others, like "avoid slippery surfaces", meant as an indication, are flexible. In fact, known theories of belief update/change/revision/merging do not distinguish between the rigid and transient beliefs. However, in everyday activities we temporarily adjust our beliefs to specific situations with no intention to change them radically. Such a shallow change, not requiring a deeper revision, has been addressed in [21].

The essential problem of belief modeling and formation when information is partly missing and inconsistent, is addressed, e.g., in [29–31, 42–44, 74, 90, 101, 110, 111, 126]. To adequately model beliefs and reasoning about them, *epistemic profiles* and *belief structures* have been introduced in [43, 45], where paraconsistent and paracomplete belief bases are used for transformations of initial raw beliefs into more abstract, mature ones. Belief dynamics is approached there via epistemic profiles permitting to model both beliefs related to states of the environment and deliberative processes of agents.

Finally, since the inception of knowledge representation and planning, beliefs have usually been modeled via various combinations of multi-modal logics [47, 50], non-monotonic logics [87, 99], probabilistic reasoning [129] or fuzzy reasoning [137], just to mention some of them. However, most of those approaches either lack tools for handling incomplete and/or inconsistent information or are too complex for real-world applications.

1.2. Classical action planning

Action planning has been an important part of artificial intelligence since its very beginning. The ability of specifying only a desired final environment state together with available actions and letting the planner figure out the steps needed to achieve the goals gives, without any doubt, outstanding possibilities in designing advanced systems in both theoretical and practical manners. For example, it is useful in designing autonomous rescue robots capable of making their own decisions out in the difficult area without connection to an operator.

Therefore, not surprisingly, an autonomous era is materializing just in front of our eyes with a lot of researchers actively digging into new possibilities of making devices smarter. Self-driving cars are gradually making their way into casual traffic, smart assistants are driving our everyday agendas, businesses are using advanced action planners to reach their goals. Each of theses inventions at some point derive from the original concept of action planning.

Although the perfect solution has not been yet found, the research resulted with a wide range of specialized planning engines. The mostly known system that is also believed to be the first major one is STRIPS (*Stanford Research Institute Planning System*) which was introduced in 1971 [54]. The language was highly influenced by a GPS (*General Problem Solver*) software from 1961 [100] but addressed some issues that GPS had. For example, it solved the frame problem [94] by a "STRIPS assumption" (stating that all literals not mentioned in delete set carry over from the before-action to the after-action state) and introduced action representation. The language and planning engine were a leading standard for classical planning for over a decade but eventually it became too limited in terms of expressiveness. This created a space for an ancestor - ADL (*Action Definition Language*) which was introduced in [103]. The new language improved its predecessor by allowing the action effects to be conditional, adding indirect effects to actions (static laws) and dividing actions themselves into static and dynamic ones [58]. Moreover, while STRIPS supported only positive literals in the states of planning, the ADL allowed for negative literals as well. Finally, when knowledge base is concerned, the ADL applied a concept of *Open World Assumption* (OWA) which meant that

truth value of every fact not entailed from the database is assumed to be $\mathfrak{u}$. This approach is opposite to STRIPS one which implemented *Closed World Assumption* (CWA) where every such element was assumed to be $\mathfrak{f}$. Finally a PDDL (*Planning Domain Definition Language*) language has been proposed in [95] which initially was designed as a planning-syntax standardization to be used during the First International Planning Competition. The language itself is based on the ADL and its syntax is actually very similar. This is not surprising as PDDL is not a single planning engine but rather a family of requirements for which different planning engines can comply. In particular STRIPS and ADL comply with some parts of PDDL. This is why the PDDL family of languages is very broad. Just to name some (except PDDL itself), there is a NDDL [56] - New Domain Definition Language - which uses timelines and activities instead of states and actions, MAPL [26] - Multi-Agent Planning Language - which introduces multi-valued state-variables and modal operators or PPDDL [136] - Probabilistic PDDL - which uses probabilistic effects for planning purposes. It is worth noticing that PDDL is still being actively used with version 3.0 officially released in [59] which introduced planning constraints and the most recently released version 3.1 [65, 79, 80].

Classical planning plays undoubtedly a very significant role in the development of this AI branch. Up to now, modern languages include some version of "precondition", "add" and "delete" sections to their action's schemas which makes these concepts somewhat universal. As the time passes, more advanced and complicated environments were required to be emulated which included also an incomplete and inconsistent knowledge (further abbreviated by $3i$) sources. Classical engines were not prepared for such requirements calling for some new approaches.

1.3. Evolution of planning for 3i cases

Planning with incomplete knowledge is currently a broad area of research with examples of publications including, among others, [49, 73, 114] (see also references there). Theoretical work introduces many interesting concepts of incomplete knowledge representation - starting from "simple" applications of OWA to internal knowledge bases and finishing on advanced data representations like databases containing a set of potentially true worlds with a representation of a real world among them [8, 13, 25, 112, 130]. Of course, the drawback of such an approach is that planner does not know which presentation of a world is correct and therefore must perform planning in each representation independently - and this results in exponential growth of knowledge. Some other works focus on a much faster solution of representing unknown data with finite sets of first order formulas and treating actions as transformations working on these formulas rather than on actual world representation [107, 108].

Approaches like described above still at some point use sets of ground literals (typically having classical two-valued truth values assigned) as the main knowledge storage. At the same time some planning systems capable of representing uncertainty tend to use other ways of data representation, e.g., Bayesian networks (used for the first time in the CONVINCE expert system [75]). Such approaches are well-described for example in [102, 119].

As for the inconsistency, for a long time researchers were actually focusing on omitting and removing inconsistencies from knowledge bases. At some point, however, the paraconsistency started to become gradually less "avoided" and currently the area is broad. The change has started back in 1989 with [10] and since then paraconsistency is used in both multi-agent systems [1, 2, 6, 138] and in the planning itself [71]. One can also find a number of articles about handling inconsistency in knowledge bases - see [72] and references there. Several approaches have been proposed but the one most appealing to us is a usage of so called QC logic (quasi-classical) which in fact is a classical logic with additional truth values (in the mentioned articles it is *true*, *false*, *both* and *unknown* - please note a

direct correspondence with [12]).

Even though the area of research in $3i$ planning is broad, most of the existing solutions tend to be based on extending classical planners mentioned earlier. For example, BURIDAN [81] introduces probability distribution over initial world states, CGP (*Conformant Graphplan*) [121] applies a smart Graphplan algorithm to multiple world representations or CNLP (*Conditional Non-linear Planner*) [104] uses non-linear planning for managing foreseen uncertainties. Moreover, the majority of contemporary planners model unknown and/or inconsistent knowledge using the two-valued classical logic which is a simplification from our point of view. Of course approaches which introduce multi-valued logics for expressing incomplete and paraconsistent knowledge exist (with examples of already mentioned QC logic, [9] which uses Łukasiewicz logic for planning purposes or [86] which extends SAT-solver with a multi-value logic) but it seems that this branch of research is a bit less expanded which is visible in "related work" parts of papers in that area.

1.4. More actions using composite actions

Complex (composite) actions refer to action definitions containing other actions using constructions known from programming languages [96].

Starting at early 1980s, plans have been thought of not only as single sequences of actions, but also as acyclic graphs with some actions executed in parallel and others perhaps sequentially. An early approach applying this idea is SIPE (System for Interactive Planning and Execution Monitoring [131]) with later successor SIPE-2 [132]. This planning system explicitly supports parallelism and conditional actions. However, the support of incomplete and inconsistent knowledge is, in our opinion, not well developed yet in SIPE-based systems recognizing general uncertainty of information represented by action's likeliness-of-success parameter.

Since then, parallel actions have been investigated in many works, e.g., in [113] (determining which actions can be executed in parallel), [83] (developing a planning architecture with parallel action executions) or [48] (supporting parallel actions prepared especially for IPC-4 planning contest). A comprehensive approach to temporal action specification based on Temporal Action Logic (TAL) has been developed in a series of papers – see, e.g., [35–37]. TAL-based composite actions with constraints are investigated in [38].

1.5. Research questions addressed in the thesis

Since the beginning of my PhD studies our goal was to enrich the 4QL language with planning capabilities. Initially, an approach of extending the inter4QL interpreter with some classical planning methods was selected and used in first projects [20, 23, 125] (with myself being the author of the planning software). As the project was strongly based on multi-agent scenarios, soon it became clear that more advanced planning methodologies should be applied to improve expressiveness. As this was a time of intensified work on beliefs, belief bases, epistemic profiles and belief structures (with these entities being exactly designed for such scenarios) the choice was made to try to use them instead of simple sets of ground literals. Moreover, to our best knowledge, no such other paraconsistent and paracomplete belief planning solution was, and still is, available which makes it a perfect candidate for a PhD research area.

Therefore the following set of research problems has been identified:

- selecting a belief state representation appropriate for planning in the context of incomplete and/or inconsistent information;
- expanding the original 4QL language with means for reasoning on and querying beliefs in a tractable manner;
- introducing a constraint mechanism for beliefs to be used during planning by extending the original definition of belief bases;
- providing a tractable belief change formalism applicable for dynamic and complex environments;
- specifying action syntax and semantics involving state representation by using paracomplete and paraconsistent belief bases;
- extending the language of action specification to allow composite actions;
- using the composite actions for specifying plan templates (useful in complex planning problems);
- implementing the theoretical results by extending the inter4QL interpreter.

One of the important goals has been to retain tractability whenever possible. In particular, this concerns a new formalization of belief bases and querying them together with their transformations by actions.

This thesis addresses problems listed above starting with the achieved results overview presented in the next section.

1.6. Discussion of achieved results

The results obtained in the thesis are strongly rooted in a history of action planning and belief change described earlier in the chapter. This section discusses the results in the context of presented research area background. Due to the fact that details of the results are yet to be described in further parts of the dissertation, we will not provide too much details here but rather place the obtained results in the appropriate context.

1.6.1. Constrained belief bases

Our approach is based on an understanding of belief bases as presented in [43, 45]. First of all, one defines *worlds* as finite sets of ground literals. Then *belief bases* are finite sets of worlds. The intuition here is that each world is a separate representation of the observed reality. While this sounds like the approach that has already been described as alternative world sets, our understanding of the worlds does not specify anything about their relations - these can be alternative worlds or complementary representations of the same world (e.g. from different cameras or detectors). We give a complete freedom of modeling in this matter.

Using this understanding, we expand the definition of belief bases by adding a finite set of formulas of the underlying logic which can be then interpreted as constraints of the belief base. In the thesis we will use a direct correspondence between a "world representation" in a belief base, a set of ground

literals, the 4QL's well-supported model and finally, a module description in the the 4QL syntax. In fact, each of these elements is just a different representation of the same underlying concept. Such an expansion of the theoretical definition of belief bases had therefore to result in an expansion of 4QL's syntax itself to ensure the expressiveness required for defining belief bases in 4QL. While constraints themselves are not novel in the AI area, to our best knowledge binding them directly to a belief base together with conditioning answers to queries based on constraints violation was novel in [21].

We also define a $\mathrm{Bel}()$ operator which can be used for querying belief bases for beliefs which are common to each world representation. This sounds similar to a ϕ operator from [107] but actually is more subtle. While ϕ operator only verifies whether a formula is true in every element of a belief base, the $\mathrm{Bel}()$ operator calculates the least upper bound (wrt 4QL's information ordering) of actual valuations from each world representation. This allows, among others, for deducing inconsistencies in situations where conflicting data is present in two separate worlds which is very useful in many situations.

Finally, we specify a lightweight *belief shadowing* operator capable of altering agent's beliefs without actually changing its knowledge base. Such an operator corresponds to suspending some agent's beliefs rather than changing them permanently which is a common case in multi-agent scenarios where cooperation is included. What is worth emphasizing here, constraints of belief bases are also affected by the shadowing. For this purpose belief base's constraint set is divided into two parts - rigid and flexible constraints. While the rigid ones are not overridable, flexible constraints can be disregarded with shadowing operator. This reflects real life scenarios where in some cases people are willing to give up some of their beliefs while keeping the others unchanged.

1.6.2. Actions over belief bases

As will be discussed in Chapter 6, the syntax of action definition we propose is, as in many other approaches, inspired by the original STRIPS/ADL action schema with classical preconditions together with add and remove sets. However, in our case the formulas can contain constructs from a $\mathrm{4QL}^{\mathrm{Bel}}$ language which provides not only native support for non-classical truth values but also additional operations on belief bases. This allows us to live with inconsistencies and unknown data as first-class citizens and react to them accordingly as they are expected and welcome.

While in most cases actions act on worlds (sets of ground literals), in our case actions act on the whole belief bases which makes each action a transformer from one set of worlds into another. Note that in this case planning can be performed taking into that account multiple worlds at the same time. The drawback of multiple planning necessity for each world (mentioned in Section 1.3) is of course also present. However, in our approach we do not just use world representations for incomplete knowledge purposes but rather treat each world representation as a valid data chunk which potentially brings more information about the actual environment.

Finally, we propose an application of composite actions idea to our planning framework with three possible types of such actions: parallel, sequential and conditional ones. Such constructs not only allow for easier specification of actions but actually make some problems solvable within our approach. For example, a problem of table raising where both table sides have to be raised at the same time, perhaps by two different robots, not to drop a glass standing on top of the table can be solved only by parallel raising of both table sides. Each sequential execution will fail as one side of the table will be raised first and the glass will fall. Of course such a limitation can be omitted when a single robot can perform the action using a specification of an action which internally rises both sides of the table at the same time. It is not so easy when the action is beyond the capabilities of a single robot. In addition, having to explicitly specify some actions sounds like a less convenient idea than having a solution

for composing already-declared actions (especially, that such mechanism has a built-in verification of constraints for executed sub-actions and appropriate mechanics for correct knowledge base update).

Moreover, composite actions blend well into a research branch connected with plans generation based on plan templates [133, 134]. Using the syntax of combining atomic actions, one can, in fact, create a template which will later be applied by the planner to a specific situation. Using such a functionality, the planning complexity can be significantly reduced what is important in action planning systems.

1.6.3. Planner - theory in practice

Apart from theoretical results, we are also proposing an experimental implementation of the planner which serves as a proof of concept and has been used for verification of examples provided within the thesis. The experimental character of the implementation has to be emphasized as no explicit advanced optimization methods were applied to the algorithms for action planning. Our goal was to provide a reliable ACTLOG planner generating correct plans for problems specified by a user with a help of the extended expressiveness of our action specification language (even at the expense of the calculation times).

Several algorithms are typically used in planning implementations, e.g., forward (progressive) state-space search, backward (regressive) relevant-states search, A* algorithm, Graphplan algorithm etc. (for a comprehensive introduction to action planning see [116]). In the current version of the planner a forward state-space search is implemented what can later be expanded with more advanced algorithms.

Also, the literature puts a high pressure on heuristics which should be used during state space searching. The planner currently contains five exemplary heuristics. The first two of them are working locally without using the actual problem goal which, surprisingly, sometimes leads to the improved performance. The remaining three heuristics base their computations on the truth of the actual planning goal which directs the whole planning process to satisfying the goal (which, in fact, not always turns out to be the best strategy) - see heuristics details in Chapter 7 and planning experiments in Chapter 8.

Although a competition with other planning solutions is not our goal, some comparisons with existing planning solutions are provided. However, we focus on unique features of our solution rather than compare general planning performance.

The planner included into the inter4QL interpreter is open-source and available at *http://4ql.org/inter4qlPhD.html*.

1.7. Thesis structure overview

The thesis is structured as follows:

- Chapter 2 - an introduction to a background theory:
  - A reminder of the original concepts of belief bases and belief structures.
  - A theoretical background of the 4QL language.
- Chapter 3 - reasoning on belief bases with $4QL^{Bel}$:
  - An introduction of $4QL^{Bel}$, an extension of 4QL, capable of querying belief bases.
- Chapter 4 - belief shadowing:

  - A belief bases extension with integrity constraints.
  - An introduction of a shadowing operator and its semantics.
  - A presentation of belief shadowing properties.
- Chapter 5 - an implementation of beliefs:
  - Details of inter4QL internals.
  - Practical examples of the reasoning about beliefs.
- Chapter 6 - an introduction to ACTLOG:
  - An introduction of atomic and composite actions.
  - Tractability of the approach.
- Chapter 7 - an implementation of the planner:
  - Technical details of inter4QL including the implementation of actions and planner's overview.
- Chapter 8 - experiments using the planner:
  - Comparison of our planner with existing planning solutions.
  - Discussion of planning heuristics.
  - Example of complex "real-life" planning scenario.
- Chapter 9 - summary and conclusions.

Chapter 2

Original 4QL and belief bases

The material presented in the chapter originates from the external literature [42, 88, 90, 124] and is just a reminder of necessary information needed to make the thesis as self-contained as possible. As a theoretical background for $4QL^{Bel}$ language is analogical to the one for 4QL, suitably modified fragments of [18, 19] are used, too.

2.1. Under the hood of 4QL

2.1.1. Overview of 4QL's syntax

The 4QL language [88, 90, 124], is a four-valued rule language designed for reasoning and querying paraconsistent and paracomplete knowledge bases. Apart from providing firm foundations for paraconsistent knowledge bases and non-monotonic reasoning, it opens the space for a diversity of applications. For surveys of closely related areas see, e.g., [16, 64]. A unique feature of the 4QL-based language family is the presence of truth values $\mathfrak{t}$ (*true*), $\mathfrak{f}$ (*false*), $\mathfrak{i}$ (*inconsistent*), $\mathfrak{u}$ (*unknown*) as well as an unrestricted use of negation in both conclusions and premises of rules while retaining intuitive results and tractable query evaluation. Though the full definition of 4QL is available in [88, 91], for clarity we recall the most important constructs of the language.

```
1 module moduleName:
2 |   domains: ...
3 |   relations: ...
4 |   rules: ...
5 |   facts: ...
6 end.
```

Module 1: Syntax of 4QL modules.

The 4QL program consists of modules, structured as shown in Module 1. Sections **domains** and **relations** are used to specify domains and signatures of relations used in rules. 4QL rules, specified in the section **rules** have the following form, where $\langle Formula \rangle$ is an arbitrary formula of the logic presented in Section 2.1.2:

$$\langle Literal \rangle \text{ :- } \langle Formula \rangle . \tag{2.1}$$

Facts, specified in the **facts** section, are rules with the empty $\langle Formula \rangle$ part (being $\mathfrak{t}$). In such cases

we simply write $\langle Literal \rangle$. A rule of the form (2.1) is "fired" for its ground instantiations when the truth value of $\langle Formula \rangle$ is $\mathfrak{t}$ or $\mathfrak{i}$.[1] As the effect:

$$\begin{array}{l} \text{– } \langle Literal \rangle \text{ is added to the set of conclusions when the truth value of} \\ \langle Formula \rangle \text{ is } \mathfrak{t}; \end{array} \tag{2.2}$$

$$\begin{array}{l} \text{– } \langle Literal \rangle \text{ and } \neg\langle Literal \rangle \text{ are added to the set of conclusions when} \\ \text{the truth value of } \langle Formula \rangle \text{ is } \mathfrak{i}. \end{array} \tag{2.3}$$

Looking at definitions (2.2) and (2.3) it is worth noting, that the semantics of ' :– ' is formalized in the 4QL-based languages by a generalization of the Shepherdson's implication [120] rather than by the $\rightarrow$ connective formalized in (2.7) (see [124]). The implication $\rightarrow$ is more suitable for evaluating formulas while the former one reflects rule evaluation principles (2.2)–(2.3). To define the semantics of ' :– ' we use ordering $\leq_s$ which is a reflexive and transitive closure of $\mathfrak{f} = \mathfrak{u} < \mathfrak{t} < \mathfrak{i}$.

The implication $\rightsquigarrow$, corresponding to ' :– ', is defined by:

$$(A \rightsquigarrow B)(L, v) \stackrel{\text{def}}{=} A(L, v) \leq_s B(L, v). \tag{2.4}$$

When the set of truth values is restricted to $\{\mathfrak{t}, \mathfrak{f}, \mathfrak{u}\}$ or $\{\mathfrak{t}, \mathfrak{f}, \mathfrak{i}\}$, the implication $\rightsquigarrow$ is the three valued implication of [120]. The semantics of rules is given by:

$$(C :\!\!- B)(L, v) \stackrel{\text{def}}{=} B(L, v) \rightsquigarrow C(L, v). \tag{2.5}$$

Modules are primarily used for assuring a structural form of a base of rules. If m is a module name, $m.A$ expresses a *reference to* m. Semantically, one can view relation symbols within a module m as (implicitly) extended by prefix '$m.$'. In order to maintain a clear semantics and tractability, a certain form of acyclicity of references is required, close in spirit to stratification in logic programming and deductive databases [3] but concerning formulas with the operator '$\in T$' rather than negation.

Definition 1 Let $M = \{m_1, \ldots, m_k\}$ ($k \geq 1$) be a set of modules. By a *reference graph* of M we understand a graph $\langle M, E \rangle$ such that for $m_i, m_j \in M$:

$$(m_i, m_j) \in E \text{ iff rules in } m_i \text{ contain a subformula of the form } m_j.\alpha.$$

If $(m_i, m_j) \in E$ then we say that module m_i *refers to* module m_j. ◁

Definition 2 By a 4QL *program* we understand a set of 4QL modules whose reference graph is acyclic. ◁

Acyclicity of a reference graph of any 4QL program can be determined in deterministic polynomial time wrt the size of the program. We also assume *strong typing* in the following sense:

- when a value occurs as an argument of a relation, it has to belong to the domain associated with that argument;
- when a variable occurs in a rule as an argument in more than one place, all such arguments have to be specified as belonging to the same domain;
- a domain consists of all values of all modules occurring as relations' arguments specified as belonging to that domain.

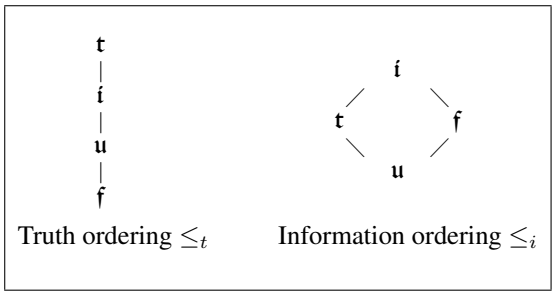

Figure 2.1: Orderings on truth values.

2.1.2. Logic behind 4QL

In the syntax of the underlying logic we assume truth constants $\mathbf{t}$, $\mathbf{f}$, $\mathbf{i}$ and $\mathbf{u}$, propositional connectives $\neg, \vee, \wedge, \rightarrow$ and operators $A \in T$, $A = t$, where A is a formula, $T \subseteq \{\mathbf{t}, \mathbf{f}, \mathbf{i}, \mathbf{u}\}$, $t \in \{\mathbf{t}, \mathbf{f}, \mathbf{i}, \mathbf{u}\}$.

To define semantics of the logic let us start with *truth ordering* (see Figure 2.1) on truth values, denoted by $\leq_t$, being the reflexive and transitive closure of ordering: $\mathbf{f} < \mathbf{u} < \mathbf{i} < \mathbf{t}$.[2] Truth ordering is used to evaluate formulas.

Information ordering $\leq_i$ reflects the process of gathering and fusing information. It is used to merge results concerning the same (perhaps negated) ground literal concluded from different rules, possibly originating from different information sources. Starting from the lack of information, in the course of belief acquisition, evidence supporting or denying hypotheses is collected, leading finally to a decision about its truth value.

For $t_1, t_2 \in \{\mathbf{f}, \mathbf{u}, \mathbf{i}, \mathbf{t}\}$, the semantics of $\neg, \wedge, \vee, \rightarrow$ is given by:

$$\neg\mathbf{f} \stackrel{\text{def}}{=} \mathbf{t},\ \neg\mathbf{u} \stackrel{\text{def}}{=} \mathbf{u},\ \neg\mathbf{i} \stackrel{\text{def}}{=} \mathbf{i},\ \neg\mathbf{t} \stackrel{\text{def}}{=} \mathbf{f}; \tag{2.6}$$

$$t_1 \wedge t_2 \stackrel{\text{def}}{=} \min\{t_1, t_2\};\ \ t_1 \vee t_2 \stackrel{\text{def}}{=} \max\{t_1, t_2\};\ \ t_1 \rightarrow t_2 \stackrel{\text{def}}{=} \neg t_1 \vee t_2; \tag{2.7}$$

where $\min, \max$ are the minimum and maximum wrt $\leq_t$.[3]

We assume that *Const* is a fixed finite set of constants, *Var* is a fixed set of variables and Rel is a fixed set of relation symbols. Following the convention, we begin variable names with upper case letters and constants with lower case letters.

Definition 3 A *literal* is an expression of the form $R(\bar{\tau})$ or $\neg R(\bar{\tau})$, with τ being a sequence of arguments, $\bar{\tau} \in (\mathit{Const} \cup \mathit{Var})^k$, where k is the arity of R. *Ground literals over Const*, denoted by $\mathcal{G}(\mathit{Const})$, are literals without variables, with all constants in *Const*. If $\ell = \neg R(\bar{\tau})$ then $\neg\ell \stackrel{\text{def}}{=} R(\bar{\tau})$. For an *assignment* $v : \mathit{Var} \longrightarrow \mathit{Const}$ and a literal ℓ, by $\ell(v)$ we understand the ground literal obtained from ℓ by substituting each variable X occurring in ℓ by constant $v(X)$. ◁

Definition 4 The *truth value* of a literal ℓ wrt a set of ground literals w and an assignment v, denoted

[1] That is, the value of $\langle Formula \rangle$ contains some truth.

[2] For motivations behind $\leq_t$ see, e.g., [5, 124].

[3] To justify equalities in (2.6) involving non-classical truth values, let us note that whenever the value of a formula is unknown (respectively, inconsistent), the value of its negation is unknown (respectively, inconsistent), too. The usage of minimum and maximum operations is justified in [5, 88, 124]. Such a semantics appears to be natural. It also reflects intuitions of classical two-valued logic when restricted to classical truth values $\mathbf{t}$, $\mathbf{f}$.

Let v : *Var* $\longrightarrow$ *Const* be an assignment, m be a $3i$ set of ground literals. $\min, \max$ are respectively minimum and maximum wrt truth ordering and A, B be formulas. Then:

- $t(m,v) \stackrel{\text{def}}{=} t$ for $t \in \{\mathsf{f}, \mathsf{u}, \mathsf{i}, \mathsf{t}\}$;
- $(\neg A)(m,v) \stackrel{\text{def}}{=} \neg(A(m,v))$;
- $(A \odot B)(m,v) \stackrel{\text{def}}{=} A(m,v) \odot B(m,v)$, for $\odot \in \{\wedge, \vee, \rightarrow\}$;
- $\big(\forall x\!:\!D(A(x))\big)(m,v) \stackrel{\text{def}}{=} \min\{\big((a \in D) \rightarrow A(a)\big)(m,v) \mid a \in \textit{Const}\}$;
- $\big(\exists x\!:\!D(A(x))\big)(m,v) \stackrel{\text{def}}{=} \max\{\big((a \in D) \wedge A(a)\big)(m,v) \mid a \in \textit{Const}\}$;
- $(A \in T)(m,v) \stackrel{\text{def}}{=} \begin{cases} \mathsf{t} & \text{when } A(m,v) \in T; \\ \mathsf{f} & \text{otherwise;} \end{cases}$
- $(A = t)(m,v) \stackrel{\text{def}}{=} (A \in \{t\})(m,v)$;

Table 2.1: Semantics of 4QL formulas.

Truth values	Logic
$\{\mathsf{t}, \mathsf{f}\}$	Classical propositional logic
$\{\mathsf{t}, \mathsf{u}, \mathsf{f}\}$	Kleene three-valued K_3
$\{\mathsf{t}, \mathsf{i}, \mathsf{f}\}$	K_3 with Priest's interpretation of the third truth value as i

Table 2.2: The relation of logic behind 4QL to other logics.

by $\ell(w,v)$, is defined as follows:

$$\ell(L,v) \stackrel{\text{def}}{=} \begin{cases} \mathsf{t} & \text{if } \ell(v) \in w \text{ and } (\neg\ell(v)) \notin w; \\ \mathsf{i} & \text{if } \ell(v) \in w \text{ and } (\neg\ell(v)) \in w; \\ \mathsf{u} & \text{if } \ell(v) \notin w \text{ and } (\neg\ell(v)) \notin w; \\ \mathsf{f} & \text{if } \ell(v) \notin w \text{ and } (\neg\ell(v)) \in w. \end{cases} \quad \triangleleft$$

Having established the truth value of a single literal ℓ, the definition can be now extended to all formulas which is done in Table 2.1.

Finally, when the logic we consider is restricted to connectives $\neg, \wedge, \vee, \rightarrow$ and projected onto two- or three-valued calculi, it becomes one of well-known logic – see Table 2.2. This is well justified by the fact that Kleene three-valued logic K_3 is the standard choice for interpreting the third truth value as u. K_3 is also a standard choice for interpreting the third value as i. Indeed, assuming that the reality is consistent, the value i can also be seen as an indicator of a lack of knowledge: we have contradictory claims that a given property is both t and f but it is (perhaps temporarily) unknown which claim is actually the right one.

To complete the understanding of 4QL's constructs, Table 2.3 provides correspondences between logical syntax presented in this section and the syntax used in inter4QL interpreter.

2.1.3. Semantics of 4QL (well-supported models)

The semantics of 4QL modules is given by *well-supported models* as formalized in [91]. A set of ground literals is a *model* of a module m if all rules of m, understood as implications (2.5), are true in the model. Intuitively, a model is well-supported when it consists of ground literals (if any) assuming that all literals it contains are conclusions of reasoning starting from facts. For any module,

Logical syntax	Syntax of inter4QL
$\mathbf{t}$ $\mathbf{f}$ $\mathbf{i}$ $\mathbf{u}$	true false incons unknown
$\neg$ $\wedge$ $\vee$	- , |
$\in T$	in T
$=$ $\leq$ $\geq$	math.eq math.leq math.geq
$<$ $>$	math.lt math.gt

Table 2.3: Correspondences between logical syntax and the inter4QL syntax.

its corresponding well-supported model is uniquely determined [89–91]. Namely, each module can be treated as a finite set of ground literals and this set can be computed in deterministic polynomial time [88, 90]. Note that well-supportedness does not entail minimality. This is an intended feature since in many contexts minimality is not desired [39, 53, 88, 117, 122].

To simplify the presentation, in the current section we assume that all literals are ground and all universal (existential) quantifiers are substituted by conjunctions (disjunctions) of formulas. In particular, rather than allowing rules with variables, we consider their ground and quantifier-free instances. The definition provided here is adopted from [90], where also related intuitions are discussed.

The difficulty in generating well-supported models for 4QL programs depends on deriving conclusions on the basis of facts being temporarily true or false and later becoming inconsistent. We iterate the following method until no new conclusions are generated:

– generate the least set of conclusions by Datalog-like reasoning;
– retract conclusions based on defeated premises;
– correct (minimally) the obtained set of literals to make all facts and rules true.

The following definitions realize this method,where by $Pos(S)$ (respectively, $Pos(L)$) we understand the 4QL program (respectively, the set of literals) obtained from S (respectively, from L) by replacing each negative literal $\neg\ell$ by ℓ', where ℓ' is obtained from ℓ by changing relation symbol of ℓ by a fresh relation symbol (for simplicity denoted by its primed symbol).

Let us start with the case of modules not referring to any other modules.

Definition 5 Let S be a set of (ground) rules. Then gen^S is obtained from the least model of $Pos(S)$ by substituting literals of the form ℓ' with $\neg\ell$. For and X, Y being sets of literals, we define:

$$\begin{aligned}
&\delta^S_X(Y) \stackrel{\text{def}}{=} Y \cup \{\ell, \neg\ell \mid \text{ there is a rule } `\ell :\!- \beta.' \in S \text{ such that } \beta(X \cup Y) = \mathbf{i} \\
&\qquad\qquad\qquad\qquad \text{and } \ell(X \cup Y) \neq \mathbf{i}\},\\
&\Delta^S(X) \stackrel{\text{def}}{=} \text{the least fixpoint of } \delta^S_X,\\
&correct^S(X) \stackrel{\text{def}}{=} X \cup \Delta^S(X).
\end{aligned}$$

◁

For X being a set of ground literals, let $incons(X) \stackrel{\text{def}}{=} \{\ell, \neg\ell \mid X(\ell) = \mathbf{i}\}$.

Definition 6 Let S be a set of (ground) rules. By the set of *pre-consequences of a set of literals X of S*, denoted by $Pre^S(X)$, we understand the set of literals defined by:

$$Pre^S(X) \stackrel{\text{def}}{=} correct^S\big(incons(X) \cup gen^{T(X)}\big),$$

where $T(X) \stackrel{\text{def}}{=} S - \{\varrho \in S \mid concl(\varrho) \in incons(X)\}$. ◁

Definition 7 For any set S of (ground) rules, the well supported model of S, denoted by W^S, is defined by $W^S = \bigcup_{i>0}(Pre^S)^i(gen^S)$, where:

$$(Pre^S)^i(X) \stackrel{\text{def}}{=} \begin{cases} Pre^S(X) & \text{when } i = 1; \\ Pre^S((Pre^S)^{i-1}(X)) & \text{when } i > 1. \end{cases}$$ ◁

Let us now consider the case when modules of a 4QL program refer to other modules. Since the reference graphs of 4QL programs are acyclic, the computation of well-supported models can be organized by performing the following steps until well-supported models of all modules are computed:

1. compute the well-supported models of modules not referring to other modules;
2. replace formulas involving '$\in$' by truth values in all modules referring to those with the already computed well-supported models.[4]

Let us emphasize that well-supportedness requires the Open World Assumption (OWA): all conclusions have to be explicitly inferred. This is an opposite to the Closed World Assumption (CWA), where conclusions which are not inferred are assumed to be false. In the 4QL family of languages, a literal is false when its negation is a consequence of a rule. Otherwise it may be unknown. However, one can easily (partially or totally) close the world.

2.1.4. Computational complexity of 4QL

While considering complexity issues, the following theorems can be presented and proven as originally presented in [88–90]. Having a 4QL program P, by $\#D$ we denote the sum of the sizes of all domains of P and by $\#P$ we denote the number of modules in P.

In the thesis by complexity we understand data complexity - see [3].

Theorem 1 *For every 4QL program P, well-supported models of its modules can be computed in deterministic polynomial time in* $\max\{\#D, \#P\}$. ◁

Theorem 2 *The querying problem for 4QL has deterministic polynomial time complexity, i.e., for every 4QL program P and formula α, the set of all tuples satisfying α in the well-supported model of P can be computed in deterministic polynomial time in* $\max\{\#D, \#P\}$. ◁

Theorem 3 *4QL captures deterministic polynomial time over linearly ordered domains. That is, every polynomially computable query (wrt the size of the database domain) can be expressed in 4QL with modules and external literals, assuming that a linear order on the database domain is available.* ◁

2.2. Belief bases and belief structures

Having reminded the most important principles of the 4QL language, let us now recall structures accompanying reasoning and querying of beliefs (as they will be used extensively across the rest of the thesis). Let us start with belief bases, which is the most basic concept.

A bottom concept underlying our approach to belief bases, initiated in [43, 45], is that of a world. Recall, that the worlds are sets of ground literals (i.e., variable-free atomic formulas) representing

[4] Note that due to the acyclicity of the reference graph, new modules not referring to other modules are obtained in this step.

feasible states of affairs. For example, an accident witness could report a victim's shallow breathing and leg injury, not being sure how serious the injury is: 3 or 4, in a given scale. In this case, the following two worlds may represent patient's conditions, where integers from 0 to 9 represent the severity degree:

$$\{\text{symptom(victim, leg, injury, 3)}, \text{symptom(victim, breathing, shallow, 6)}\}, \tag{2.8}$$

$$\{\text{symptom(victim, leg, injury, 4)}, \text{symptom(victim, breathing, shallow, 6)}\}. \tag{2.9}$$

Gathering the worlds (2.8) and (2.9) together, we obtain a belief base $\big\{(2.8), (2.9)\big\}$ representing the two alternatives. It can be augmented with information about the heart failure and its severity:

$$\big\{(2.8), (2.9), \{\text{symptom(victim, heart, failure, 7)}\}\big\}. \tag{2.10}$$

Note that worlds can represent alternatives and/or add new information, perhaps originating from another sources. However, belief base designers do not indicate whether a world provides an alternative or augments other worlds. This implicitly follows from the worlds' contents.

In many papers, e.g., related to the AGM theory of belief revision (for survey see [105]),[5] a belief base consists of a set of formulas of the underlying logic, not necessarily closed on consequences. Here we do not consider the consequence relation. By restricting formulas to ground literals we are able to use querying machinery assigning truth values to the results. This allows us to obtain a tractable framework also when we allow rules, making the specification of belief bases more uniform and concise.

As we use truth values $\mathbf{t}$, $\mathbf{f}$, $\mathbf{i}$ and $\mathbf{u}$, a belief base becomes a structure capable of storing beliefs originating from nondeterministic environments. Belief bases are systems' passive components reacting on requests and queries via a suitable query processing engine.

Let *Const* be a fixed finite set of constants. If S is a set then $\textsc{Fin}(S)$ denotes the set of all finite subsets of S. By $\mathbb{C}(\mathit{Const}) \stackrel{\text{def}}{=} \textsc{Fin}(\mathcal{G}(\mathit{Const}))$ we denote the set of all finite sets of ground literals over the set of constants *Const* (for definition of ground literal see Definition 3).

Definition 8 By a $3i$*-world* over *Const* (world with incomplete and/or inconsistent information) we shall understand a finite set of ground literals with all constants belonging to *Const*. ◁

Definition 9 By a *belief base over a set of constants Const* we understand any finite set $\Delta = \{m_1, \ldots, m_k\}$, where $k \geq 1$ and for $i = 1, \ldots, k$, m_i is a $3i$-world over *Const*. In particular, any finite set $\Delta \subseteq \mathbb{C}$ is a belief base. ◁

Each $3i$-world m_i in a belief base represents a feasible or augmenting (perhaps still incomplete and/or inconsistent) view of the world. For example, a belief base can consist of following three $3i$-worlds: one containing facts based on measurements received from a ground robot's sensor platform, the second containing facts interpreting video streams from a drone's camera while the third representing views provided by ground operators.

Sets constituting belief bases can be seen as a four-valued generalization of Kripke-like possible worlds or impossible worlds [110].

Now let us recall belief structures and epistemic profiles as introduced in [42–44].

[5] AGM is an acronym referring to names of originators of the theory: C. Alchourrón, P. Gärdenfors and D. Makinson [4].

Definition 10

- By a *constituent* we understand any set $C \in \mathbb{C}$;
- by an *indeterministic epistemic profile* we understand any function $\mathcal{E}$ of the sort $\textsc{Fin}(\mathbb{C}) \longrightarrow \textsc{Fin}(\mathbb{C})$;
- by an *indeterministic belief structure over an indeterministic epistemic profile* $\mathcal{E}$ we mean $\mathcal{B}^{\mathcal{E}} = \langle \mathcal{C}, \mathcal{F} \rangle$, where:
  - $\mathcal{C} \subseteq \mathbb{C}$ is a nonempty set of constituents;
  - $\mathcal{F} \stackrel{\text{def}}{=} \mathcal{E}(\mathcal{C})$ is the set of *consequents* of $\mathcal{B}^{\mathcal{E}}$. $\triangleleft$

Belief structures represent a complex paraconsistent and paracomplete knowledge representation structure for agents. In short, epistemic profiles encapsulate agents' or groups' reasoning, perception and communication capabilities, including context-specific methods of both information disambiguation and completion. To make the approach as general as possible, epistemic profiles are mappings transforming one belief base (*constituents*) into another belief base (*consequents*). While constituents contain sets of beliefs acquired by perception, expert-supplied knowledge, communication and other ways, consequents contain final, "mature" beliefs or beliefs capturing an updated view on the environment. Together with epistemic profiles, constituents and consequents constitute belief structures. Importantly, belief structures ensure a unified approach to both individual and group reasoning [46]. They have been already applied in argumentation [41] and modeling complex dialogues [40], where their use led to a simplified and tractable framework.

Chapter 3

Reasoning about beliefs - 4QL$^{\text{Bel}}$

This chapter is based on [18] and uses extensive fragments of the article in an unchanged form.

Let us now present 4QL$^{\text{Bel}}$, a rule-based four-valued query language introduced in [18] capable of paraconsistent and paracomplete doxastic reasoning with belief bases and belief structures. The language extends 4QL (described in details in Section 2.1). The use of this particular query language was suggested in [42–44].

As mentioned in Section 2.2, belief bases consist of finite sets of ground literals what creates a natural correspondence between 4QL modules and belief bases, thus also between 4QL and belief structures. However, the original concept of 4QL does not include belief operators nor other constructs useful in doxastic reasoning. Therefore this chapter defines the syntax and semantics of 4QL$^{\text{Bel}}$ by introducing constructs for reasoning about beliefs, belief bases and belief structures. We also rephrase the tractability results of computing and querying well-supported models for 4QL$^{\text{Bel}}$ programs from the analogical result for 4QL.

In order to offer a more flexible formalism, the chapter also extends the definition of semantics of formulas given in [44] by belief operators. Roughly speaking, in addition to referring to belief structures using belief operators, references to their belief bases (that is their constituents and consequents) are allowed.

3.1. Base logic meets belief bases

The 4QL$^{\text{Bel}}$ language inherits most of its features from 4QL (in particular its extended version 4QL^{+} of [124]) - including underlying logic. For common definitions please see Section 2.1.2 while newly introduced base logic's syntax is shown below in Table 3.1. Most of the formulas presented there are common to both languages but the following new constructs are introduced:

- $\text{Bel}_{\Delta}\big(\langle Formula\rangle\big)$, expressing beliefs rooted in belief bases (indicated by Δ);
- $f(\Delta).\alpha$, allowing one to evaluate $\text{Bel}\big()$-free formulas in belief bases: here f is a mapping transforming a belief base into a single set of ground literals, e.g., $f(\Delta)$ may be $\bigcup_{D\in\Delta} D$ or $\bigcap_{D\in\Delta} D$ (further denoted by $\bigcup\Delta$, $\bigcap\Delta$, respectively).

In general, f occurring in $f(\Delta).()$ is an arbitrary information fusion method, intended as a means to combine information included in $3i$-worlds of Δ.

$$
\begin{array}{rcl}
\langle Formula \rangle & ::= & \langle Literal \rangle \mid \neg \langle Formula \rangle \mid \langle Formula \rangle \wedge \langle Formula \rangle \mid \\
& & \langle Formula \rangle \vee \langle Formula \rangle \mid \langle Formula \rangle \rightarrow \langle Formula \rangle \mid \\
& & \forall X \, \langle Formula \rangle \mid \exists X \, \langle Formula \rangle \mid \\
& & \langle Formula \rangle \in \langle TruthValues \rangle \mid \\
& & \mathrm{Bel}_{\Delta}\big(\langle Formula \rangle \big) \mid f(\Delta).\langle Formula \rangle
\end{array}
$$

where:

- $\langle Literal \rangle$ represents the set of literals;
- $\langle TruthValues \rangle$ represents nonempty subsets of $\{\mathtt{t}, \mathtt{f}, \mathtt{i}, \mathtt{u}\}$;
- Δ is a belief base (as formalized in Definition 9),
- f is a mapping transforming a belief base into a (single) finite set of ground literals; if f is not specified, by default $f(\Delta) \stackrel{\text{def}}{=} \bigcup \Delta$, i.e., $\Delta.\alpha \stackrel{\text{def}}{=} (\bigcup \Delta).\alpha$.

Table 3.1: Syntax of the base logic for 4QL$^{\text{Bel}}$.

3.2. The 4QL$^{\text{Bel}}$ Language

3.2.1. Syntax of 4QL$^{\text{Bel}}$

Just like in 4QL, 4QL$^{\text{Bel}}$'s basic components are modules. The syntax of the module can be seen in Table 1 (Page 21) in this thesis. Moreover, 4QL$^{\text{Bel}}$ also shares the rule format in following form:

$$\langle Literal \rangle \text{ :- } \langle Formula \rangle . \tag{3.1}$$

Note that formulas at the right-hand side of :- can now, among others, have the form $f(\Delta).\alpha$. In cases when Δ consists of a single set of ground literals represented by a module m, we often write $m.\alpha$ rather than $\{m\}.\alpha$. However, just like in case of 4QL, these references cannot include cycles. The definition for the reference graph has been already provided for the 4QL language in Definition 1 as part of Section 2.1. The definition of a 4QL$^{\text{Bel}}$ program can be therefore specified analogously to the 4QL's one as presented in Definition 11.

Definition 11 By a 4QL$^{\text{Bel}}$ program we understand a set of 4QL$^{\text{Bel}}$ modules whose reference graph is acyclic. ◁

3.2.2. Semantics of 4QL$^{\text{Bel}}$

The semantics of 4QL$^{\text{Bel}}$ modules (just like 4QL ones) is given by well-supported (as described in 2.1.3). A $3i$-world is a model of a module m if all rules of m, are true in the model. Moreover, each 4QL$^{\text{Bel}}$ module uniquely specifies its well-supported model, so it can be identified with a $3i$-world. That way:

$$\begin{array}{l} \text{4QL}^{\text{Bel}} \text{ modules have a very important role as a tool for} \\ \text{concise and uniform specification of } 3i\text{-worlds.} \end{array} \tag{3.2}$$

Indeed, when facts of a module are updated, its well-supported model is changed accordingly. Therefore, rather than list all facts constituting a $3i$-world, one can provide rules reflecting the way in which

- $(\mathrm{Bel}_\Delta(t))(v) \stackrel{\text{def}}{=} t$, for $t \in \{\mathbf{t}, \mathbf{f}, \mathbf{i}, \mathbf{u}\}$;
- $\big(\mathrm{Bel}_\Delta(\alpha)\big)(v) \stackrel{\text{def}}{=} \textsc{Lub}\{\alpha(D, v) \mid D \in \Delta\}$;
- $(f(\Delta).\alpha)(v) \stackrel{\text{def}}{=} \alpha(f(\Delta), v)$,

where:
- Δ is a belief base;
- $v : \mathit{Var} \longrightarrow \mathit{Const}$ is an assignment of constants to variables;
- α is a first-order formula (for nested $\mathrm{Bel}()$ s, one starts with the innermost one.)
- LUB denotes the least upper bound wrt the information ordering (see Figure 2.1).

Table 3.2: Semantics of the $\mathrm{Bel}()$ and $f()$. operators.

derived facts are obtained on the basis of given facts. For example, facts in a module may indicate patient's symptoms while rules may provide diagnoses. Taking the principle (3.2) into account, from now on modules can appear wherever $3i$-worlds are allowed, in particular as elements of Δ.

The semantics of propositional connectives, quantifiers is given in Table 2.1. For for a first order formula a and a belief base Δ, we define $\alpha(\Delta, v) \stackrel{\text{def}}{=} \alpha(\bigcup \Delta, v)$. The semantics of doxastic operators is given in Table 3.2.

3.2.3. Complexity of 4QL$^{\text{Bel}}$

Complexity theorems for 4QL$^{\text{Bel}}$ are analogous to the ones for 4QL presented in Section 2.1.4 and can be proved analogously as results for 4QL [88–90]. For readability, however, we place the theorems in a rephrased form here.

The intuition behind the complexity results for 4QL$^{\text{Bel}}$ is as follows. Let us note that worlds, represented either directly as sets of a ground literals or indirectly as a 4QL$^{\text{Bel}}$ programs, are computable in a polynomial time which results from Theorem 1 presented for the 4QL language. Using Theorem 2 one can note, that the same polynomial time is also achieved in the case of queries to such worlds. As the $\mathrm{Bel}()$ operator is a series of world queries, the results of these queries have to be merged to compute its result. One can note, that such operation is also deterministically polynomial with respect to the number of worlds as size of each world is polynomial as well. Therefore the entire $\mathrm{Bel}()$ operator can be computed in a polynomial time.

When considering a 4QL$^{\text{Bel}}$ program P, by $\#D$ we denote the sum of the sizes of all domains of P and by $\#P$ we denote the number of modules in P.

Theorem 4 *For every 4QL^{Bel} program P, well-supported models of its modules can be computed in deterministic polynomial time in* $\max\{\#D, \#P\}$. ◁

Theorem 5 *The querying problem for 4QL^{Bel} has deterministic polynomial time complexity, i.e., for every 4QL^{Bel} program P and formula α, the set of all tuples satisfying α in the well-supported model of P can be computed in deterministic polynomial time in* $\max\{\#D, \#P\}$. ◁

Theorem 6 *4QL^{Bel} captures deterministic polynomial time over linearly ordered domains. That is,*

every polynomially computable query to a belief base (therefore, to a belief structure, too) can be expressed in 4QL^{Bel}. ◁

3.3. Specifying Belief Structures in 4QL$^{\mathbf{Bel}}$

To implement belief structures using 4QL$^{\text{Bel}}$ we simply encode constituents and consequents by 4QL$^{\text{Bel}}$ modules and query them using, among others, the $\mathrm{Bel}()$ operator indexed by constituents and consequents, respectively. Epistemic profiles can be defined by auxiliary 4QL$^{\text{Bel}}$ modules or by rules directly included in modules specifying consequents.

In order to evaluate $\mathrm{Bel}()$-free formulas in belief structures one can use formulas indexed by constituents or consequents, as needed. To illustrate the idea let us consider robots equipped with temperature and pressure sensors together with a camera. In this case beliefs can be formed on the basis of robots' perception resulting from sensor measurements and interpretations of camera images.

A natural belief structure for this simple scenario is $\mathcal{B}$ with constituents represented by modules {`press`, `temp`, `cam`} shown in Figure 3.1, where (identifying modules with their well-supported models):

- `press` gathers measurements of pressure sensors, `press`={p(4.0), p(4.6)};
- `temp` gathers measurements of temperature sensors, `temp`={t(85), t(82)};
- `cam` gathers information extracted from camera, `cam`={p(smoke), ¬p(smoke)}.

```
1 module press:
2 | domains:
3 | | float pvalue.
4 | relations:
5 | | p(pvalue).
6 | facts:
7 | | p(4.0).
8 | | p(4.6).
9 end.
```

```
1 module temp:
2 | domains:
3 | | integer tvalue.
4 | relations:
5 | | t(tvalue).
6 | facts:
7 | | t(85).
8 | | t(82).
9 end.
```

```
1 module cam:
2 | domains:
3 | | literal detection.
4 | relations:
5 | | p(detection).
6 | facts:
7 | | p(smoke).
8 | | ¬ p(smoke).
9 end.
```

Figure 3.1: Sample constituents.

The $\mathcal{B}$'s epistemic profile is encoded by rules in modules `f1` and `f2` shown in Figure 3.2. The well-supported models of these modules are consequents of $\mathcal{B}$. They represent two alternative views on the world: `f1`={danger()} and `f2`={danger(),¬danger()}.

```
module f1:
    relations:
        danger().
    rules:
        danger() :–
            ∃ X(press.p(X)∧ X≥ 4.3)
        ¬danger() :–
            ∀ X(temp.t(X)→ X≤ 60).
end.
```

```
module f2:
    relations:
        danger().
    rules:
        danger() :–
            cam.p(fire)∨cam.p(smoke).
end.
```

Figure 3.2: Sample consequents and rules defining an epistemic profile.

Now, for example $\mathrm{Bel}_{\{\texttt{f1},\texttt{f2}\}}(\mathit{danger}) = \mathfrak{i}$ and $\{\texttt{press}, \texttt{temp}\}.(t(10) \wedge \neg p(4.6)) = \mathfrak{f}$ (the latter because $\bigcup\{\texttt{press}, \texttt{temp}\} = \{p(4.0), p(4.6), t(85), t(82)\}$).

Chapter 4

Belief shadowing and constraints - 4QL$^{\text{Bel+}}$

This chapter is based on [21] and uses extensive fragments of the article in an unchanged form.

When beliefs are flexible or change frequently, it hardly makes sense to adjust the entire belief base accordingly. A better choice is to suspend them for the time being. The belief interference addressed in this chapter is a potentially transient swap of beliefs called *belief shadowing* introduced originally in [21]. For example, when two belief bases participate in reasoning, one of them may turn out to be more important or up to date. Then, the conflicting part of the "weaker" base may be shadowed by the "stronger" one. With such phenomena we deal frequently during teamwork and other forms of cooperation. Individuals joining a group are expected to accept the group beliefs and suspend their conflicting ones. Therefore, a swap from individual to group beliefs, and then perhaps back, is needed. Importantly, the ability of shadowing rather than updating, revising or merging beliefs can result in a substantial improvement of the performance of agent systems relying on doxastic reasoning what is particularly important from the systems' engineering point of view.

As a lightweight form of belief change, shadowing may introduce inconsistencies. In the approach inconsistencies are first class citizens, so much heavier belief revision, often meant as a remedy for inconsistencies, is not required. Another phenomenon of realistic environments, calling for an attention, is the unavoidable information incompleteness. Therefore, both paraconsistent and paracomplete reasoning is needed in the spirit of [68].

Belief bases represent snapshots of the environment and the agents' mindsets, both evolving over time. An exemplary high level agents' belief bases architecture is summarized in Figure 4.1, where, for $1 \leq l \leq k$, $\mathcal{B}_l$ are belief bases, $m_{l1}, \ldots, m_{lr_l}, n_{l1}, \ldots, n_{ls_l}$ are 4QL$^{\text{Bel}}$ modules, $w_{l1}, \ldots, w_{lr_l}$ are the $3i$-worlds being respectively well-supported models of modules $m_{l1}, \ldots m_{lr_l}$, solid lines represent queries among modules, dotted lines represent correspondences between modules and $3i$-worlds, and dashed lines represent agent's queries. Note that agents may use multiple belief bases, some of which may be private, some owned by groups, and some may be available to all agents. Query manager may be a 4QL$^{\text{Bel}}$ interpreter or another database querying engine.

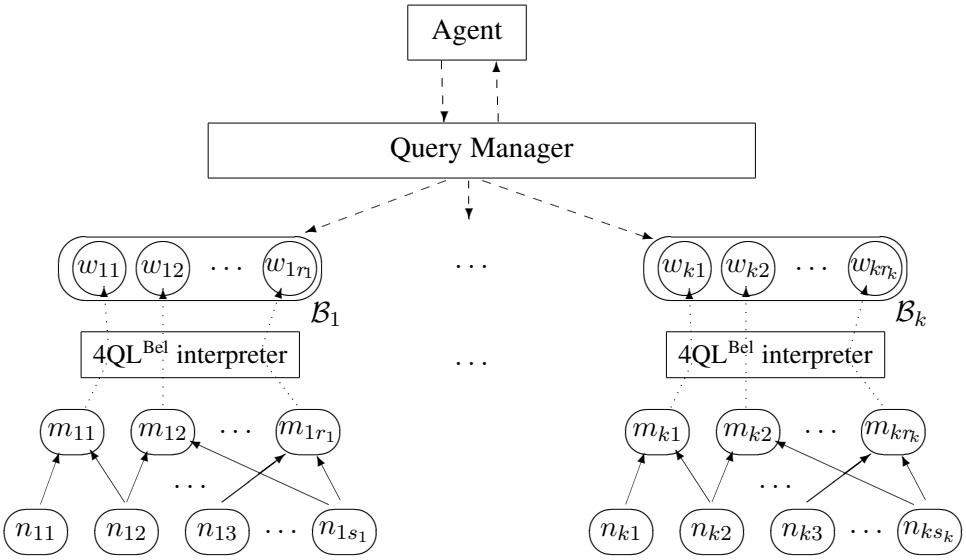

Figure 4.1: High level agent's belief bases architecture.

In AI systems the already mentioned evolution of environment and the agents' mindset should be supervised, especially when rules are machine learned or data mined in a human-free manner. Achieving this is possible by introducing integrity constraints. Though they are well-known in database systems (see Section 1.2), their shadowing is novel, pertaining admissible modes of behavior at desired abstraction levels, in a semantically meaningful manner.

The belief shadowing framework is *lightweight*: (i) the shadowing operator is efficient, (ii) does not require changes in the belief bases involved, and (iii) does not assume agents' familiarity with details of other agents' belief bases. The requirement (i) is crucial for systems' performance and for efficient swapping between contexts in which different shadowings apply. The requirements (ii) and (iii) are vital in cooperation/teamwork of heterogeneous agents designed by separate parties.

4.1. Belief bases with constraints

Similarly to 4QL$^{\text{Bel}}$, the idea of a world is a bottom concept underlying our approach to belief base. Let us remind that worlds are sets of ground literals (i.e., variable-free atomic formulas) representing feasible states of affairs. In order to formally define belief bases with constraints, we extend the Definition 9 by assuming that constraints are their inherent parts.

Definition 12 Let *Const* be a fixed finite set of constants. By a *belief base over a set of constants Const* we understand any pair $\mathcal{B} = \langle \Delta, \mathcal{C} \rangle$ consisting of:

- $\Delta = \{m_1, \ldots, m_k\}$, where $k \geq 1$ and for $i = 1, \ldots, k$, m_i is a $3i$-world (as defined in Definition 8);
- $\mathcal{C}$, called *constraints* of $\mathcal{B}$, being a finite set of (universally closed) formulas of the underlying logic, which are true in Δ (for formal definitions, see Section 2.1.2 and Section 4.2). ◁

Each m_i ($i = 1..k$) in a belief base represents a feasible or augmenting (perhaps still incomplete and/or inconsistent) view of the world.

4.2. Logic behind 4QL$^{\text{Bel}}$ extended for constraints

The syntax of the underlying logic is based on the one from 4QL therefore for common definitions please see Section 2.1.2. Just like before we assume truth constants $\mathbf{t}$, $\mathbf{f}$, $\mathbf{i}$ and $\mathbf{u}$, propositional connectives $\neg, \vee, \wedge, \rightarrow$, quantifiers $\forall, \exists$, operators $A \in T, A = t$, where A is a formula, $T \subseteq \{\mathbf{t}, \mathbf{f}, \mathbf{i}, \mathbf{u}\}$, $t \in \{\mathbf{t}, \mathbf{f}, \mathbf{i}, \mathbf{u}\}$ and belief operator $\mathrm{Bel}_{\mathcal{B}}(A)$ where $\mathcal{B}$ is a belief base.

Let v : *Var* $\longrightarrow$ *Const* be an assignment, $\mathcal{B} = \langle \Delta, \mathcal{C} \rangle$ be a belief base, m be a $3i$-world and A be a formulas. Then:

- $A(\Delta, v) \stackrel{\text{def}}{=} A\big(\bigcup_{m \in \Delta} m\ , v\big)$;
- $\big(\mathrm{Bel}_{\mathcal{B}}(A)\big)(v) \stackrel{\text{def}}{=} \begin{cases} \textsc{Lub}\{\ \ A(m, v) \,|\, m \in \Delta\} \text{ when} \\ \qquad\quad \text{for all } C \in \mathcal{C},\ C(\Delta, v) = \mathbf{t}; \\ \mathbf{u} \qquad \text{otherwise;} \end{cases}$
- $A(\mathcal{B}, v) \stackrel{\text{def}}{=} \begin{cases} A(\Delta, v) & \text{when for all } C \in \mathcal{C}, C(\Delta, v) = \mathbf{t}; \\ \mathbf{u} & \text{otherwise.} \end{cases}$

Table 4.1: Constraint-ready semantics of $\mathrm{Bel}()$-free formulas in 4QL$^{\text{Bel}}$.

We may now proceed to the final definition of semantics of $\mathrm{Bel}()$-free formulas in 4QL$^{\text{Bel}}$- see Table 4.1. The semantics of operations on belief bases has been extended with actual checks of constraint violation. Like in previous analogical definitions, LUB is the least upper bound wrt information ordering defined in Figure 2.1 (see Page 23). Observe that C, being a constraint, does not contain free variables, so v in $C(\Delta, v)$ is redundant. In similar cases we will use $C(\Delta)$ rather than $C(\Delta, v)$, and $C(\mathcal{B})$ rather than $C(\mathcal{B}, v)$.

4.3. Discussion on integrity constraints - 4QL$^{\text{Bel+}}$

Though belief bases, as introduced in Section 4.1, contain integrity constraints, their use in our framework deserves further discussion. In particular, belief shadowing (introduced in the Section 4.5), similarly to belief revision or update, may result in creating undesirable conclusions. To avoid such risky cases one could construct some specific rules but a much better idea is to formulate a general integrity constraints which will not require remembering about adding specific rules.

As mentioned in Section 1.2, a concept of integrity constrains is intensively discussed in the literature which often separates them to hard ones (which cannot be violated by any means) and soft ones (which are allowed to be violated in some situations). In our case a distinction between non-shadowable ("hard") and shadowable ("soft") constraints also appears useful. To avoid terminological misunderstandings we shall further call them *rigid* and *flexible* ones, respectively.

In rule languages constraints are typically expressed by rules with empty heads expressing what is disallowed. Dually, in our approach constraints express what should always be true. The separation between constraints and rules gives the former ones an axiom-like flavor. Constraints can be specified in modules and in belief bases in two subsections, separating rigid and flexible ones, $\mathcal{C} = \mathcal{C}_R \cup \mathcal{C}_F$. In inter4QL constraints are defined as shown in Module 2.

By 4QL$^{\text{Bel+}}$ we will denote the rule language obtained from 4QL$^{\text{Bel}}$ by allowing the constraints section.

```
1 module abc:
2 |  constraints:
3 |  |  rigid: ...
4 |  |  flexible: ...
5 end.
```

Module 2: Constraints specification.

The distinction between rigid and flexible constraints does not affect the semantics of $\mathcal{C}_R$ and $\mathcal{C}_F$ unless they appear in the context of the shadowing operator (see Section 4.5). Importantly, we require constraints to be true (cf. the last item of Table 4.1). This might seem restrictive in a four-valued framework. However, formulas of the form '$A \in T$' can be used, so requirements like "A is true or inconsistent" can easily be expressed by '$A \in \{\mathbf{t}, \mathbf{i}\}$' being $\mathbf{t}$ when the truth value of A is in the set $\{\mathbf{t}, \mathbf{i}\}$.

Though specified within modules, constraints may be local (limited to a single module) and global (span over multiple modules). Local constraints refer solely to relations in the same module. Accordingly, global constraints can contain literals referring to multiple, perhaps all, modules as long as references do not create cycles in the reference graph. Technically, to avoid cycles, additional modules/belief bases can be created as containers for constraints. Such additional structures can be viewed as being "above" modules referenced by non-local constraints.

We extend Definition 1 from Section 2.1 to deal with constraints as follows:

Definition 13 By the *reference graph* of a set Π of modules with constraints we mean the reference graph for Π seen as a 4QL$^{\text{Bel}}$ program (disregarding constraints), augmented with edges from m to n whenever constraints of m contain a reference to n, i.e., a subexpression of the form 'n.'. ◁

Definition 14 By a 4QL$^{\text{Bel+}}$ program we mean a set of 4QL$^{\text{Bel}}$ modules with constraints such that its reference graph (in the sense of Definition 13) is acyclic and all constraints in modules are true. ◁

Note that for every 4QL$^{\text{Bel+}}$ program Π, well-supported models of Π's modules do exist and, as in the case of 4QL$^{\text{Bel}}$ programs, are uniquely determined. For a 4QL$^{\text{Bel+}}$ module m, by $wsm(m)$ we denote the well-supported model of m. Using this correspondence between modules and well-supported models, being themselves $3i$-worlds, we can specify any belief base $\mathcal{B} = \langle \Delta, \mathcal{C} \rangle$ by:

$$\mathcal{B} = \langle m_1, \ldots, m_k \rangle \,, \tag{4.1}$$

where $m_1, \ldots, m_k$ are 4QL$^{\text{Bel+}}$ modules. In such a case,

- $\Delta \stackrel{\text{def}}{=} \{wsm(m_1), \ldots, wsm(m_k)\}$;
- $\mathcal{C}$ consists of all rigid and flexible constraints collected from $m_1, \ldots, m_k$.

To simplify notation, we often identify single modules with belief bases, assuming that:

$$\text{module } m \text{ represents the belief base } \langle m \rangle. \tag{4.2}$$

Of course, specifications of belief bases of the form (4.1) inherit all advantages of rule-based specifications. In particular, comparing to Definition 12, 4QL$^{\text{Bel+}}$-based specifications are typically much more concise and easier to understand and maintain.

4.4. New syntax constructs in 4QL$^{\text{Bel+}}$

The 4QL$^{\text{Bel+}}$ language, implemented in inter4QL, inherits a fair amount of elements from 4QL, including basic program syntax and semantics. It extends the language with a significant elements for doxastic reasoning, though. This section will cover the newly introduced parts.

First of all, so far we have not defined any syntax for declaring belief bases. This can be done now as the final definition of a belief base has been already introduced. The inter4QL's syntax which should be used for specifying belief bases is specified in Figure 4.2.

```
1 beliefs beliefBaseName:
2 |  constraints:
3 |  |  // see Section 4.3 ...
4 |  worlds:
5 |  |  // list of 4QL^Bel modules specifying 3i-worlds
6 end.
```

Figure 4.2: Syntax of belief bases.

An important feature of belief bases related to the syntax of 4QL$^{\text{Bel+}}$ language is that domains of their worlds (see Section 2.1 for introduction of domains) become their domains (a belief base is internally treated as a simple module - in particular it also contains domains), accessible in their constraints. If, in different worlds, a domain 'dom' appears – the corresponding belief base's domain, 'dom', is the union of all domains 'dom' appearing in the belief base's worlds (assuming the same types of domain elements).

Mostly for purposes of expressing constraints, 4QL$^{\text{Bel+}}$'s implementation introduces an explicit possibility to use quantifiers and implication in expressions. Table 2.3 provides correspondences between logical syntax used in Section 4.2 and the syntax used in the inter4QL. The table can be now extended to match new constructs introduced in 4QL$^{\text{Bel+}}$ as presented in Table 4.2.

Logical syntax	Syntax of inter4QL
$\rightarrow$	->
$\forall x$: dom	forall X: dom
$\exists x$: dom	exists X: dom
$\mathrm{Bel}_{\mathcal{B}}(\,)$	Bel[$\mathcal{B}$]( )
$f(\mathcal{B}).(\,)$	$\mathcal{B}$.( )

Table 4.2: Correspondences between logical syntax of 4QL$^{\text{Bel+}}$ and the inter4QL syntax extending Table 2.3.

4.5. The Shadowing Operator

To avoid semantical complexity, we treat shadowing as a formal expression rather than a belief base. However, to simplify presentation, syntactically we treat such formal expressions as belief bases. Thus, slightly abusing notation, we allow them to occur in the $\mathrm{Bel}()$ operator. Belief shadowing is defined by $\mathrm{Bel}_{\mathcal{B}_1 \,\mathrm{as}\, \mathcal{B}_2}(A)$ intuitively returning $\mathrm{Bel}_{\mathcal{B}_2}(A)$ when it is $\mathbf{t}$, $\mathbf{i}$ or $\mathbf{f}$, or $\mathrm{Bel}_{\mathcal{B}_1}(A)$, when $\mathrm{Bel}_{\mathcal{B}_2}(A)$ is $\mathbf{u}$. However, suitable constraints have to be validated. If they are not, $\mathrm{Bel}_{\mathcal{B}_1 \,\mathrm{as}\, \mathcal{B}_2}(A)$ returns $\mathbf{u}$ for any query A.

Belief shadowing, denoted by 'as', is a left-associative operation. That is,

$$\mathcal{B}_1 \mathtt{as}\, \mathcal{B}_2 \mathtt{as} \mathcal{B}_3 \stackrel{\text{def}}{=} (\mathcal{B}_1 \mathtt{as}\, \mathcal{B}_2) \mathtt{as}\, \mathcal{B}_3.$$

To define belief shadowing we need an auxiliary operator $\rtimes$ allowing one to fuse beliefs. Let, in (4.3) and Definitions 15, 16, $\mathcal{B}^1 = \langle \Delta^1, \mathcal{C}^1_R \cup \mathcal{C}^1_F \rangle$ and $\mathcal{B}^2 = \langle \Delta^2, \mathcal{C}^2_R \cup \mathcal{C}^2_F \rangle$ be belief bases. Then:

$$\mathrm{Bel}_{\mathcal{B}_1 \rtimes \mathcal{B}_2}(A) \stackrel{\text{def}}{=} \begin{cases} \mathrm{Bel}_{\mathcal{B}_2}(A) & \text{when } \mathrm{Bel}_{\mathcal{B}_2}(A) \in \{\mathbf{t}, \mathbf{f}, \mathbf{i}\}; \\ \mathrm{Bel}_{\mathcal{B}_1}(A) & \text{when } \mathrm{Bel}_{\mathcal{B}_2}(A) = \mathbf{u}. \end{cases} \tag{4.3}$$

We are now ready to define integrity constraints and belief shadowing, $\mathcal{B}_1 \mathtt{as}\, \mathcal{B}_2$, the central concepts of our approach.

Definition 15 By *integrity constraints of* $\mathcal{B}_1 \mathtt{as}\, \mathcal{B}_2$ we understand the set $\mathcal{C}^1_R \cup \mathcal{C}^2_R \cup \mathcal{C}^2_F$ with $\mathcal{C}^1_R \cup \mathcal{C}^2_R$ being *rigid constraints* and $\mathcal{C}^2_F$ being *flexible constraints of* $\mathcal{B}_1 \mathtt{as}\, \mathcal{B}_2$. ◁

Definition 16 The *belief operator* over belief base $\mathcal{B}_1$ *shadowed by* belief base $\mathcal{B}_2$, $\mathrm{Bel}_{\mathcal{B}_1 \mathtt{as}\, \mathcal{B}_2}(\,)$, is defined by:

$$\mathrm{Bel}_{\mathcal{B}_1 \mathtt{as}\ \mathcal{B}_2}(A) \stackrel{\text{def}}{=} \begin{cases} \mathrm{Bel}_{\mathcal{B}_1 \rtimes \mathcal{B}_2}(A) & \text{when for any } C \in \mathcal{C}^1_R \cup \mathcal{C}^2_R \cup \mathcal{C}^2_F, \\ & \qquad C'(\mathcal{B}_1) = \mathbf{t} \\ \mathbf{u} & \text{otherwise,} \end{cases}$$

where C' is obtained from C by substituting references to $\mathcal{B}_1$ in subformulas of the form $\mathrm{Bel}_{\ldots \mathcal{B}_1 \ldots}(\ldots)$ by $\mathcal{B}_1 \rtimes \mathcal{B}_2$. ◁

Though constraints of belief bases are always true, as required in Definition 14, constraints of $\mathcal{B}_1 \mathtt{as}\, \mathcal{B}_2$ may be unsatisfied for some $\mathcal{B}_1$ and $\mathcal{B}_2$. When this occurs, we assume that any query to $\mathcal{B}_1 \mathtt{as}\, \mathcal{B}_2$ returns the empty set of tuples with the truth value $\mathbf{u}$. Note that some queries may also return $\mathbf{u}$ when constraints are satisfied but $\mathcal{B}_1 \mathtt{as}\, \mathcal{B}_2$ contains no facts supporting or denying such queries. These cases can be distinguished without recalculating constraints, e.g., using the query $\mathrm{Bel}_{\mathcal{B}_1 \mathtt{as}\, \mathcal{B}_2}(\mathbf{t})$ which returns $\mathbf{t}$ when constraints are satisfied and $\mathbf{u}$ otherwise.

4.6. Properties of Shadowing and complexity

For any belief bases, $\mathcal{B}_1$, $\mathcal{B}_2$, the operator $\mathrm{Bel}_{\mathcal{B}_1 \mathtt{as}\, \mathcal{B}_2}(\,)$ satisfies $KD45_n$ axioms [28].

Proposition 1 *For any belief bases,* $\mathcal{B}_1$, $\mathcal{B}_2$ *and formula* A,

$$\mathrm{Bel}_{\mathcal{B}_1 \mathtt{as}\, \mathcal{B}_2}(A) \rightarrow \neg \mathrm{Bel}_{\mathcal{B}_1 \mathtt{as}\, \mathcal{B}_2}(\neg A); \tag{4.4}$$

$$\mathrm{Bel}_{\mathcal{B}_1 \mathtt{as}\, \mathcal{B}_2}(A) \rightarrow \mathrm{Bel}_{\mathcal{B}_1 \mathtt{as}\, \mathcal{B}_2}(\mathrm{Bel}_{\mathcal{B}_1 \mathtt{as}\, \mathcal{B}_2}(A)); \tag{4.5}$$

$$\neg \mathrm{Bel}_{\mathcal{B}_1 \mathtt{as}\, \mathcal{B}_2}(A) \rightarrow \mathrm{Bel}_{\mathcal{B}_1 \mathtt{as}\, \mathcal{B}_2}(\neg \mathrm{Bel}_{\mathcal{B}_1 \mathtt{as}\, \mathcal{B}_2}(A)). \tag{4.6}$$

Note, however, that axioms (4.4)–(4.6) do not have the classical meaning. For example, when the truth value of A is $\mathbf{i}$, the implication (4.4) holds but it does not mean that (an inconsistent) belief in A prevents (inconsistent) belief in $\neg A$. Indeed, in this case, both $\mathrm{Bel}_{\mathcal{B}_1 \mathtt{as}\, \mathcal{B}_2}(\mathbf{i})$ and $\neg \mathrm{Bel}_{\mathcal{B}_1 \mathtt{as}\, \mathcal{B}_2}(\neg \mathbf{i})$ are $\mathbf{i}$.

Complexitywise, similarly to results in Section 3.2.3, we have the following propositions, where for any 4QL$^{\text{Bel+}}$ program P, $\#D$ denotes the sum of the sizes of all domains of P; $\#P$ denotes the number of modules in P.

The intuition for the complexity results here is quite similar to the one presented for the 4QL$^{\text{Bel}}$ language in Section 3.2. One can note, that the only difference between belief bases from 4QL$^{\text{Bel}}$ language and the ones declared for the 4QL$^{\text{Bel+}}$ are constraints and the possibility of performing a belief shadowing operation. As stated earlier, the constraints are just a formulas of 4QL$^{\text{Bel}}$ language. Therefore a verification of the constraints consists of a series of queries to worlds (possibly one world if constraints are used in a simple module). Each such query can be computed in a polynomial time as stated in the complexity results for the 4QL$^{\text{Bel}}$ language. The combination of final results requires calculating conjunction of all separate responses therefore the whole constraints handling can be computed in a polynomial time. On the other hand, the computation of the belief shadowing requires traversing of a shadowing expression and asking polynomial time queries to appropriate belief bases (or single worlds). The size of that expression is polynomial parameterized with $\#P$. Therefore, also the whole shadowing operation is computable in deterministic polynomial time.

Proposition 2 *For every 4QL^{Bel+} program P, both:*

- *checking the existence of the well-supported models of modules of P;*
- *computing well-supported models of modules of P,*

can be done in PTIME in $\max\{\#D, \#P\}$. ◁

Proposition 3 *Given belief bases $\mathcal{B}_1, \mathcal{B}_2$ expressed using modules of a 4QL^{Bel+} program P, the problem of computing queries involving expressions of the form* $\mathrm{Bel}_{\mathcal{B}_1 \, as \, \mathcal{B}_2}()$ *has deterministic polynomial time complexity in* $\max\{\#D, \#P\}$. ◁

Since shadowing is defined in terms of belief base queries, as a consequence of the corresponding result in Section 3.2.3, we have the following proposition.

Proposition 4 *Assuming that domains are linearly ordered, every polynomially computable shadowing can be expressed in 4QL^{Bel+}.* ◁

From the perspective of systems' engineering, the above complexity results are important. However, even tractability does not guarantee scalability over big data. Belief shadowing can be made horizontally scalable when recursive queries are not allowed, assuming that all 4QL$^{\text{Bel}}$ modules are already computed and belief bases consist of the resulting $3i$-worlds. In this case, $\mathrm{Bel}()$-free formulas are equivalent to first-order (and non-recursive SQL) queries which can be evaluated in a horizontally scalable manner. Queries involving belief operators can also be easily horizontally distributed (with a separate thread evaluating a given query in each $3i$-world of a given belief base).

Chapter 5

Implementation of beliefs

One of the important requirements for the thesis was bringing up all theoretical results to life and testing them in actual use cases. As the whole theoretical work was done using 4QL, the inter4QL interpreter was a natural choice.[1]

5.1. History of inter4QL

inter4QL is an interpreter of the 4QL language which had its first release in August 2011 (v1.0) as a part of M.Sc. thesis [123]. Since then it has been actively developed with bug fixes and new features implemented until v2.0 was released in August 2013 as part of my M.Sc. thesis [17]. After several minor releases mostly containing bug fixes, a release of v3.0 was made in March 2016 (see [27]). It contained a concept of *group modules* which could be considered as an early version of belief bases. Querying of this component was done by a voting method which additionally allowed for customizable voting rules (e.g., $\#incons > \#true + \#false$ which means responding i for a query if a number of inconsistent responses from modules contained in the group is higher than the number of t and f responses added together). This was a step towards information fusion rather than belief bases. The actual doxastic reasoning was introduced in October 2018 with the v4.0 release.[2] The software introduced a syntax for specifying belief bases with constraints as well as $\mathrm{Bel}()$ and belief shadowing operators. Finally, in December 2019 the inter4QL was updated to version 5.0 which included the support for defining actions and action planning. Since then the interpreter has been extended with additional planning heuristics and bug fixes resulting in the latest 5.2 version being released in April 2020. This version of the interpreter was used for verification of the examples in this thesis.

The implementation details of inter4QL 4.0 (and newer) will be discussed in further sections of the chapter.

5.2. General architecture

The inter4QL interpreter is written in C++ language and therefore is structuralized into classes with appropriate relations between them. The UML class diagram of the interpreter is shown in Figure 5.1. One can distinguish the following main parts of the interpreter:

[1] Apart from that, I am actively working on and with the interpreter for almost 7 years now so I know it pretty well.

[2] For a detailed changelog of each version please visit *http://4ql.org*

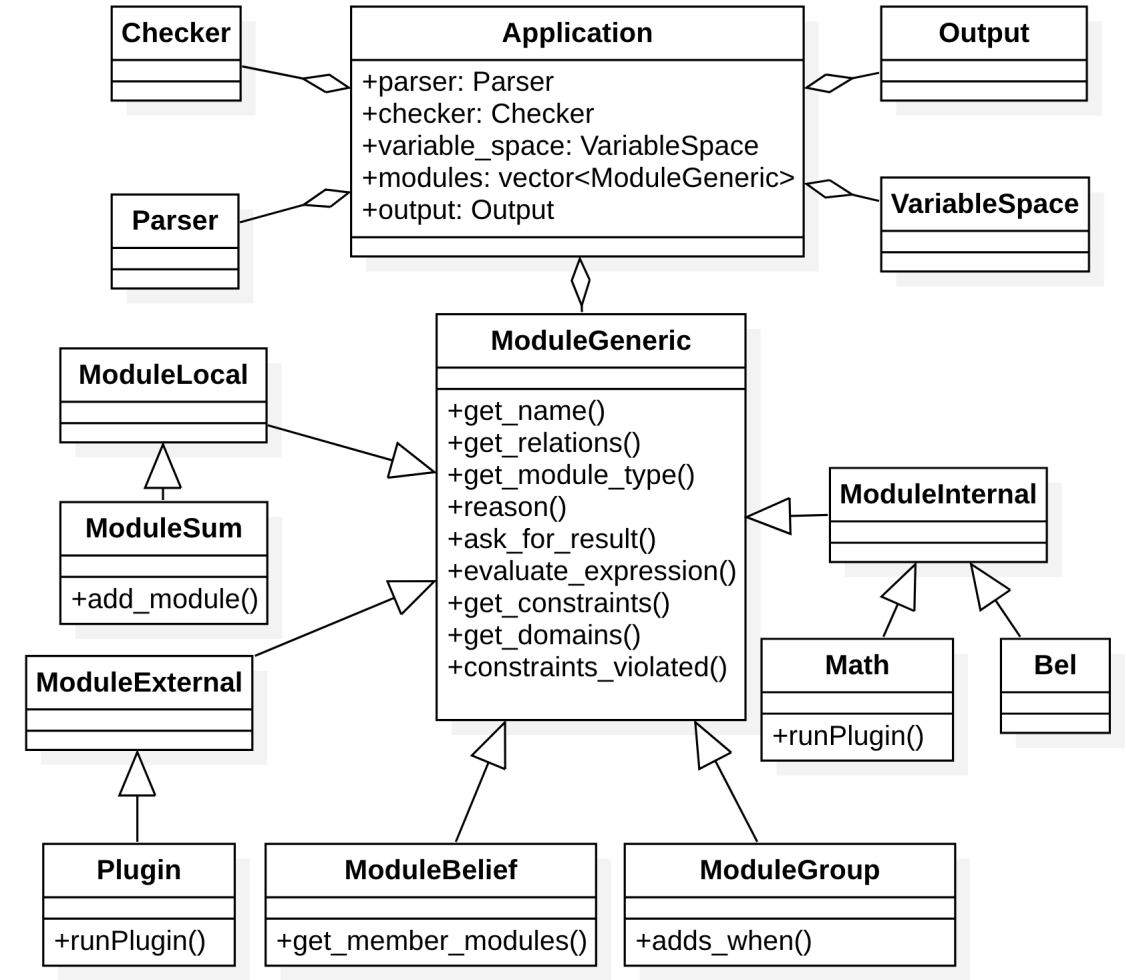

Figure 5.1: Overview of inter4QL 4.0 architecture (simplified).

- *Application* class - the main class of the interpreter containing the main loop and storing all data needed for parsing, validating queries and modules, managing the database and reasoning on it.
- *ModuleGeneric* class - an interface between Application class and multiple types of modules.
- *ModuleExternal* class - a class representing external modules e.g. plugins (separate applications implementing a given interface which can be queried from the interpreter - see [17]).
- *ModuleInternal* class - a class providing an abstraction layer for modules built into the interpreter - e.g. *Math* (providing mathematical operators) or *Bel* (internal data provider for the $\mathrm{Bel}()$ operator).
- *ModuleLocal* class - a class representing all user-defined modules declared in a $4\mathrm{QL}^{\mathrm{Bel}}$ program.
- *ModuleSum* class - a helper class used for calculating a sum of models for belief bases. It is a sub-class of a *ModuleLocal* class in order to inherit a querying mechanism (models are summed during module adding, later querying is the same as for standard local module).
- *ModuleGroup* class - a class implementing group modules described in Section 5.1.

- *ModuleBelief* class - representation of belief bases containing a list of worlds (modules) included in it and querying methods.

While looking at the schema please note, that constraints are handled at the *ModuleGeneric* level (and not *ModuleBelief*). By such a design all modules can be guarded with constraints, not only belief bases. This is in line with a statement from Section 3.2.2 saying that "[...] modules can appear wherever $3i$-worlds are allowed, in particular as elements of Δ in a belief base". In fact, each type of a module in inter4QL implements a common interface *ModuleGeneric* which allows for treating all of the modules (including belief bases) in the uniform manner.

5.3. Summary of the most important data structures and algorithms

5.3.1. Well-supported model

Well-supported models are the heart of each (local) module in inter4QL.[3] Whenever a user specifies a module, a reasoning algorithm is executed to compute such a model. The high-level algorithm of computing the *wsm* was already given in Section 2.1.3 but here we will dive a bit more into its internal implementation details. Algorithm 4 presents a pseudocode of processing a 4QL program, including a *wsm* computation.

As modules can reference each other, ordering them before reasoning is crucial. Otherwise we could evaluate a module which needs data from another module but the referenced module does not yet have a model ready - this would cause wrong results.

Deriving new literals from rules may introduce inconsistencies invalidating previously inferred data. Let us illustrate this phenomenon with an example shown in Module 3 dating back to one of the first publications about 4QL [91].[4]

```
1  module ex5:
2      relations:
3          w(literal).
4          o(literal).
5          r(literal).
6      rules:
7          w(x) :- o(x) | r(x).
8          r(x) :- w(x).
9          -o(x) :- r(x).
10     facts:
11         o(x).
12 end.
```

Module 3: Example of knowledge aggregation

If we try to simulate the Algorithm 4 on Module 3, we will start with a database filled with $\{o(x)\}$ (the only fact in the module). The first iteration of the algorithm will add $\{w(x)\}$ (using the first rule) - now the model contains two elements: $\{o(x), w(x)\}$. The second iteration will add $\{r(x)\}$ (using the second rule) and the third will add $\{-o(x)\}$ (using the third rule). The model contains

[3] Plugins and beliefs are dynamically computed so technically they do not contain a well-supported model.

[4] This example is also included as "ex5.4ql" in inter4QL package.

the following elements now: $\{o(x), -o(x), w(x), r(x)\}$. Please note that this is a model of this module - a minimal model to be exact. Each rule in the module, when evaluated with respect to this set of ground literals, obtains the truth value **t**. However, this is not a well-supported model - the value **t** assigned to $w(x)$ lost its support in the first rule as its supporting literal $o(x)$ is now inconsistent. The algorithm will therefore continue with the first rule and add $-w(x)$ to the model followed by $-r(x)$ (as $r(x)$ looses its support in the second rule). Now the model looks as follows: $\{o(x), -o(x), w(x), -w(x), r(x), -r(x)\}$ which is a fixed point of the computation. Please note, that also here all rules evaluate to **t** - but now each value assigned to literals is supported by some rule and the support chain starts from module's facts.

Input: a list l of parsed modules from a 4QL file
Result: module list out containing modules where each has a well-supported model computed

```
1  set out = [] ;
2  while l ≠ [] do
3      Select module m which is not referenced by any other module;
4      if m ≠ nullptr then
5          remove m from l;
6          set m.database = {}, database_temp = {};
7          add m.facts to m.database;
8          while database_temp ≠ m.database do
9              database_temp = m.database;
10             foreach r ∈ m.rule do
11                 // (produces possibly many valuations of
12                 // variables with assigned logic values)
13                 calculate valuations v for r.body wrt m.database;
14                 foreach val ∈ v do
15                     if val.value = t then
16                         valuate r.head with val;
17                         add valuated r.head to m.database;
18                     else
19                         if val.value = i then
20                             valuate r.head with val;
21                             add valuated r.head and −r.head to m.database;
22                         end
23                     end
24                 end
25             end
26             // here m.database is filled with newly generated literals
27             // let's store them in database_temp and continue until a fixed point is obtained
28         end
29         add m to out;
30     else
31         throw exception "Cycle in module graph";
32     end
33 end
```

Algorithm 4: Overview of implementation of well-supported model computation.

As for the *wsm* data storage, the inter4QL is using a map-based data structure. The key of the map represents a "relation name" and the value is a vector of valuations assigned to the relation. For the Module 3 the internal knowledge base looks as shown in Figure 5.2.[5]

```
Database for module ex5:

Key logic:
Key date_time:
Key date:
Key string:
Key real:
Key literal:
TRUE (x (literal);)
Key integer:
Key r:
INCONSISTENT (x (literal);)
Key w:
INCONSISTENT (x (literal);)
Key o:
INCONSISTENT (x (literal);)
```

Figure 5.2: Internal database structure for module "ex5".

One can see that the first couple of entries is occupied by domain names built into the inter4QL. This is where the interpreter gets and stores the knowledge about elements of each domain (for the discussion of domains see the next subsection). Then entries for relations in the module are kept in a form of a variable valuation and truth value. In the above example one can see the database contains the following literals: $\{o(x), -o(x), w(x), -w(x), r(x), -r(x)\}$ (which matches the example result above).

5.3.2. Domains

Domains play a very significant role in inter4QL as they not only greatly improve readability but also split all defined constants into separate sets. As the previous subsection shows, to generate a well-supported model of a module we need to compute a truth value of a rule's body which sometimes needs a couple of iterations over constants in the program. $Math$ operations are the best example here as they are computed dynamically and therefore need to be grounded. We do not want to force the user to always ground its calls to mathematical operations so we internally achieve grounding by filling variables with constants from the 4QL program. This would take a long time if all constants were just placed in one, common bag.

Please also note, that calculating *wsm* needs a couple of iterations until a fixed point in calculations is reached. Repeating such iterations over constants multiple times during the process can add up to a quite significant time - this is where domains substantially reduce the computational complexity.

For each relation in a 4QL program, the types of its attributes have to be specified. This is required for the *Checker* module to type-check all references to relations and report errors when type incompatibility occurs. Apart from some default types built into the interpreter (see Figure 5.2), users can

[5] It can be verified in inter4QL using the *debug "database" enable* command.

specify their own labeled type names based on the default ones - see Modules 5, 6.

```
1  module a:
2      domains:
3          literal module_name.
4      relations:
5          helloFromModule(module_name).
6          name(module_name).
7      rules:
8          helloFromModule(X) :- name(X).
9      facts:
10         name(a).
11         -name(b).
12 end.
```

Module 5: Exemplary module "a".

```
1  module b:
2      domains:
3          literal module_name.
4      relations:
5          helloFromModule(module_name).
6          name(module_name).
7      rules:
8          helloFromModule(X) :- name(X).
9      facts:
10         name(b).
11         -name(a).
12 end.
```

Module 6: Exemplary module "b".

Such labeled domains are stored separately from default domains (see Figure 5.3). This means that all iterations over constants will be limited to the appropriate domain only which improves the time of computation significantly.

```
Database for module a:

Key date_time:
Key date:
Key string:
Key real:
Key literal:
Key integer:
Key module_name:
TRUE (a (literal);)
Key name:
FALSE (b (literal);)
TRUE (a (literal);)
Key helloFromModule:
TRUE (a (literal);)
Key logic:
```

Figure 5.3: Internal database structure for module "a".

Figure 5.3 requires one more explanation. When we look at Module 5 then we see that two literal constants are actually used in it ("a" and "b") while in database shown in Figure 5.3 only "a" is visible. This is caused by following rules of adding an element to a domain:

- only positive literals are used for the initial filling up of the domains (please note, that inconsistent facts will also be added as they are specified with both positive and negative literals).
- while reasoning, grounded attributes of a rule's head are added to domains only if its body evaluates to either **t** or **i**.

The above assumptions were made during the design of version 2.0 of the interpreter. Intuitively they are correct - true and inconsistent facts express the presence of some positive part of the knowledge and therefore their attributes can be added to domains. Negative (and unknown) literals do not contain such a knowledge. Therefore even if containing some values "in text", these attribute values should not be included into domains as they are not "confirmed". For example, when specifying a fact that John is not a dog, we do not want John to be included into a domain of dogs.

5.4. Querying belief bases with inter4QL

As one could notice already, beliefs are handled in the interpreter by two major components :

- *ModuleBelief* class which stores a list of worlds and implements a method of querying them.
- The *Bel* class (being an internal module) which is responsible for managing logic behind the $\mathrm{Bel}()$ operator.

One of the reasons why the case is divided is querying of the belief bases.

Let us recall Table 3.1 (Page 30) which specified two ways of querying a belief base - directly using the "f" belief base transformer (which transforms a belief base into a single $3i$-world) and by using the $\mathrm{Bel}()$ operator. The same differentiation is therefore introduced into inter4QL by introducing two separated implementations of module querying. *ModuleBelief* contains the implementation of both methods (either by creating a *ModuleSum* instance, "filling it" with world modules and querying it for results or by the standard evaluation of the query in each world module and combining results) however the *Bel* class is responsible for selection of the appropriate one in a given case.

Let us look at the actual querying examples. Modules 5, 6 specified earlier can be used as worlds and Belief Base 7 declares a simple belief base containing these worlds.

```
1 beliefs c:
2 |  worlds:
3 |  |  a.
4 |  |  b.
5 end.
```

Belief Base 7: Exemplary belief "c"

Below some exemplary queries and outputs of the interpreter are presented:

```
        Interpreter for 4ql, version 5.2, http://www.4ql.org/
# import "abcExample.4ql".
Program loaded!
# c.helloFromModule(X).
== Results ==
X: a = true
X: b = true
```

In the above example the problem is loaded and direct query to a belief base is made. To compute the response, a sum of well-supported models is prepared (containing $\{name(a), -name(b), name(b), -name(a), helloFromModule(a), helloFromModule(b)\}$) and the query is evaluated in this set giving the result above. Now, let us try with $\mathrm{Bel}()$ operator:

```
# Bel[c](helloFromModule(X)).
== Results ==
X: a = true
X: b = true
```

In this case the query needs to be valuated separately with each element from the domain *c.literal* (being a sum of domains from modules "a" and "b" and containing $\{a, b\}$). This will make it a grounded expression acceptable by the $\mathrm{Bel}()$ operator (note the valuation v in Table 4.1). For $helloFromModule(a)$. we will receive $\mathfrak{t}$ from module "a" and $\mathfrak{u}$ from module "b". In an analogous way, $helloFromModule(b)$. will receive $\mathfrak{u}$ from module "a" and $\mathfrak{t}$ from module "b". In both cases $\textsc{Lub}\{true, unknown\} = true$ - hence the result above.

```
# a.name(X).
== Results ==
X: b = false
X: a = true
# b.name(X).
== Results ==
X: b = true
X: a = false
# c.name(X).
== Results ==
X: b = inconsistent
X: a = inconsistent
# Bel[c](name(X)).
== Results ==
X: b = inconsistent
X: a = inconsistent
```

Now let us look at the relation $name$. Modules "a" and "b" are locally closing the world here for the purpose of demonstration. The first two queries show that locally, in separate worlds, each world has its own view on its $name$ with explicitly denying other names. When enclosed in a belief base though (another two queries), the general belief about the $name$ is inconsistent as we are getting contradictory results from each world. These two queries also show, that in this particular case there is no difference in querying the belief base via the $\mathrm{Bel}()$ operator and directly, using summed worlds. However, it it not true in the general case.

```
# (c).(name(a) | name(b)).
== Results ==
inconsistent
# Bel[c](name(a) | name(b)).
== Results ==
true
# exit
Thanks for using!
```

The above example of a query differentiates these two ways of belief querying. The first query uses a sum of world-sets ($\{name(a), -name(b), name(b), -name(a), helloFromModule(a), helloFromModule(b)\}$). In this case $name(a) \mid name(b)$ transforms to a logical operation $\mathfrak{i} \vee \mathfrak{i} = \mathfrak{i}$ (the first query's result).

The second query however, asks $name(a) \mid name(b)$ to each member of the belief base and applies

LUB to the results. In this case $name(a) \mid name(b)$ evaluated in "a" gives $\mathbf{t} \vee \mathbf{f} = \mathbf{t}$ and analogously we obtain $\mathbf{t}$ from module "b". The final result is calculated as $\text{LUB}(true, true) = true$. This explains the difference in results above.

5.5. Constraints by examples

In inter4QL both local and belief modules can declare rigid and flexible constraints, with a syntax from Module 2, which behave in a way described in Table 4.1. If violated, the constraint will force the response $\mathbf{u}$ to each query. Internally, such constraints are just a lists of expressions which had to be $\mathbf{t}$ in order to allow the queries to be evaluated in the module. For the purpose of constraints new constructs were introduced to the interpreter - like quantifiers or implication. The improved expression syntax can be used while declaring constraints and, e.g., in the $\text{Bel}()$ operator.

Let us add some constraints to our exemplary module "a":

```
1  module a:
2      constraints:
3          flexible:
4              exists X:module_name (helloFromModule(X) = false).
5      domains:
6          literal module_name.
7      relations:
8          helloFromModule(module_name).
9          name(module_name).
10     rules:
11         helloFromModule(X) :- name(X).
12     facts:
13         name(a).
14         -name(b).
15 end.
```

Module 8: Updated module "a"

The constraint introduced to the updated module "a" makes sure that there exists a member of domain *a.module_name* which, when applied as argument of relation *helloFromModule*, evaluates to $\mathbf{f}$. When we look at members of the domain, we can see that only "a" is there - see Section 5.3.2 for an explanation:[6]

```
# a.module_name(X).
== Results ==
X: a = true
```

Therefore there is no domain member for which *helloFromModule* has the $\mathbf{f}$ value which violates the constraints. All queries to module "a" are now responded with the $\mathbf{u}$ value:

```
# a.name(X).
== Results ==
unknown
```

[6]Negative facts do not add members to domains so "b" was not initially added. It was also not deduced during the reasoning phase so it did not make its way to the *a.module_name* domain.

```
# Bel[a](name(X)).
== Results ==
unknown
```

Also queries to a belief base “c” are affected - the belief base behaves now just like it simply was a “b” world:

```
# c.helloFromModule(X).
== Results ==
X: b = true
# Bel[c](helloFromModule(X)).
== Results ==
X: b = true
# c.name(X).
== Results ==
X: a = false
X: b = true
# Bel[c](name(X)).
== Results ==
X: a = false
X: b = true
# (c).(name(a) | name(b)).
== Results ==
true
# Bel[c](name(a) | name(b)).
== Results ==
true
```

Using this method one can control the desired knowledge state in a convenient way. Now, in this subsection, it is possibly not that interesting as the module is static and it will not change but during a dynamic action planning some unwanted states are easily achievable and it is convenient to have a tool to react to them.

5.6. Belief shadowing in inter4QL

As stated in Section 4.5, the shadowing operator provides a lightweight way of temporary changing of the beliefs. By lightweight we mean that no actual change in database is done. This is also how it is implemented in inter4QL. The implementation corresponds exactly to definitions given in Section 4.5 - when querying a shadowed belief base, first the right operand of “as” operator is queried and if it responds with some tuples (having either $\mathfrak{t}$, $\mathfrak{f}$ or $\mathfrak{i}$ logic values assigned) then these tuples are used as a response of the whole query (otherwise, the left operand is queried and its responses are returned). Observe no modifications to databases - all evaluations are temporary and really quick to compute.

We can now look at following example (we are still using Modules 6 and 8):

```
# (a as b).(helloFromModule(X)).
== Results ==
X: b = true
# Bel[a as b](helloFromModule(X)).
== Results ==
X: b = true
```

Even though the flexible constraints are violated in module “a”, we are still receiving responses as shown above. It is due to shadowing of flexible constraints described in Section 4.5. Constraints of an “a as b” belief contain rigid constraints from modules “a” and “b” and flexible ones from module “b” (note no flexible constraints from module “a” and hence no constraint violation).

As both modules “a” and “b” have facts for relation *helloFromModule*, in the above example the response for module “b” is returned. If we extend module “a” with facts accessible only in that module (e.g., the fact *literalReadFrom(a).*) then we will obtain following results:

```
# Bel[a](literalReadFrom(X)).
== Results ==
unknown
# Bel[a as b](literalReadFrom(X)).
== Results ==
X: a = true
```

Note that querying the literal directly from module “a” still suffers from violated constrains and the response is “unknown”. However, when the module is shadowed with module “b” (which removes the violated constraint) a meaningful response is obtained.

Of course shadowing is not limited to two belief bases only - one can build any “as” expression which matches current needs (module “c” below is declared analogously to “b”):

```
# Bel[a as b as c](helloFromModule(X)).
== Results ==
X: c = true
# Bel[a as b as c](literalReadFrom(X)).
== Results ==
X: a = true
# Bel[a as (b as c)](literalReadFrom(X) | helloFromModule(Y)).
== Results ==
X: c, Y: c = true
```

Chapter 6

Actions on beliefs - ACTLOG

This chapter is based on [22] and uses extensive fragments of the article in an unchanged form.

Reasoning about actions and change is an essential ingredient of AI systems. Throughout the years a variety of advanced solutions has been introduced, developed, verified and used in this field (see Section 1.1 for an overview of the domain). Despite a broad and intensive research on related problems (see, e.g., [99] and references there), the issue of inconsistent information has rarely been addressed in this context. However, in informationally complex environments, due to the heterogeneity of distributed information sources of diverse quality and credibility, *inconsistent* and *incomplete information* is a common phenomenon.

This attitude lies at the heart of our approach and is shared by many researchers. In particular, the importance of addressing inconsistencies in a robust manner is emphasized in [67] (see also [68]), where *inconsistency robustness* is phrased as:

> "information system performance in the face of continually pervasive inconsistencies – a shift from the previously dominant paradigms of inconsistency denial and inconsistency elimination attempting to sweep them under the rug."

For related discussion see also, e.g., [14], in particular an overview in [15] where, among others, the authors point out that:

> "inconsistency is useful in directing reasoning, and instigating the natural processes of argumentation, information seeking, multi-agent interaction, knowledge acquisition and refinement, adaptation, and learning."

The ultimate goal of this chapter is to define a planning system that is rich enough to cope with $3i$. The key is to focus on a novel approach to actions' specification, while keeping in mind a perspective of automated planning. We are defining a formal language ACTLOG for specifying actions in informationally complex environments, enjoying the following features:

- concise *rule-based specification* of actions and their effects in the presence of $3i$;
- *flexibility* in evaluation of formulas in distributed paraconsistent belief bases;
- *tractability* of computing actions' preconditions and the resulting belief bases;
- *practical expressiveness* meaning that all actions (and only such) with preconditions and effects computable in deterministic polynomial time can be specified in ACTLOG.

ACTLOG belongs to the 4QL family of four-valued, rule-based languages. It builds on 4QL$^{\text{Bel+}}$ (see Chapter 4), which, in turn, extends the 4QL rule language (see Section 2.1). While 4QL already permits to flexibly resolve/disambiguate $3i$ at any level of reasoning, 4QL$^{\text{Bel+}}$ includes means for doxastic reasoning by specifying paraconsistent belief bases with constraints and referring to them in rules.

6.1. Atomic Actions

Let us now extend 4QL$^{\text{Bel+}}$ towards specifying actions. Our approach reflects the general idea of action definition. As a novelty, an ACTLOG action is a belief bases transformer: a state of the environment, expressed as a belief base, is transformed by an action into the resulting belief base. Next, the use of 4QL$^{\text{Bel+}}$ to represent actions' effects ensures their concise representation which is one of our important goals. Finally, due to tractability results for 4QL$^{\text{Bel}}$ (Section 3.2.3) and 4QL$^{\text{Bel+}}$ (Section 4.6), effects of actions can be computed in a tractable manner.

All back-end operations like reasoning management is handled by 4QL$^{\text{Bel+}}$.

Let us start from defining actions' specification. The syntax is presented as Action 1, where:

- `act` is the action name and $\bar{x}$ are its parameters;
- $\alpha(\bar{x})$ is an arbitrary formula of 4QL$^{\text{Bel+}}$, called the *precondition* of action `act`;
- $\beta^+(\bar{x}), \beta^-(\bar{x})$ are 4QL$^{\text{Bel+}}$ programs, representing effects of action `act` by specifying sets of literals to be added ($\beta^+(\bar{x})$) and to be removed ($\beta^-(\bar{x})$);
- it is assumed that α, β^+ and β^- contain no free variables other than those in $\bar{x}$.

By an *instance* of action `act`$(\bar{x})$ we mean `act`$(\bar{a})$, where $\bar{a}$ is a tuple consisting of constants.

```
1 action act(x̄):
2 |  preconditions:
3 |  |  α(x̄)
4 |  postconditions:
5 |  |  add:
6 |  |  |  β⁺(x̄)
7 |  |  remove:
8 |  |  |  β⁻(x̄)
9 end.
```

Action 1: Syntax of actions in ACTLOG.

Recall that one of our goals is to achieve concise specifications of actions' pre- and postconditions, like:

$$\{\underbrace{\textit{safe-path(X,Y)}}_{\text{true}}, \underbrace{\textit{in(X)}, X \neq Y}_{\text{true}}\}\ \texttt{move(X,Y)}\ \{\underbrace{\neg \textit{in}(X)}_{\text{true}}, \underbrace{\textit{in}(Y)}_{\text{true}}\}; \tag{6.1}$$

$$\{\underbrace{\textit{safe-path(X,Y)}}_{\text{inconsistent}}, \underbrace{\textit{in}(X), X \neq Y}_{\text{true}}\}\ \texttt{move(X,Y)}\ \{\underbrace{\neg \textit{in(X)}}_{\text{true}}, \underbrace{\textit{in(Y)}}_{\text{inconsistent}}\}; \tag{6.2}$$

$$\{\underbrace{\textit{safe-path(X,Y)}}_{\text{unknown}}, \underbrace{\textit{in(X)}, X \neq Y}_{\text{true}}\}\ \texttt{move(X,Y)}\ \{\underbrace{\neg \textit{in(X)}}_{\text{true}}, \underbrace{\textit{in(Y)}}_{\text{unknown}}\}. \tag{6.3}$$

Action 2 provides a concise specification of (6.1)–(6.3) in ACTLOG.

```
1  action move(ID,X,Y):
2      preconditions:
3          safe-path(X,Y) ∈ {t, i, u} ∧ position(ID,X) ∧ X ≠ Y
4      postconditions:
5          add:
6              position(ID, Y) :– safe-path(X,Y).
7              ¬ position(ID,X).
8          remove:
9              position(ID, X).
10 end.
```

Action 2: A concise specification of (6.1)–(6.3) in ACTLOG.

It is also important to notice that rules in action specification may use operators like, e.g., $\mathrm{Bel}_{\Delta}()$, referring to belief bases or shadowing expressions. This, among others, allows one to deal with distributed belief bases. Since such bases are known from the context, we sometimes omit the subscript indicating a belief base.

Definition 17 Tuples $\langle a_1, \ldots, a_k \rangle, \langle b_1, \ldots, b_l \rangle$ consisting of variables and/or constants are called *compatible* if $k = l$ and, for $i = 1, \ldots, k$, at least one of a_i, b_i is a variable or both $a_i, b_i \in$ *Const* and $a_i = b_i$.

Given a $3i$-world w, specification expressed as Action 1 and a tuple of constants $\bar{a}$ compatible with $\bar{x}$, the action $\mathtt{act}(\bar{a})$ is *executable* on w when its precondition $\alpha(\{w\}, v) = \mathbf{t}$, where v assigns constants $\bar{a}$ to variables $\bar{x}$, respectively.

An action is *executable* on a belief base Δ, if it is executable on some $w \in \Delta$. ◁

Note that in preconditions of actions (formula α of Action 1) one can use any formula of the form defined in Table 3.1 (Page 30), in particular involving the $\mathrm{Bel}()$ operator as well as the operator '$\in T$', allowing one to react to inconsistency and lack of knowledge. Therefore an action can be executed when the state is inconsistent and/or some/all literals are unknown. Running actions in such circumstances is a unique feature of ACTLOG.

When action $\mathtt{act}(\bar{a})$ is executed, it transforms its input belief base Δ into the resulting belief base Δ' as shown in Algorithm 9, where Δ' represents *effects of* action $\mathtt{act}(\bar{a})$ on Δ.

6.2. Composite Actions

Composite actions' specifications are important in applications. Apart from simplifying specifications, they can allow for more efficient plan building. Namely, their use as kinds of templates frequently occurring in a given application area can substantially reduce the branching factor when searching for plans by avoiding explorations of useless branches. For example the sequence 'locate-lift-move', consisting of three atomic actions, can be used in planning without the necessity to construct this sequence during the planning process.

For simplicity we concentrate on sequential and parallel compositions, and if-then-else operator only. First, these operations do not increase the number of $3i$-worlds within the resulting belief bases. Second, their use does not affect tractability of the approach.

Input: action $\texttt{act}(\bar{a})$, specified as Action 1, where $\bar{a}$ is a tuple of constants;
belief base Δ;
Result: belief base Δ' representing effects of executing $\texttt{act}(\bar{a})$ on belief base Δ;

set $\Delta' = \emptyset$
foreach $L \in \Delta$ **do**
  if action $\texttt{act}(\bar{a})$ is executable on L **then**
    set $M^+ = \emptyset$; **set** $M^- = \emptyset$;
    compute the well-supported model of $\beta^+(\bar{a}) \cup L$ adding to M^+
     each literal obtained as a consequence of a rule of $\beta^+(\bar{a})$;
    compute the well-supported model of $\beta^-(\bar{a}) \cup L$ adding to M^-
     each literal obtained as a consequence of a rule of $\beta^-(\bar{a})$;
    add the set $\big((L \cup M^+) \setminus M^-\big)$ **to** Δ'
  else
    add the set L **to** Δ'
  end
end

Algorithm 9: Computing effects of actions.

Composite actions are specified as shown in Action 3, where γ is an expression consisting of atomic actions (with parameters), built using ';' (sequential composition), $\Rightarrow$ (conditional 'if-then-else') and '||'(parallel composition).

action $\texttt{act}\,(\bar{x})$ **:**
  composite:
    $\gamma(\bar{x})$
end.

Action 3: Syntax of composite actions in ACTLOG.

The full syntax of composite actions is given in Table 6.1.

$\langle Composite \rangle$	::=	$\langle Atomic \rangle \mid \langle Composite \rangle \,;\, \langle Composite \rangle \mid$ $\langle Formula \rangle \Rightarrow \langle Composite \rangle \,/\, \langle Composite \rangle \mid$ $\langle Composite \rangle \,||\, \langle Composite \rangle$

Table 6.1: Syntax of composite actions.

We assume that arguments of actions in γ belong to arguments $\bar{x}$ of the action $\texttt{act}$ and we disallow recursion. To formally define this requirement, for a set of action specifications consider a *reference graph* $\langle V, E \rangle$ where V is a set of nodes labeled by action names and $(n_1, n_2) \in E$ iff n_2 occurs in n_1's **composite** section. In ACTLOG we only allow action specifications whose reference graph is acyclic.

Operators ';', '$\Rightarrow$' and '||' transform belief bases into belief bases. Given a belief base Δ, and action $\texttt{act}$, by $\texttt{act}(\Delta)$ we denote the belief base representing effects of $\texttt{act}$. While the semantics of ';' and '$\Rightarrow$' is rather straightforward, let us explain our approach to '||'. When there are no conflicts between actions $\texttt{act}_1$ and $\texttt{act}_2$, their parallel composition $\texttt{act}_1||\texttt{act}_2$ simply adds to $3i$-worlds

literals added by $\texttt{act}_1$ or by $\texttt{act}_2$ and removes literals deleted by $\texttt{act}_1$ or by $\texttt{act}_2$. However, when there are conflicts (e.g., $\texttt{act}_1$ attempts to add a literal ℓ and at the same time $\texttt{act}_2$ attempts to remove it), we solve the conflict by assuming that, in the resulting $3i$-world, ℓ is inconsistent. As an example of a conflict consider actions `pourWater` (resulting in removing the literal $fire$) and `lightFire` (resulting in adding $fire$). In that case, the execution of:

$$\texttt{pourWater} \,||\, \texttt{lightFire}$$

results in adding $fire$ and $\neg fire$ to the $3i$-world, making $fire$ inconsistent. Of course, using 4QL$^{\text{Bel+}}$ one can later disambiguate such conflicts, e.g., taking into account the relative strengths of actions (if known).

Note that in parallel composition $\texttt{act}_1||\texttt{act}_2$ both actions are executed when their preconditions are both true. If this is not the case, one or none of $\texttt{act}_1$, $\texttt{act}_2$ is executed. To make sure that both actions are actually executed, one can use conditional specification with the condition being the conjunction of preconditions of $\texttt{act}_1$ and preconditions of $\texttt{act}_2$.

The semantics of action instances is given in Table 6.2.

- For any atomic action instance $\texttt{act}$, $\texttt{act}(\Delta) = \Delta'$, where Δ' is defined by Algorithm 9;
- $(\texttt{act}_1;\texttt{act}_2)(\Delta) \stackrel{\text{def}}{=} \texttt{act}_2(\texttt{act}_1(\Delta))$;
- $(F \Rightarrow \texttt{act}_1/\texttt{act}_2)(\Delta) \stackrel{\text{def}}{=} \begin{cases} \texttt{act}_1(\Delta) & \text{when } F(\Delta) = \mathbf{t}; \\ \texttt{act}_2(\Delta) & \text{otherwise;} \end{cases}$
- $(\texttt{act}_1||\texttt{act}_2)(\Delta) \stackrel{\text{def}}{=} \big\{\big(M^+_{\texttt{act}_1}(L) \cup M^+_{\texttt{act}_2}(L)\big)\setminus\big(M^-_{\texttt{act}_1}(L) \cup M^-_{\texttt{act}_2}(L)\big) \cup \{\ell, \neg\ell \mid \ell \in (M^+_{\texttt{act}_1}(L) \cap M^-_{\texttt{act}_2}(L)) \cup (M^-_{\texttt{act}_1}(L) \cap M^+_{\texttt{act}_2}(L))\} \mid L \in \Delta\big\}$;

where, for $i \in \{1, 2\}$, $M^+_{\texttt{act}_i}$ and $M^-_{\texttt{act}_i}$ refer to sets of literals computed in Algorithm 9 applied to action $\texttt{act}_i$.

Table 6.2: Semantics of composite actions.

6.3. Tractability of the approach

For any ACTLOG specification of an action $\texttt{act}(\bar{x})$, by $\#D$ we denote the sum of sizes of all domains in the specification, $\#L$ stands for the sum of lengths of composite actions specifications and by $\#M$ we denote the number of 4QL$^{\text{Bel}}$ modules occurring in the specification. For belief base Δ, by $\#\Delta$ we denote the number of all literals appearing in Δ. Note that $\#\Delta$ is polynomial in the size of $\#D$ (the size of relations is constant). In real-world applications, $\#M$ as well as $\#D$ are manageable by the hardware/database systems used, so is $\#\Delta$.

The intuition for the complexity results should be added for the ACTLOG as well. One can easily see, that validating preconditions translates to a simple query to a world and therefore takes a polynomial time. Computing the effects of the actions, as they are rules, requires computing a well-supported model in a polynomial time. An action can potentially modify each of the $\#M$ modules but at the same time each modification will impact (add of remove) some polynomially-bounded amount of literals which makes this operation PTIME as well. Finally, when looking at Table 6.2 one can notice,

that composite actions do not increase the number of worlds making sure that computing the effects of actions stays within polynomial computability time.

The following theorems can be proved similarly to analogous results for 4QL (see Section 2.1.4) and 4QL$^{\text{Bel}}$ (see Sections 3.2.3 and 4.6).

Theorem 7 Let Δ be a belief base. For every ACTLOG specification of action $\texttt{act}(\bar{x})$ and a tuple of constants $\bar{a}$, compatible with $\bar{x}$, the preconditions and effects of $\texttt{act}(\bar{a})$ can be computed in deterministic polynomial time in $\max\{\#D, \#L, \#P, \#\Delta\}$. ◁

Theorem 8 ACTLOG captures deterministic polynomial time over linearly ordered domains. That is, every atomic action with polynomially computable preconditions and effects can be expressed in ACTLOG. ◁

Chapter 7

Implementation of actions - inter4QL planner

The implementation of actions caused many new challenges in the development of inter4QL. So far "only" queryable modules and belief bases were supported. That is, with minor customizations, the same interfaces could have been used for all types of modules. When actions were introduced, some changes were required as two completely new entities appeared - actions and planning problem specifications.

While actions have been described in details (together with their syntax and semantics) in Chapter 6 (based on [22]), the implementation was mostly focusing on specifying a grammar of actions and providing appropriate data structures to store them. Also, the specification of planning problems were not that hard to provide as we could base our structure on some STRIPS-like problem definition. The real challenge appeared when static knowledge base structures from original inter4QL were to be transformed to dynamically updated states especially in cases where applications of actions were not linear (e.g., parallel ones). Using the whole belief bases in such a way could lead to long database handling times during planning. Therefore, a less computationally demanding solution was required.

Moreover, [22] introduced the theoretical part of the planning (with examples) but in fact did not mention anything about an actual planning method. To provide a software working in a reasonable time we have selected the forward state-space search method (with some tricks for composite actions) implemented as a DFS algorithm over world states. Such a choice also allowed to simplify the implementation due to a less complex planning state handling which was important to us as well.

7.1. Updated architecture

To meet the new requirements, additional classes had to be added to the interpreter. Although actions and planning problems are not $4QL^{Bel+}$ modules (they cannot be queried and they do not need to perform any reasoning), it turns out that storing them as another type of a module is the easiest way of building that functionality into inter4QL with, at the same time, keeping the original data structure consistency. For example, by approaching the problem that way, new elements could be passed to the *Checker* module without any additional changes to that module (apart from specifying dedicated constraints for actions and problems, of course).

A representation of actions and planning problems is presented in Figure 7.1. One can see, that apart from two new classes representing actions and problems, a set of new methods has been added to a generic module class, dedicated for managing two types of databases containing facts to be added

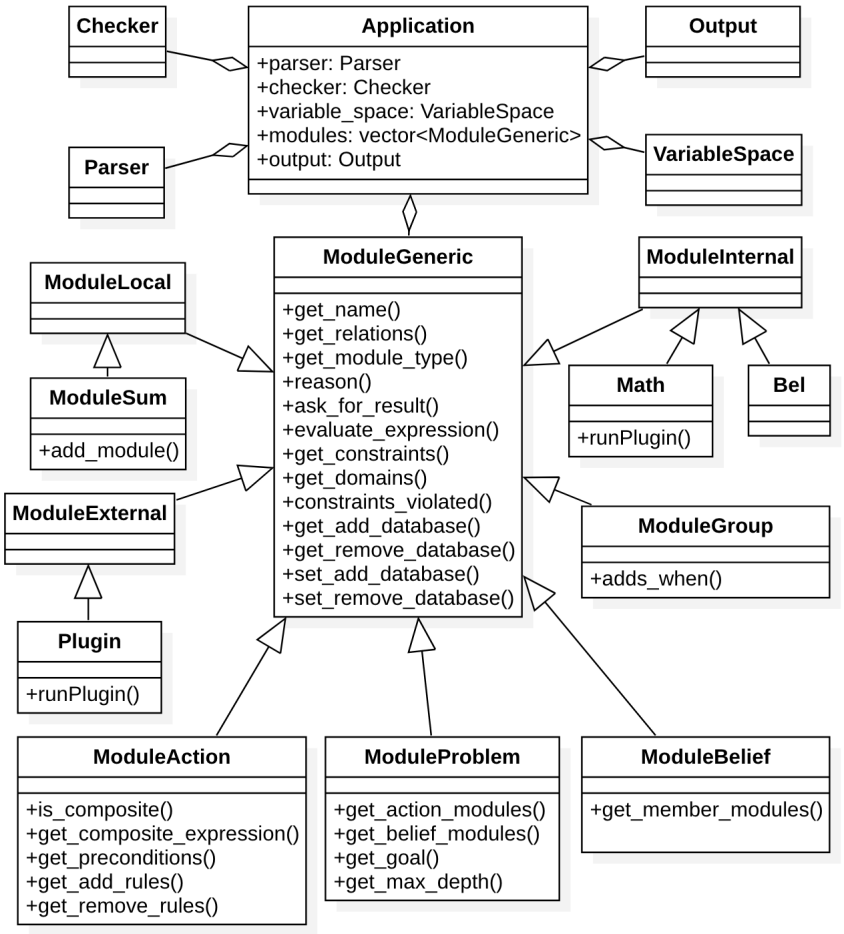

Figure 7.1: A simplified overview of the inter4QL architecture.

and removed. These databases have been introduced as a lightweight representations of planning state. Rather than handling the whole belief base, the planning engine has to deal only with facts that actually change (for details see Section 7.2).

7.1.1. Planning problem - new entity

The syntax of actions has been already presented in Actions 1 and 3, as part of Chapter 6. Each action specification in 4QL code corresponds to exactly one *ModuleAction* and contains all the information required for action's application during planning. The definition of a planning problem however was not defined in papers [19, 22] and therefore is proposed here in Problem 1.

Looking at the planning problem's syntax, setting up the planning engine calls for providing the following data:

- a list of beliefs to be used as a base for the planning process;
- a list of actions to be tried on each world in belief list;
- a $4QL^{Bel+}$ expression describing a goal of the planning;
- heuristics to be used during planning (if not specified, the planner will try to apply actions in a blind way) - see Section 7.3.

Apart from the above attributes, the maximum depth of searching has been introduced as the planning process may create infinite paths and eventually cause planner crashes. Clearly, users should have a control over this value as they know their planning problems. This attribute is mandatory but may, of course, be set to a high value. The user has to remember however that the planning process often requires an exponential time so the higher the value the longer planning process will take. Therefore some kind of balance is needed.

```
1  problem name:
2  |  beliefs:
3  |  |  ...
4  |  actions:
5  |  |  ...
6  |  goal:
7  |  |  ...
8  |  max_depth:
9  |  |  ...
10 |  heuristics:
11 |  |  ...
12 end.
```

Problem 1: Syntax of planning problem in ACTLOG.

7.2. Planning algorithm

7.2.1. Extended data storage

As stated before, the planning algorithm is based on a graph depth-first search (DFS) working on databases of literals. What is important, the actions do not modify the actual *wsm* databases of each module - this could be too expensive in terms of computation time. Instead, each *ModuleGeneric* in the planner contains its own set of two additional databases: `to_add_database` and `to_remove_database`. These databases, just like a *wsm*, are sets of ground literals organized in chunks like presented in Subsection 5.3.1. When any *ModuleGeneric* is queried for facts, the final output of the query is constructed (as a sum of sets) from the following parts:

1. the result of the query asked in module's *wsm*-database after removing all the literals that are included in `to_remove_database`;
2. the result of the query asked to `to_add_database` database after removing all the literals that are included in `to_remove_database`.

This way if an action does not modify the database, the queries will be directed to the *wsm* database of the module. At the same time, if the action adds a literal to a module's database, the added element will appear in the query result as a part of the query described in Point 2. Finally, if the action removed a literal, it will be removed from queries 1-2 and therefore will not appear in the final result as expected. One can notice that this would cause literals added and removed by the same action not to be included in the final database. This is expected as stated in Line 7 of Algorithm 9 where "positive" effects of actions are handled first and then "negative" ones are processed, effectively removing all previously processed literals from the result, even these just added.

With this minor optimization, some extensively used planning operation on databases, like making a copy or restoring them to previous states, are not that time consuming.

7.2.2. Design overview and justifications

The implemented algorithm consists of the following parts:

- the frame of the algorithm being a standard, recursive implementation of the DFS algorithm;
- the action's application method for atomic actions;
- the composite action processing algorithm.

Let us now explain why composite and atomic actions are divided into separate parts. This is because the atomic part was implemented first, using the method of action application described in Subsection 7.2.5. When composite actions were introduced, it turned out that while conditional and sequential actions can be processed with the same approach, the parallel actions need a more sophisticated algorithm. Moreover, composite actions can be used as a part of other composite actions which basically implied the use of a stack of actions to be applied (with some configuration tags between them).

7.2.3. DFS - the frame of planning

The planning algorithm is constructed using several modules dedicated to different types of actions. All of them are, however, put in one algorithmic frame being a recursive implementation of a DFS algorithm summarized in Algorithm 10. The algorithm is executed each time a user requests plan's generation via planner's console.

Please note, that Algorithm 10 is a part of `ModuleProblem`, so it has a direct access to the class members listed in Subsection 7.1.1 (like the planning goal and beliefs included in the planning problem).

The algorithm contains several elements that call for explanations. First of all, one can see dedicated functions for module/action/valuation selection instead of a simple iteration over lists (Lines 7-11). It is caused by the existence of planning heuristics (see more in Section 7.3) which may affect the order of data processing. Then in Line 10 action's preconditions are evaluated (producing a valuation of variables) which is described with examples in Subsection 7.2.4. Moving forward, in Line 15 one can see the `process_non_composite` function which is responsible for applying atomic action's effects to a given world. The detailed description of that part can be found in Subsection 7.2.5. Finally, in Lines 20 -26 one can see data structures and functions connected with composite action processing - these elements are described in Subsection 7.2.6.

If the root call of the function returns "false" then "No plan found" message is shown, otherwise actions read from "plan" string list will be printed on the console. What is important, the additional list had to be introduced (instead of simply printing action names during exiting recursive call) due to the fact, that such an approach would print the list backwards (from last to first). To maintain a good user experience, the actions are printed in a natural order, from first to last, using that additional list after planning is finished.

```
1  Function find_and_apply_action(int step, std::vector<std::string>& plan) : bool
2      foreach belief b in beliefs do
3          if goal evaluates to t in b then
4              return true;
5          end
6      end
7      while belief = selectModule(beliefs) do
8          while world = selectModule(b.worlds) do
9              while action = selectAction(b.actions) do
10                 evaluate action.preconditions in world, store result valuation in val
11                 while valuation = selectValuation(val) do
12                     if valuation.value = t then
13                         set success = false;
14                         if action.isComposite = f then
15                             process_non_composite(world, action, val);
16                             if !violated_constraints(belief) then
17                                 success = find_and_apply_action(step + 1, plan);
18                             end
19                         else
20                             composite_stack.push(EXPRESSION,
21                                   action.get_composite_expression());
22                             if !violated_constraints(belief) then
23                                 success = traverse_composite_expr(composite_stack,
24                                       add_database_stack, remove_database_stack,
25                                       step, world, belief, plan);
26                             end
27                         end
28                         if success then
29                             plan.push_back(action.get_name() + params)
30                             return true
31                         end
32                     end
33                 end
34             end
35         end
36     end
37     return false
38 end
```

Algorithm 10: The main frame of DFS implementation.

7.2.4. Action's preconditions

As presented in Actions 1 and 3 in Chapter 6, apart from add and remove rules, actions can contain an expression being a precondition of a given action. Preconditions are important not only because they allow for checking whether the action is executable in some world. Evaluating preconditions also provides us with valuations of variables together with assigned truth values in such a way, that whenever a variable in the preconditions are assigned given values, the whole expression will evaluate to the corresponding truth value.

Let us consider Actions 4 (an atomic one) and 5 (a composite one).

```
1 action take(WHAT, FROM_WHAT):
2 |  preconditions:
3 |  |  on(WHAT, FROM_WHAT).
4 |  postconditions:
5 |  |  add:
6 |  |  |  hold(WHAT).
7 |  |  remove:
8 |  |  |  on(WHAT, FROM_WHAT)
9 end.
```

Action 4: Simple "take" action.

```
1 action takeAndPut(WHAT,
  FROM_WHAT, ON_WHAT):
2 |  preconditions:
3 |  |  on(WHAT, FROM_WHAT),
4 |  |  forall X : object on(X,
  |  |  ON_WHAT) in {unknown,
  |  |  false}.
5 |  composite:
6 |  |  take(WHAT, FROM_WHAT);
  |  |  put(WHAT, ON_WHAT).
7 end.
```

Action 5: Composite "takeAndPut" action.

For the purposes of the example let us assume, that in our world we have one object (apple), two places (table and floor) and the apple is currently on the floor. Execution of Action's 4 preconditions in such a world will return the following tuples:

```
# world.on(WHAT, FROM_WHAT).
== Results ==
FROM_WHAT: floor, WHAT: apple = true
```

As the apple is currently not on the table, it is not included in the result due to its "unknown" truth value. If we look at the preconditions result for Action 5, the evaluation will generate the following tuples:

```
# (world).(on(WHAT, FROM_WHAT),
forall X : object (on(X, ON_WHAT) in {unknown, false})).
== Results ==
FROM_WHAT: floor, WHAT: apple, ON_WHAT: table = true
FROM_WHAT: floor, WHAT: apple, ON_WHAT: floor = false
FROM_WHAT: table, WHAT: apple, ON_WHAT: floor = false
```

One can clearly see, that here the unknown relationship between the apple and the table is explicitly used by the "forall" statement. Interestingly, such preconditions do not allow for taking and putting the some object on the same place. It is so due to the fact, that during the precondition execution there is something already on the target place - the object to be moved. This can be a useful feature to limit the amount of planning paths to take.

After the precondition evaluation, the planner iterates over the result valuations and looks at the action as if variables in the action were assigned with current values taken from the valuation. Please note, that there can be several tuples with `t` assigned - the planner will try each version of the action.

In our case, the valuated actions to be applied will look as presented in Actions 6 and 7. Please note, that the attributes of each action (put in parentheses after actions' names) are also valuated for the purpose of result presentation. What is important, all attributes of the action have to be grounded by the preconditions. Otherwise the action will throw a runtime exception during the evaluation.

```
1 action take(apple, floor):
2 |  postconditions:
3 |  |  add:
4 |  |  |  hold(apple).
5 |  |  remove:
6 |  |  |  on(floor, apple)
7 end.
```

Action 6: Valuated "take" action with effects.

```
1 action takeAndPut(apple,
    floor, table):
2 |  composite:
3 |  |  take(apple, floor); put(apple,
  |  |   table).
4 end.
```

Action 7: Valuated "takeAndPut" action.

Such a requirement introduces quite a significant restrictions to specifying composite actions. Let us imagine, that we just want to provide the planner with a generic template of an action, without actually valuating the attributes of each action included into it. Such a possibility is also introduced - the user should just not include the variable in the attribute list of the composite action - see Action 8 (when no preconditions are provided, they are assumed to be always true).

```
1 action takeAndPut():
2 |  composite:
3 |  |  take(WHAT, FROM_WHAT); put(WHAT, ON_WHAT).
4 end.
```

Action 8: Composite "takeAndPut" action without preconditions.

In this case the planner will allow the variables to be valuated by the preconditions of the actions inside. What is important, it will still apply restrictions about the valuations emerging from previous planning decisions - e.g. "WHAT" value applied by "take" action will have to be the same for the "put" action as they were defined in sequence.

Moreover, during the result presentation, the information about the assigned values will not be lost as each composite action is printed out together with its internal actions:[1]

```
# execute putAppleOnTable.
Executing planning for problem putAppleOnTable...

Plan found:
1. takeAndPut():
#>    take(FROM_WHAT: floor (literal), WHAT: apple (literal))
#>    put(WHAT: apple (literal), ON_WHAT: table (literal))
```

[1] Example used in this subsection can be found in the planner's package as the example file ex11.4ql

```
1  Function process_non_composite(ModuleGeneric m, ModuleAction a, Valuation v)
2      set valuated_rules_to_add = {};
3      foreach Rule r in a.rules_to_add do
4          set valuated_rule = valuate(r, v);
5          add valuated_rule to valuated_rules_to_add;
6      end
7      set valuated_rules_to_remove = {};
8      foreach Rule r in a.rules_to_remove do
9          set valuated_rule = valuate(r, v);
10         add valuated_rule to valuated_rules_to_remove;
11     end
12     set to_add_snapshot_final = make_snapshot(m.to_add_database);
13     set to_remove_snapshot_final = make_snapshot(m.to_remove_database);
14     set to_remove_snapshot = make_snapshot(m.to_remove_database);
15     add valuated_rules_to_add to m.rules;
16     generate new WSM for m
17     foreach Rule r in valuated_rules_to_add do
18         set result_to_add = m.query(r.head);
19         foreach valuation v_add in result_to_add do
20             if v_add.value = t then
21                 set valuated_head = valuate(r.head, v_add);
22                 add valuated_head to to_add_snapshot_final;
23                 remove valuated_head from to_remove_snapshot_final;
24             end
25         end
26     end
27     remove valuated_rules_to_add from m.rules;
28     set m.to_remove_database = to_remove_snapshot;
29     add valuated_rules_to_remove to m.rules;
30     generate new WSM for m
31     foreach Rule r in valuated_rules_to_remove do
32         set result_to_remove = m.query(r.head);
33         foreach valuation v_remove in result_to_remove do
34             if v_remove.value = t then
35                 set valuated_head = valuate(r.head, v_remove);
36                 add valuated_head to to_remove_snapshot_final;
37                 remove valuated_head from to_add_snapshot_final;
38             end
39         end
40     end
41     remove valuated_rules_to_remove from m.rules;
42     set m.to_add_database = to_add_snapshot_final;
43     set m.to_remove_database = to_remove_snapshot_final;
44     generate new WSM for m
45 end
```

Algorithm 11: Application of atomic action to a module.

7.2.5. Atomic actions - implementation details

Let us now look at the functionality of applying a single atomic action to a world - see Algorithm 11. From now on, when we introduce a function without specifying its result type, we assume the function to return no value (void). The code resembles the generic planning Algorithm 9 from Chapter 7 pretty closely. Several design decisions have to be justified, however, when looking at the pseudocode. First of all, as two databases are used (a separate for added literals and another one for facts to be removed), modification of one database has to imply a contrary modification of the other (when we add a literal to the positive database, we have to remove the same literal from the negative one) - otherwise literals removed by an action and later added by another action would never appear in query results.

```
1 action take(WHAT, FROM_WHAT):
2     preconditions:
3         ...
4     postconditions:
5         add:
6             hold(WHAT).
7         remove:
8             on(WHAT, FROM_WHAT)
9 end.
```

Action 9: Simple "take" action.

```
1 action put(WHAT, ON_WHAT):
2     preconditions:
3         ...
4     postconditions:
5         add:
6             on(WHAT, ON_WHAT)
7         remove:
8             hold(WHAT).
9 end.
```

Action 10: Simple "put" action.

There is also a more serious problem with this approach which can be illustrated using Actions 9 and 10. Let's assume that during some plan generation the action "put" was applied with arguments "WHAT: apple, ON_WHAT: table" and followed by action "take" valuated with "WHAT: apple, FROM_WHAT: table". This means that after putting the apple from the table, module's "add" database contains {on(apple, table)} and "remove" database contains {hold(apple)}. Now, during the application of the "take" action, the algorithm will try to add "hold(apple)" to module's "add" database by adding "take" action's "add" rules to module's rule set (see Line 15 of Algorithm 11) and executing reasoning in that module (Line 16). This way only desired action's effects will be added to the module's well-supported model (including "hold(apple)"). Then it will query the module for heads of "add" rules (Lines 17-18) - so in this case it will query for "hold(apple)" - but the very same literal that is stored in module's "remove" database from the previous planning step. The query will therefore return the empty result and the action will be only partially applied. This result is wrong as the literal was removed in the previous step of planning and since then has been inferred again.[2]

The solution to this issue is quite simple - the original reasoning algorithm for local modules had to be extended with removing newly inferred literals from module's "remove" database - this way facts that are added to the module's well-supported model are not removed from it by previously added "remove" literals.

Finally, one can see that snapshots of "add/remove" databases are created and restored before calculation of each part of action's effects (Lines 12-14, 28 and 42-43). If we look at the Algorithm 9 (Page 58) then we can see, that calculation of "positive" effects of an action and the "negative" ones happen in the same starting state, with the same state of module's database. Please note that generating new well-supported model may change $to\_remove\_database$ as mentioned in previous paragraph - this is why the database is restored after calculation of the positive effects. Additionally, a snapshot of initial databases is done for the sole purpose of adding the effects to it ($to\_add\_snapshot\_final$ and

[2]The issue was present in first version of the planner, inter4QL 5.0. It is fixed in the latest version - see http://4ql.org

$do\_remove\_snapshot\_final$). Adding the positive effects have to be done to a separate database as they may affect the results for the negative ones. At the very end, the databases with all the effects applied are set as a new "add/remove" databases for module m.

7.2.6. Composite actions - complex effects

Application of the composite actions require a bit more work as the syntax of composite expressions can generate trees of actions (composite actions may include other composite actions). Although processing the leafs of these trees utilizes the same atomic action application algorithm as described in Subsection 7.2.5, getting to these leafs may require a bit of state juggling which will be described in this subsection.

Let us remind that the planner needs to support following composite actions: sequential actions (simply speaking one action has to be executed directly after another), conditional actions (depending on the evaluation result of the condition, either "then" or "else" action is applied) and parallel actions (both actions are applied at the same time and the same starting state and the results are combined using rules described in Table 6.2).

As a data structure for keeping our composite planning state we propose a stack of elements which can be divided into four major types: $START\_PARALLEL$, $RESET\_ENV\_PARALLEL$, $END\_PARALLEL$ and $EXPRESSION$. The first three types are control elements for parallel action application containing some additional information required for appropriate state handling. The fourth one is an expression which can be an actions' name (either atomic or composite, with parameters) or a composite expression.

While traversing of the composite expression is nothing more than reading data from a stack, the magic takes place during the extraction of the composite expression to this stack. The traversing algorithm does not have to worry about, e.g., checking which action in a conditional one has to be applied - this is decided while extracting conditional action to the stack. Algorithm 12 presents the overview of how each type of composite action is extracted.

Sequential and conditional action types are relatively simple. Please note that all elements are pushed to the stack in a backward order (e.g. for sequential execution first the second action is pushed and then the first). It is expected as Stack is a type of LIFO queue (Last In - First Out) which means that elements that are pushed last will pop as first (so the planning order will be correct).

The more complicated logic starts during parallel action application. This can be simplified if we notice that:

- each parallel action has to start from the same state of database (so we need to store snapshots of databases somewhere);
- databases with effects of the first parallel action will be needed after the second parallel action is applied (so we need a link to last parallel control element to put the databases there);
- according to Section 6.2, an action inside a composite one is executed if its preconditions are true (otherwise one or none of the actions can be executed). To fulfill this, we need to know if an action in parallel execution is a root action of that execution (for actions extracted from such roots, e.g., when the root is a composite action, we still want to fail the planning upon failed preconditions).

```
1  Function extract_composite_to_stack(Expression expr, Stack s, ModuleGeneric
     w)
2    if expr.type = PARALLEL then
       /* Each control element receives it's own copy of database as
          each copy may get modified e.g. during an atomic action
          processing. Moreover, each expression here is marked as a
          root of a parallel action execution. Note: "push"
          expressions below are simplified for readibility.        */
3      set to_add_snapshot = make_snapshot(w.to_add_database);
4      set to_remove_snapshot = make_snapshot(w.to_remove_database);
5      s.push(END_PARALLEL, to_add_snapshot, to_remove_snapshot);
6      s.push(EXPRESSION, expr.right_expr);
7      set to_add_snapshot = make_snapshot(w.to_add_database);
8      set to_remove_snapshot = make_snapshot(w.to_remove_database);
       /* Here some more things are set - e.g. a pointer to the
          corresponding END_PARALLEL element.                      */
9      s.push(RESET_ENV_PARALLEL, to_add_snapshot, to_remove_snapshot);

10     s.push(EXPRESSION, expr.left_expr);
11     set to_add_snapshot = make_snapshot(w.to_add_database);
12     set to_remove_snapshot = make_snapshot(w.to_remove_database);
13     s.push(START_PARALLEL, to_add_snapshot, to_remove_snapshot);
14   end
15   if expr.type = SEQUENTIAL then
16     s.push(EXPRESSION, expr.right_expr);
17     s.push(EXPRESSION, expr.left_expr);
18   end
19   if expr.type = CONDITIONAL then
20     evaluate expr.condition in w, store result valuation in val
21     if val contains variable assignments then
22       throw runtime exception
23     end
24     if val.value = t then
25       s.push(EXPRESSION, expr.left_expr);
26     else
27       s.push(EXPRESSION, expr.right_expr);
28     end
29   end
30   if expr.type = TERM then
31     s.push(EXPRESSION, expr.get_term());
32   end
33 end
```

Algorithm 12: Procedure for extracting composite expression to the stack.

Now, when we know how a composite expressions can be extracted to the stack, we can look at Algorithm 13 being the main part of a composite action execution.[3]

```
1  Function traverse_composite_expr(Stack composite_stack, Stack
     add_database_stack, Stack remove_database_stack, int step, ModuleGeneric world,
     ModuleGeneric belief, std::vector<std::string>& plan) : bool
2    if composite_stack.empty() then
3      return false
4    end
5    set stack_elem = composite_stack.top();
6    set is_control_elem = false;
7    composite_stack.pop();
8    if stack_elem.type == START_PARALLEL then
9      set is_control_elem = true;
       /* Each layer of parallel execution has own layer of
          add/remove databases.  Let's prepare them and reset "world"
          */
10     add_database_stack.push(new database);
11     remove_database_stack.push(new database);
12     world.to_add_database = stack_elem.stored_add_database;
13     world.to_remove_database = stack_elem.stored_remove_database;
14   end
15   if stack_elem.type == RESET_ENV_PARALLEL then
16     set is_control_elem = true;
       /* First parallel action is done, let's remove its effects
          from database stacks...                                    */
17     to_add_database = add_database_stack.top();
18     add_database_stack.pop();
19     to_remove_database = remove_database_stack.top();
20     remove_database_stack.pop();
       /* ...  store them in END_PARALLEL element -
          RESET_ENV_PARALLEL element has a pointer to END_PARALLEL
          ...                                                        */
21     stack_elem.parallel_end.stored_do_add_partial = to_add_database;
22     stack_elem.parallel_end.stored_do_remove_partial = to_remove_database;
       /* ...  and prepare clean databases for second parallel action
          (and reset "world")                                        */
23     add_database_stack.push(new database);
24     remove_database_stack.push(new database);
25     world.to_add_database = stack_elem.stored_add_database;
26     world.to_remove_database = stack_elem.stored_remove_database;
27   end
```

[3]For more details one can investigate the ModuleProblem.cc source file in inter4QL's source code. Function names in provided pseudocodes are the same as in the file so they are easy to find.

```
28
29  if stack_elem.type == END_PARALLEL then
30      set is_control_elem = true;
        /* Second parallel actions is done.  Let's remove its effects
           from database stack                                        */
31      to_add_second = add_database_stack.top();
32      add_database_stack.pop();
33      to_remove_second = remove_database_stack.top();
34      remove_database_stack.pop();
        /* We have effects from the first action stored from
           RESET_ENV_PARALLEL                                         */
35      to_add_first = stack_elem.stored_do_add_partial;
36      to_remove_first = stack_elem.stored_do_remove_partial;
        /* Now we have to build final results from effect's of both
           actions.  This function puts combined results into
           "to_add_first"                                             */
37      combine_parallel_parts(to_add_first, to_remove_first, to_add_second,
38          to_remove_second);
        /* Let's reset the world to original state.                   */
39      world.to_add_database = stack_elem.stored_add_database;
40      world.to_remove_database = stack_elem.stored_remove_database;
        /* Apply results either to "world" or to databases on parallel
           stack (if this is an embedded parallel call).              */
41      if add_database_stack.empty() then
42          database_to_add = world.to_add_database;
43      else
44          database_to_add = add_database_stack.top();
45      end
        /* Only "add" database is filled.  "Remove" databases has
           alredy been used in combine_parallel_parts for prevention
           of adding certain literals to "to_add_first".              */
46      add to_add_first to database_to_add;
47  end
48  if is_control_elem then
49      if !violated_constraints(belief) then
50          set success = false;
51          if add_database_stack.empty() then
52              success = find_and_apply_action(step + 1, plan);
53          else
54              success = traverse_composite_expr(composite_stack,
55                    add_database_stack, remove_database_stack,
56                    step, world, belief, plan);
57          end
58          if success then
59              return true
60          end
61      end
62  end
```

```
63
64  if stack_elem.type == EXPRESSION then
65      set expression = stack_elem.expression();
66      if expression.type == TERM then
            /* The expression is a term so it is action's name.      */
67          set action_module = get_module(expression.expr);
68          evaluate action_module.preconditions in world, store result valuation in val
69          if val.empty() == true then
70              if !disallow_failed_preconditions then
71                  continue planning without applying action_module;
                    /* If planning returned "true" then add action's name
                       to "plan" list with information about failed
                       preconditions.                                   */
72              end
73          else
                /* Checking whether action is composite or atimic and
                   iteration over valuations (world and action modules
                   are already set) are almost the same as in Algorithm
                   10.  Differences are described below.                */
74              apply action_module to the world by iterating over val;
75          end
76      else
            /* The expression is a composite one - we need do extract
               it further.                                              */
77          extract_composite_to_stack(expression.expr);
78          if !violated_constraints(belief) then
79              set success = traverse_composite_expr(composite_stack,
80                    add_database_stack, remove_database_stack,
81                    step, world, belief, plan);
82              if success then
83                  add function name to plan
84                  return true
85              end
86          end
87      end
88  end
89  put stack_elem back to composite_stack;
90  restore add_database_stack and remove_database_stack;
91  return false;
92 end
```

Algorithm 13: Non-trivial part (parallel actions processing) of the composite expression traversing procedure.

Line 74 of the Algorithm 13 requires some explanation. As most of the code is exactly the same as in Algorithm 10, this common part has been truncated to a single line to avoid duplications. However, there are some differences between these algorithms so we will explain that now. First of all, if there is something on any of the database stacks ("add_database_stack" or "remove_database_stack") then action's effects will not be added directly to the world but rather will be put to the databases on these stacks. Because of that, the results of parallel sub-actions will be put into appropriate temporary databases and later will be combined together before putting them to the actual world (or to the next databases from the stack in the case of nested parallel actions). Apart from that, Algorithm 10 decides whether to continue planning using $find\_and\_apply\_function$ or $traverse\_composite\_expr$ basing only on the type of the action. Within Line 74, however, the $traverse\_composite\_expr$ function is used when the $composite\_stack$ is not empty. Otherwise, if no more actions are prepared to be executed in a composite mode then the execution returns to a default state and proceeds with $find\_and\_apply\_function$.

Also Line 90 should be described in more details. In that line none of the previous executions (neither control elements nor expressions) finished with a successful plan and the $stack\_elem$ was already put back to the stack (the algorithm may read it again with different valuation of variables). Therefore restoring databases on the stack is crucial as otherwise some leftovers may influence future action's effects. To restore them correctly, the algorithm has to do the reverse operations than $extract\_composite\_to\_stack$ function. So if the $stack\_elem$'s type is "START_PARALLEL" then we should pop one element from both "add_database_stack" and "remove_database_stack". In the case of the "RESET_ENV_PARALLEL" we need to pop one element from both stacks and put empty databases instead. Finally, for "END_PARALLEL" it is sufficient just to put empty databases to both stacks.

As one can see, the $traverse\_composite\_expr$ acts like a custom execution of the original DFS frame algorithm. The recursive style has been preserved here as well so that both algorithms could integrate in a simplest possible manner. Traversing of the composite expression does not increment the step number as the whole execution represents an application of a single action (hence single step number). Also, please note that the goal is not checked within the function and is verified after composite action is fully applied (in next call of $find\_and\_apply\_function$ as shown in Algorithm 10).

In theory, integrating both algorithms into one iterative function with a single stack is possible and could be a nice exercise to perform. From the practical point of view however, creating such a huge blob of code will decrease the readability significantly. Changing the recursive style to the iterative one is of course beneficial due to execution depth point of view. However, taking the experimental character of the implementation into account, it does not seem extremely important as it seems sufficient for our applications.

Not surprisingly, the very important part in Algorithm 13 plays the correct restoring of the status after failed planning iteration. When executing parallel actions, databases on stack are modified and eventually popped from the stack to be used in $combine\_parallel\_parts$ function. If the planning was later unsuccessful, then the flow will return to the iteration over valuations to try the next one - but the databases on the stack will be missing (as it was popped in previous valuation try). This is why both keeping the stack size consistent and also clearing the databases between each try is crucial for obtaining correct results.

The only part missing here is the $combine\_parallel\_parts$ function which takes effects of two parallel actions and combine them using instructions given in Table 6.2 - see Algorithm 14. Nothing unusual can be spotted there. Firstly, positive effects of the first action are altered with data provided from negative effects of the first and the second actions. Then positive effects of the second action are

migrated to the the first database but only if they were not removed by the the second action (if removed by first one, they will receive the “inconsistent” value and still be migrated). Finally, “remove” databases are merged.

When all functions from this subsections are put together, a common algorithm for composite actions emerges. Of course the description has been presented to some extent of detail, leaving some subtle details to be discovered in the actual source code.

```
1  Function combine_parallel_parts(database to_add_first, database to_remove_first
     , database to_add_second, database to_remove_second)
2      foreach touple t in to_add_first do
3          foreach valuation v in t.valuations do
               /* Literal added by first action but removed by the second
                  */
4              if v exists in to_remove_second then
5                  set v.value = inconsistent;
6              end
               /* Literal added and removed by the first action          */
7              if v exists in to_remove_first then
8                  set v.value = unknown;
9              end
10         end
11     end
12     foreach touple t in to_add_second do
13         foreach valuation v in t.valuations do
               /* Literal added by second action but removed by the first
                  */
14             if v exists in to_remove_first then
15                 add v to to_add_first with value "inconsistent"
16             end
               /* Literal added by second action and not removed by it   */
17             if v exists in to_remove_second with value other than "true" then
18                 add v to to_add_first with value "true"
19             end
20         end
21     end
22     foreach touple t in to_remove_second do
23         foreach valuation v in t.valuations do
               /* Simply rewrite literals                                */
24             add v to to_remove_first with value v.value
25         end
26     end
27 end
```

Algorithm 14: Function combining effects of parallel actions.

7.3. Planning heuristics in inter4QL

The planning algorithm presented in the previous section is based on searching the whole space of states so that eventually it will find a solution. Of course, this may take a significant amount of time. This problem was also widely recognized in the literature. Researchers were trying to find some ways to guide the algorithms during their computations. The approach could, e.g., be based on defining some conditions or functions which, if optimized during planning by selecting the best actions, will potentially direct the algorithm to the solution faster then during a blind search (and possibly produce more optimized plan).

The planner implemented as a part of this PhD thesis is an experimental one, but during the implementation and testing it was clear that blind search can be still improved (as even simple planning could take non-proportionally long time). Therefore some improvements have been done in the planning algorithm to improve planning times and produced plans.

As a result five heuristics were implemented:

- "disallow_failed_preconditions" - disallows continuation of planning with failed preconditions during parallel action execution (effectively limiting planning time);
- "composite_first" - do not iterate over actions in order as they are defined, place composite actions first on the list;
- "goal_probing" - based on rating the goal's "truth level" after action's application and sorting the actions on basis of the rating (turning the blind planning into a goal-oriented one); it does not look at global situation though, i.e., sorting is done within each step only;
- "increase_rating_only" and "strict_increase_rating_only" - improvements for the "goal_probing" heuristic. Given valuation during the planning phase is allowed to be applied only if its goal rating is greater or equal (for "increase_rating_only") or greater (for "strict_increase_rating_only") than rating used in the previous planning step. Please note, that a usage of any of these heuristics automatically enables "goal_probing".

In addition one can choose a "random" option. In this case the planning algorithm does not simply iterate over beliefs/worlds/actions/valuation but rather selects them randomly (can be combined with "composite_first"). It is a variation of a blind search algorithm. It is mainly introduced for a comparison purposes.

7.3.1. Data-reordering heuristics

"Random" and "composite_first" methods are the ones that are focusing on reordering the data during planning. In theory, without them each planning should take approximately the same amount of time and internally visit the data in the same order. However, due to the C++ internals, this is not true in practice. As highlighted before in Subsection 5.3.1, the databases internally are C++ maps (unordered multimaps, to be exact). These maps store pointers to planner's entities (and even though the map is said to be an unordered one, it seems to be internally sorting the data based on the pointer addresses) which means that during each run of the planner different addresses can be assigned to valuations and therefore the order of valuations is different. This results in a significant time differences between different runs of the planner on the same planning problem (one run can take 1.5 seconds and the other 6 seconds because of the mentioned behavior).

Therefore, to normalize this a bit, data-reordering heuristics were introduced. Their job is to eliminate the internal data ordering fluctuations and potentially provide a better average planning time.

As composite actions can be seen as action templates, sometimes it is better to try them first, rather than spending time on atomic actions. This is where "composite_first" heuristics proves to be useful. It can also be combined with the "random" one resulting in composite actions to be randomized separately and used first, before randomized atomic actions.

7.3.2. Goal-oriented planning

By default, actions are ordered either as they were defined in a 4QL file or they are mixed with one or many data-reordering heuristics. The literature, however, mentions heuristics dedicated to directing the planning towards the goal and this is what "goal_probing" heuristic is implemented for. If used, data-reordering heuristics become meaningless as each valuation gets its own rating and is sorted separately based on the actual rating value.

This heuristics is based on a goal rating which measures to what extent the goal is satisfied. The rating is calculated as presented in Table 7.1.

Let v : *Var* $\longrightarrow$ *Const* be an assignment, w be a $3i$-world and A, B be formulas. Then:

- $\big(rating(t)\big)(w,v) \stackrel{\text{def}}{=} \begin{cases} 2 \text{ when } & t = \mathbf{t} \\ 0 & \text{otherwise;} \end{cases}$ for $t \in \{\mathbf{f}, \mathbf{u}, \mathbf{i}, \mathbf{t}\}$;
- $\big(rating(A)\big)(w,v) \stackrel{\text{def}}{=} \begin{cases} 2 \text{ when } & A(w,v) = \mathbf{t} \\ 0 & \text{otherwise;} \end{cases}$
- $\big(rating(\neg A)\big)(w,v) \stackrel{\text{def}}{=} \begin{cases} 2 \text{ when } & A(w,v) = \mathbf{f} \\ 0 & \text{otherwise;} \end{cases}$
- $\big(rating(A \wedge B)\big)(w,v) \stackrel{\text{def}}{=} \big(rating(A)\big)(w,v) + \big(rating(B)\big)(w,v)$
- $\big(rating(A \vee B)\big)(w,v) \stackrel{\text{def}}{=} \max\{\big(rating(A)\big)(w,v), \big(rating(B)\big)(w,v)\}$

Table 7.1: Formula rating calculation rules.

Simply speaking, a rating for a formula is assigned the value 2 if the formula evaluates to $\mathbf{t}$ in a given world w. Ratings are summed within conjunction of formulas and a maximum of ratings is selected for a disjunction of these. Please note, that the definition does not distinguish special types of formulas like quantified ones or implications as the rating is based only on the value the formula evaluates to.

The introduction of ratings requires a slight modification of the planning algorithm. Most of the modifications take place in $find\_and\_apply\_function$ function. Instead of simply selecting an action for application and proceeding with planning, the algorithm checks actions preconditions, applies the action and sets a flag that goal probing is ongoing. When this flag is set, the first encountered goal check (meaning the closest execution of $find\_and\_apply\_function$ function) will always return "false" (no matter whether the goal is satisfied or not) and apart from that, will return goal's rating (the rating will be changed to 9999 if the goal is satisfied). This means, that the algorithm will probe one step ahead (trying one atomic action or one composite action, no matter how complex it is). When all actions are tested, they are sorted with respect to goal rating and executed (normally this time) starting with the most promising one.

The modifications are also required in $traverse\_composite\_expr$ function as otherwise composite actions will remain deaf to the rating hints and would have to be reapplied blindly. Of course such

a blind application would be assigned with an appropriate goal rating, but internally each atomic action will not know how to order its valuation to get to that highest rating. This is why we introduce $composite\_ratings$ map storing the information about:

- the current step of planning;
- the current step inside of $traverse\_composite\_expr$ call;
- belief and world name;
- current action's name;
- current action's parameters;
- rating assigned to the above data.

Such information is enough to uniquely identify the valuation applied during composite action handling. What is important, each entry stores the maximum rating that can be obtained for the current action. This means that the deeper recursive executions may improve the ratings for shallower ones if better goal rating was found. Because of that, orderings made on these shallower calls will select the best valuations first and then move to the worse ones.

In practice the entries in the map can look like this:[4]

```
Adding key "0.2.take.env.environment.table (literal).c (literal).
    0.3.put.env.environment.c (literal).b (literal)."
    to database with rating: 4
Adding key "0.2.take.env.environment.table (literal).c (literal)."
    to database with rating: 4
```

The above keys added to the map inform that, if in planning step 0 and composite step 2 we apply action "take" in belief "env" and world "environment" with parameters "table" and "c" then the best rating so far would be 4. This rating can be achieved if in the next composite step we execute action "put" with attributes "c" and "b". Observe, that the best rating from a deeper execution is assigned to the shallower one (if it better than the one already stored there) in order to allow the valuation selector function to select the best valuation starting from the very beginning of composite processing.

The computation of goal's rating is achieved by extending standard expression evaluation algorithm. Except from truth value, it is now also capable of returning expression's rating using rules from Table 7.1.

Global valuation ordering

One can notice, that although the valuations are sorted within one planning step, there is still a possibility that one locally best valuation will lure the planning engine to a deep and useless path with actually lower goal ratings ahead. To avoid that, two additional heuristics were introduced: "increase_rating_only" and "strict_increase_rating_only". Using these heuristics, a provider of a planning problem can indicate that planning should go only in a direction of increasing ratings or strictly increasing ratings. Thanks to such information, the planner is able to cut worse paths at the very beginning of the planning.

[4] To enable such logs in the planner please execute *debug "probing" enable.* and start the planning problem.

Please note two problems with composite actions here. Firstly, the $composite\_ratings$ map is constructed in a way that the same "best" rating is kept on the whole composite path. This means, that such actions would never be applicable if "strict_increase_rating_only" is enabled. To fix this, in $traverse\_composite\_expr$ function only, we always allow for equal ratings (which seems logical as whole composite action is treated as one). Secondly, if the rating threshold would be updated on each step of composite action handling, we would set it to the best achievable rating during the very first step and then ignore everything that is lower than that - which is obviously wrong. Fortunately, the solution here is easy - only ratings calculated after the whole successfully applied actions should affect the threshold. Therefore it is always updated before execution of atomic actions handling and before the whole composite action checking.

Chapter 8

Actions in action

All of the chapters so far where guiding the reader through the internals of the beliefs, belief bases, actions and finally implementation to the point where all these ideas can be tested in some practical applications. The chapter is divided into three subsections containing discussions on three different scenarios. The first two scenarios are focusing on the comparison between ACTLOG and already existing, well known planning engine - PDDL. These two sections also provide a comparison of the planning performance using heuristics built into the planner. The third scenario is a bit different as it presents an actual realistic situation of bomb defusing where robots are usually used. Apart from presenting the scenario only, the third section will propose and discuss an ACTLOG's action set dealing with the situation.

It is important to mention here that all commands and programs presented in the chapter are included into the release of inter4QL interpreter available at *http://4ql.org/inter4qlPhD.html*.[1] The reader can switch to the interpreter to experiment on his own with the modules and verify the results.

Times presented in the chapter were gathered using 2018's Apple MacBook Air computer (1,6 GHz Intel Core i5, 8 GB DDR3 RAM) and, of course, may differ from the times obtained on other computers.

8.1. Scenario 1 - raising the table

The scenario is available in the file "example10.4ql" in the inter4QL's package.

The scenario is quite simple - we have a table standing in the middle of a room. On the table there is a glass with a water. The goal is to raise the table without flipping over the glass and spilling the water. If a robot raises only one side of the table, the glass will obviously fall or flip over. This is why we assume here, that the robot has two arms and can raise both sides of the table simultaneously - this is the key to the success.

Representation in ACTLOG

Let's now look at the actual specification of the scenario in ACTLOG starting with the module "environment" storing the current state of the world during planning:

```
module environment:
```

[1] Each section will point to a specific example file in interpreter's package. Please follow instructions on the web page or interpreter's "help" command output to load them and experiment.

```
    domains:
        literal element.
        literal side.
        literal stat.
    relations:
        raised(side).
        status(stat).
    facts:
        side(left).
        side(right).
        stat(ok).
        stat(fail).
        -status(fail).
end.
```

One can see that we have two sides of the table - left and right. Both of the sides can be raised and the status of each side will be described by the "raised" relation. If both sides are raised with respect to scenario's constraints, the status of the planning will be changed to *ok* (*fail* otherwise). One can note, that adding additional artificial "status" is not needed as planning goal can directly check the required conditions also containing inconsistencies and partial knowledge. However, this scenario is also meant to present unique abilities of the planner.

As stated earlier, planning operates on belief bases so we define the following belief base:

```
beliefs env:
    constraints:
    rigid:
        (raised(left) in {true, incons},
            raised(right) in {true, incons})
        |
        (raised(right) in {unknown},
            raised(left) in {unknown}).
    worlds:
        environment.
end.
```

The above belief base contains one world "environment" but also includes a constraint for this problem. The only rigid constraint here is responsible for keeping the state safe by defining safe as "both sides of the table are raised or none sides of the table are raised". All other states of the environment violate the constraints and discontinue planning.

Finally, we need actions:

```
action raiseLeft():
    // cannot raise twice...
    preconditions:
        raised(left) in {unknown},
    add:
        raised(left).
    // artificial remove - just to test inconsistency
    remove:
        raised(right).
```

```
end.

action raiseRight():
    // cannot raise twice...
    preconditions:
        raised(right) in {unknown}.
    add:
        raised(right).
    // artificial remove as well...
    remove:
        raised(left).
end.
```

These actions are self explanatory. The only unusual specifications here are a bit artificial "remove" sections. Removing them will, of course, not affect the correctness of the specification but it introduces inconsistency which shows how easily the planner can deal with inconsistent planning goals (let us remind, that if one action in parallel execution adds a literal to a database and the other removes it then the final output of the action adds the literal with a truth value **i**).

The above actions will not be able to fulfill the goal's requirements though. We need an action that will raise both sides of the table at the same time. We can define such an atomic action (call it, e.g., *raiseBothSides*) and combine the above "add" and "remove" sections into one. This, however, will not correspond with the natural feeling of the atomic action (such *raiseBothSides* action does not feel atomic any more, it in fact contains two separate actions artificially squeezed into one).

This is, among others, why composite actions were introduced. The following action looks and feels much better:

```
action raiseBoth():
    composite:
        raiseLeft() || raiseRight()
end.
```

This composite action represents exactly our intentions - we would like to "execute two separate actions at the same time". Each action will check its precondition independently and fail if needed (also independently from the other allowing for only partial action execution).

Finally we define status reporting (introduced mainly for testing if-then-else composite actions):

```
action addOK():
    add:
        status(ok).
end.

action addFAIL():
    add:
        status(fail).
end.

action verifyRaise():
    composite:
        // if => then / else
        raised(left) = incons,
```

```
            raised(right) = incons
        => addOK() / addFAIL()
end.
```

The planning problem definition is following:

```
problem raiseTable:
    beliefs:
        env.

    actions:
        raiseLeft.
        raiseRight.
        raiseBoth.
        verifyRaise.

    goal:
        status(ok), -status(fail)

    max_depth:
        10

    heuristics:
        none.
end.
```

Please note that *addOK* and *addFail* actions are not included into the planning problem. They are locally used by *verifyRaise* composite action but do not affect the planning times for the whole problem (the planner does not have to iterate through them in each step).

Representation in PDDL

For comparison let us present the same problem (without artificial elements added) implemented in PDDL. The PDDL planning problems consist of two separate files: planning domain definition and problem definition (in ACTLOG we can have both elements in one 4QL file). To verify the PDDL specification we have used an online PDDL planner at *http://solver.planning.domains*.

The "domain" part of the problem definition can be inspected below:

```
(define (domain TABLE_DOMAIN)
    (:requirements :negative-preconditions :strips)
    (:predicates
        (raisedLeft)
        (raisedRight)
        (none)
    )

    (:action raiseLeft
        :precondition
            (and
                (not (raisedLeft))
```

```
                (not (raisedRight))
            )
        :effect (raisedLeft)
    )

    (:action raiseRight
    :precondition
        (and
            (not (raisedLeft))
            (not (raisedRight))
        )
    :effect (raisedRight)
    )
)
```

Although the domain defined above is rather standard in PDDL, let us discuss some parts of it which may not be obvious. First of all, please note the "requirements" section. PDDL is a general planning language so its features have been divided into plug-ins. Due to this decision, the user can adjust the tools as much as possible to fit the current application. Therefore, for example, to be able to use the "NOT" operator in preconditions, we have to include a special ":negative-preconditions" requirement.[2] Also, planning features have been extracted with two most common ones being ":strips" (STRIPS-based planning algorithm) and ":adl" (extensions to STRIPS containing quantified expressions and conditional effects of actions). In this particular application the pure STRIPS is sufficient.

Actions defined above correspond almost directly to the ones declared in ACTLOG in the previous subsection. The difference is that the logic underneath PDDL is two-valued which implied two-valued action preconditions. Moreover, "raise" actions defined in ACTLOG do not explicitly check the status of the other side of the table as the constraints make sure that either both sides of the table are raised or not. Here using the same mechanism was, unfortunately, not possible. Theoretically, starting with version 3.0, PDDL supports the ":constraints" command defined in planning problems which would be ideal here but due to some unknown reasons we were not able to make this running (producing "segmentation fault" in the planning engine). Therefore we changed the constraint approach to an explicit check of both sides of the table directly in action's preconditions (for PDDL only). However, the corresponding 4QL file can be easily modified to match this approach.

Now let us look at the planning problem itself:

```
(define (problem RAISE_TABLE)
    (:domain TABLE_DOMAIN)
    (:init
        (none)
    )
    (:goal
        (AND
            (raisedLeft)
            (raisedRight)
        )
    )
)
```

[2] Interestingly, the online planning engine did not report any errors when this was not included. However, to be aligned with the PDDL documentation, I am leaving it in the code.

Here, the one main difference between PDDL and ACTLOG can be spotted. Namely, the initial state of the planning is built into the PDDL's problem definition. In ACTLOG the initial well-supported models of the modules included in the belief base define the initial state. Apart from that, it seems that the PDDL parser does not allow for missing or empty "init" command so we created an additional "none" predicate to mitigate this.

Let us now look at the output of the planning engine:

```
BFS search completed
Planner found 0 plan(s) in 0.328secs.
```

As one can see, PDDL does not seem to allow parallel action execution. Unfortunately, it also does not allow for defining complex actions which could provide a hint to the planning engine that some of the actions are worth to be tried together. This case can be of course fixed by adding the following action:

```
(:action raiseBoth
    :precondition
        (and
            (not (raisedLeft))
            (not (raisedRight))
        )
    :effect (and (raisedLeft) (raisedRight))
)
```

Now the output look as follows:

```
IW search completed
0.00100: (raiseboth)
Planner found 1 plan(s) in 0.286secs.
```

What is also interesting here, the PDDL planning engine changes planning methods on the basis of the problems passed to it. During the first try it was using the BFS algorithm (Breadth-First Search) which stores more statuses in the memory at the same time but finds an optimal plan for a given problem. In the second try the engine utilized the IW algorithm (Iterative Width) which consists of a sequence of BFS calls parameterized with a constant iterated with each call. The constant is used for limiting the "novelty" parameter of a state during search (with "novelty" of a state being defined as a size of the smallest tuple of facts that hold in that state for the very first time since the start of the planning process). In other words, each BFS call within IW algorithm is given with constant k and prunes all the states which "novelty" parameter greater then k. For more details please see, e.g., [11].

Heuristics in the scenario

Let us now return to the ACTLOG's specification for the scenario. If we execute it in the version described in this section, we will end up with planner's output analogical to following one:

```
# execute raiseTable.
Executing planning for problem raiseTable...

Plan found:
1. raiseBoth():
#>    raiseLeft()
#>    raiseRight()
```

```
2. raiseBoth():
#>   raiseLeft() - failed preconditions
#>   raiseRight() - failed preconditions
3. raiseBoth():
#>   raiseLeft() - failed preconditions
#>   raiseRight() - failed preconditions
4. raiseBoth():
#>   raiseLeft() - failed preconditions
#>   raiseRight() - failed preconditions
5. raiseBoth():
#>   raiseLeft() - failed preconditions
#>   raiseRight() - failed preconditions
6. raiseBoth():
#>   raiseLeft() - failed preconditions
#>   raiseRight() - failed preconditions
7. raiseBoth():
#>   raiseLeft() - failed preconditions
#>   raiseRight() - failed preconditions
8. raiseBoth():
#>   raiseLeft() - failed preconditions
#>   raiseRight() - failed preconditions
9. raiseBoth():
#>   raiseLeft() - failed preconditions
#>   raiseRight() - failed preconditions
10. verifyRaise():
#>   addOK()

Planning took 66 milliseconds.
```

Here DFS (Depth-First Search) is always trying to go as deep as possible and perform the search there first. In general, this is not a good feature for a planning algorithm but on the other hand the planning state required to be used can contain only differences between module models in consecutive planning steps (opposite to the need of storing full module in case of e.g. BFS algorithm to be able to return to that state at any time). To balance the reasonable implementation simplicity and to avoid planning outputs as shown above, planning heuristics were introduced.

"disallow_failed_preconditions"

The first heuristics to be discussed is "disallow_failed_precondition". Let us remind that, as specified in Subsection 6.2, not always both actions in a parallel execution are applied due to failed preconditions. It is possible that one of none of these actions are executed which in fact can make such composite action the empty one - this is the case in current scenario. Of course, the appropriate action's preconditions should be used to avoid this but one more possibility was introduced being the "disallow_failed_precondition" planning heuristic. Using this option, planning output looks as follows:

```
Plan found:
1. raiseBoth():
#>   raiseLeft()
#>   raiseRight()
```

```
2. verifyRaise():
#>    addOK()

Planning took 27 milliseconds.
```

In this particular scenario the heuristics fixed the issue and the obtained plan is optimal. In general case however, using this heuristics may completely block the possibility of finding the plan or may not provide optimal plan if other action's preconditions are not well defined.

This is why the heuristic is provided for the user (just like the ":requirements" section in PDDL domains) so that the user can decide whether such functionality can improve plans and planning times or break the planning problem.

"composite_first"

In this scenario the "composite_first" heuristics does not have any influence on the efficiency of the produced plan. Please note, that the problem with the original plan is caused by a feature of parallel actions which allows them to be partially applied. This happens only when "verifyRaise" action is applied before the "raiseBoth" one. This means that the order of these composite actions is important. Due to its design, "composite_first" heuristic is not able to influence the order of actions within the composite collection so the generated plan suffers from the same inefficiency as the original plan.

At the same time please note, that the optimal plan consists of composite actions only. Therefore using this heuristic improves planning time which is decreasing from around 70 milliseconds to around 50.

"random"

Randomization of applied actions and valuations cannot be easily judged in this scenario. Of course it can lead to providing an optimal plan but couple of seconds later it may cause planning to last 3.5 seconds and return a non-optimal plan:

```
# execute raiseTable.
Executing planning for problem raiseTable...

Plan found:
1. raiseBoth():
#>    raiseLeft()
#>    raiseRight()
2. verifyRaise():
#>    addOK()

Planning took 33 milliseconds.

# execute raiseTable.
Executing planning for problem raiseTable...

Plan found:
1. raiseBoth():
#>    raiseLeft()
#>    raiseRight()
2. raiseBoth():
```

```
#>    raiseLeft() - failed preconditions
#>    raiseRight() - failed preconditions
3. raiseBoth():
#>    raiseLeft() - failed preconditions
#>    raiseRight() - failed preconditions
4. raiseBoth():
#>    raiseLeft() - failed preconditions
#>    raiseRight() - failed preconditions
5. verifyRaise():
#>    addOK()

Planning took 3559 milliseconds.
```

This method cannot be reliably used as a way of improving returned plans but can be really good for testing the quality of provided planning problems. When randomization is enabled, all orders assigned to actions and valuations during problem writing (by a user) and parsing (be the planner) is lost so planning may enter paths that were not explicitly assumed by the creator of the planning problem. For example, over 3.5 second planning time was caused by repetitions of "verifyRaise" action which was causing useless planning paths with "status(fail)" added to database at the very beginning of the path. Therefore, even if further planning found a solution, the goal could not be satisfied. Such issues can be then fixed using appropriate preconditions or planning constraints.[3]

"goal_probing"

Goal probing is the most advanced heuristic in the planner. As described in the previous chapter, it applies an action for testing purposes only, calculates goal's rating and orders actions for an execution based on these ratings. In the case of this scenario it works well.

Please note, that the problem with the plan generated for this scenario follows from the fact, that first we raise both sides of the table and then we can have potentially endless sequence of empty "raiseBoth" actions applied until finally the goal is verified by "verifyRaise" action. The problem results from the fact that empty "raiseBoth" action has the same priority as "verifyRaise" action. We can change that using "goal_probing" heuristic.

Initially, the planner has a choice from "raiseBoth" and "verifyRaise" actions (all other actions will be assigned with rating -50 due to failed constraints). The rating of "raiseBoth" equals 2 (due to "-status(fail)" literal) while rating of "verifyRaise" is calculated to be 0 ("-status(fail)" no longer evaluates to true) - the planner will therefore choose "raiseBoth" for evaluation. When the table is raised, the goal rating of each empty "raiseBoth" does not change (it is still 2) while rating of "verifyRaise" action jumps to 4 as both parts of the goal evaluate to true (in fact, the final rating, due to fulfilled goal, will be changed to 9999 as highlighted in Subsection 7.3.2). This is why the plan will always be optimal:

```
Plan found:
1. raiseBoth():
#>    raiseLeft()
#>    raiseRight()
2. verifyRaise():
#>    addOK()
```

[3]The planner provides a way of inspecting the planning process using the "debug" command. One can enable debug logs before the actual execution to see what values are checked and in which order.

```
Planning took 53 milliseconds.
```

The price for the optimal plan is the planning time being almost twice higher than without the probing. It is visible in this small example but with problems complex enough (see next two scenarios) it turns out that it is worth paying some time for good planning hints (saved time exceeds the time spent on probing significantly).

Please note that using "increase_rating_only." and "strict_increase_rating_only." does not influence the result here as we are initially start with a negative rating threshold which is then updated to 2 after "raiseBoth" and to 4 after "verifyRaise" (so the path is already strictly increasing).

8.2. Scenario 2 - BLOCKS-4-0 problem from IPC 2000 planning contest

The scenario is available in the file "example13.4ql" in the inter4QL's package.

Planning contests play a very significant role in the development process of planning engines. Developers around the world annually prepare their planners to compete in a series of planning problems. Some engines (like PDDL) were created especially with such contests in mind which makes the competition a serious and important event for planner creators.

Although competing in such a contest would be a great experience, the inter4QL planner is developed as a proof of concept rather than a fully developed tool ready to compete with mature engines developed for many years by international communities. However, the planning problems used in competitions are inspiring and are the important benchmarks for testing our solutions.

Our second test scenario is an IPC's (International Planning Competition) problem from year 2000 called BLOCKS-4-0. The idea is the following: we have 4 blocks laying on the table (called A, B, C and D) and one robotic arm which can take and put the block either on the table or on another block. The goal of the planning problem is to stack the blocks in a way that B is on A, C is on B and D is on C (with A staying on the table).

The scenario does not seem very complicated but this in fact is a good its feature. Its simplicity makes it a good sandbox for discussing different heuristics. Moreover, it shows, that despite some limitations our planner is capable of approaching problems from planning contests which is a positive prognostics for future ACTLOG's development.

Representation in ACTLOG

Just like in the case of the previous scenario, let us start with defining the planning world - both the actual module which will store the planning state but also a belief base which will be used in the planning:

```
module environment:
    domains:
        literal element.
        literal side.
        literal stat.
    relations:
        on(element, element).
        hold(element).
    facts:
        on(d, table).
```

```
        on(c, table).
        on(b, table).
        on(a, table).
end.

beliefs env:
    constraints:
        rigid:
            // table must not stand on anything else...
            forall X:element (on(table, X) in {false, unknown}).
            // table must not be held...
            hold(table) in {false, unknown}.
    worlds:
        environment.
end.
```

Again, here we have some planning constraints which allow for keeping the state of planning consistent. In this planning problem we do not want to take nor hold the table and we do not want the table to stand on anything (even though such state does not explicitly emerge from actions defined later, it is still worth defining general rules of the world).

Now we have to define some actions that will move the blocks:

```
action take(WHAT, FROM_WHAT):
    preconditions:
        // you can take sth if:
        // - you are not holding anything right now
        // - it is on something
        // - there is nothing on top of it
        forall X:element (hold(X) in {false, unknown}),
            on(WHAT, FROM_WHAT) in {true, incons},
            forall X:element (on(X, WHAT) in {false, unknown}).
    add:
        // we are now holding it!
        hold(WHAT).
    remove:
        // that thing stops being on sth
        on(WHAT, FROM_WHAT).
end.

action put(WHAT, ON_WHAT):
    preconditions:
        // you can put sth if:
        // - you are holding it
        // - there is nothing placed on the destination
        // - you are not trying to put sth on itself
        hold(WHAT) in {true, incons},
            forall X:element (on(X, ON_WHAT) in {false, unknown}),
            -math.eq(WHAT, ON_WHAT).
    add:
```

```
        // our thing is now on sth...
        on(WHAT, ON_WHAT).
    remove:
        // we are not holding it anymore...
        hold(WHAT).
end.
```

Please note complex action preconditions. They are necessary to correctly select an object to take/put as missing preconditions can cause, e.g., an object with something on top to be taken. Again, the specification here may be a bit too complex for the tackled problem (in the original problem solution those quantified formulas were simply represented by "clear" relation stating that nothing is on top of a "clear" brick) but again, this is to show several features of the planner in one place.

What is missing here is planning problem:

```
problem bricks4:
    beliefs:
        env.

    actions:
        take.
        put.

    goal:
        on(a, table), on(b, a), on(c, b), on(d, c)

    max_depth:
        6

    heuristics:
        none.
end.
```

Now, using this planning problem the planner has all it needs to perform planning. We will discuss the obtained plans below.

Representation in PDDL

As this scenario is directly taken from the IPC 2000 contest, the PDDL solution is available on the Internet. However, it is placed there in a more advanced form then needed for this particular problem so we will cite here only relevant parts of the solution. Let us start with domain file:

```
(define (domain BLOCKS)
    (:requirements :strips)
    (:predicates
        (on ?x ?y)
        (ontable ?x)
        (clear ?x)
        (handempty)
        (holding ?x)
    )
```

```
    (:action pick-up
        :parameters (?x)
        :precondition (and (clear ?x) (ontable ?x) (handempty))
        :effect
            (and (not (ontable ?x))
            (not (clear ?x))
            (not (handempty))
            (holding ?x)))

    (:action stack
        :parameters (?x ?y)
        :precondition (and (holding ?x) (clear ?y))
        :effect
            (and (not (holding ?x))
            (not (clear ?y))
            (clear ?x)
            (handempty)
            (on ?x ?y)))
```

The domain declares five relations: “on”, “ontable”, “clear”, “handempty” and “holding”. It can be noticed, that while ACTLOG solution uses negation extensively, the PDDL’s one tends to stay in a non-negative world and defines separate relation for each state (e.g. “handempty” instead of checking if “holding” is false). Apart from that the PDDL actions are quite similar to the ones defined in ACTLOG.

Let us now look at the problem file:

```
(define (problem BLOCKS-4-0)
    (:domain BLOCKS)
    (:objects D B A C )
    (:INIT
        (CLEAR C) (CLEAR A) (CLEAR B) (CLEAR D)
        (ONTABLE C) (ONTABLE A) (ONTABLE B) (ONTABLE D)
        (HANDEMPTY)
    )
    (:goal (AND (ON D C) (ON C B) (ON B A)))
)
```

The problem definition basically specifies the same as ACTLOG but in a different way - bricks here are explicitly added to object list while in ACTLOG they are automatically added to the appropriate domain in a module. Similarly as in Scenario 1, the initial state of the world is given in the problem while in ACTLOG’s approach it is passed via the module (and can even be inferred automatically).

When we run the problem in planning engine we obtain the following output:

```
IW search completed
0.00100: (pick-up b)
0.00200: (stack b a)
0.00300: (pick-up c)
0.00400: (stack c b)
0.00500: (pick-up d)
```

```
0.00600: (stack d c)
Planner found 1 plan(s) in 0.452secs.
```

Execution on the remote server does the job, the whole plan is ready within half a second. Let us now look at how ACTLOG is dealing with that problem.

Heuristics in the scenario

Let us try to execute the "bricks4" problem from the "ex13.4ql" file using the ACTLOG planner:

```
# import "../examples/ex13.4ql".
Program loaded!
# execute bricks4.
Executing planning for problem bricks4...

Plan found:
1. take(FROM_WHAT: table (literal), WHAT: b (literal))
2. put(WHAT: b (literal), ON_WHAT: a (literal))
3. take(FROM_WHAT: table (literal), WHAT: c (literal))
4. put(WHAT: c (literal), ON_WHAT: b (literal))
5. take(FROM_WHAT: table (literal), WHAT: d (literal))
6. put(WHAT: d (literal), ON_WHAT: c (literal))

Planning took 1562 milliseconds.
```

It seems that it is not so bad. We have an optimal plan obtained in less that 1.6 seconds (without any heuristics applied). However, there is a small trick used here - in the planning problem we have defined "max_depth" to be 6 which matches the length of the optimal plan.[4] As there are no other plans with such length, the planner correctly and quickly finds the optimal answer. If we change the value to, e.g., 10, the plans still remain correct, but are not optimal any more (similarly to Scenario 1):

```
Plan found:
1. take(FROM_WHAT: table (literal), WHAT: c (literal))
2. put(WHAT: c (literal), ON_WHAT: b (literal))
3. take(FROM_WHAT: b (literal), WHAT: c (literal))
4. put(WHAT: c (literal), ON_WHAT: d (literal))
5. take(FROM_WHAT: table (literal), WHAT: b (literal))
6. put(WHAT: b (literal), ON_WHAT: a (literal))
7. take(FROM_WHAT: d (literal), WHAT: c (literal))
8. put(WHAT: c (literal), ON_WHAT: b (literal))
9. take(FROM_WHAT: table (literal), WHAT: d (literal))
10. put(WHAT: d (literal), ON_WHAT: c (literal))

Planning took 6661 milliseconds.
```

Let us leave the planning depth limit set to 10 and see if we can help here with any of the heuristics.

[4] Note, that planners used in the international contests have a timeout features as well, e.g., the one that we are using for PDDL planning times out after 10 seconds by default.

“disallow_failed_preconditions” and “composite_first”

There are no composite actions in the scenario so these heuristics do not affect the output plan.

“random”

As stated in the previous Scenario, random planning does not provide repeatable good results. The more complex the planning problem and longer the output plans are, the less probable is that random guessing will “hit” the optimal spot. Even if planning time can be sometimes shortened a bit, the produced plans are usually still not optimal.

Plan length	Planning time (s)
10	27.047
8	27.527
10	30.359
10	75.145
10	7.998
10	36.618
10	4.652
10	1.434
10	38.282
10	63.113

Table 8.1: Comparison of 10 planning tries using Scenario 2 and “random” method.

Table 8.1 presents a comparison of 10 tries of random planning executions with obtained plan length and planning time. It is clearly visible that apart from the lack of optimal plans there, planning times can be really high - in practice the “random” heuristic is therefore nothing more than a nice benchmark for maximum planning times.

“goal_probing”

Probing the goal, although extremely efficient in some cases, in others can not only be unhelpful but even can make planning time much worse. When we look at planning goal “on(a, table), on(b, a), on(c, b), on(d, c)” then we can see, that the initial goal rating for it is 2 due to “on(a, table)”. Therefore although taking “a” to hand will have the smallest rating (equal to 0), all other actions will have exactly the same goal rating and the planning will not move in a correct direction (placing “c” on “b” is as good as placing “b” on “a”). In this sense, goal probing, despite good intentions, can even direct into unsuccessful paths.

Even if we introduce a composite template of action which automatically takes and puts the brick, it will not improve the situation as the same problem as before holds - placing “c” on “b” is as promising as placing “b” on “a”.[5]

[5]Goal probing proves itself to be extremely efficient in a similar problem of reversing a brick tower (“table -> c -> b -> a” to “table -> a -> b -> c”). The description and implementation can be found and tested using file “ex10.4ql” included into a planner package. The heuristic allows for limiting the planning time from almost 5 seconds to 180 milliseconds and always returns an optimal plan there. However, we have decided that showing limitations of the implementation is far more interesting than presenting only successful applications so we leave these experiments to the reader.

"increase_rating_only" and "strict_increase_rating_only"

If we look closely at the planning goal of this problem then we can see that the plan should fill the goal "from left to right", constantly increasing the rating of it. Unfortunately, we have two separate actions for taking and putting (where putting can actually increase the goal rating) which is why we cannot use "strict_increase_rating_only" directly ("No plan found (planning took 195 milliseconds)."). On the other hand, using "increase_rating_only" does not really help as the planning engine will stay for a while in the area where rating is equal to 2 and then will move towards the goal (plan will be correct but not optimal).

Fortunately, we know that if we take something we will always put it so we can make a hint to the planner about that. Let us extent the action set with following "template" action (we do not tell the planner anything about the valuations - we just let it know that "put" goes after "take"):

```
action takeAndPutComposite():
    composite:
        take(WHAT, FROM_WHAT); put(WHAT, ON_WHAT)
end.
```

This action is treated by the planning engine as a single atomic one which means that it should always increase the goal rating. Therefore "goal_probing" improved with "strict_increase_rating_only" is ideal to be used here:

```
Plan found:
1. takeAndPutComposite():
#>    take(FROM_WHAT: table (literal), WHAT: b (literal))
#>    put(WHAT: b (literal), ON_WHAT: a (literal))
2. takeAndPutComposite():
#>    take(FROM_WHAT: table (literal), WHAT: c (literal))
#>    put(WHAT: c (literal), ON_WHAT: b (literal))
3. takeAndPutComposite():
#>    take(FROM_WHAT: table (literal), WHAT: d (literal))
#>    put(WHAT: d (literal), ON_WHAT: c (literal))

Planning took 575 milliseconds.
```

The plan here will always be optimal and the time of computations is quite decent (max_depth is still set to 10) as for a relatively slow laptop.

Work smart, not hard

Using planning heuristics is nice but in some cases we could try to use the characteristics of the planning problem itself. In this particular case one can notice, that we would like to put a brick only on the top of a tower - starting with "a" being the top. So if we mark "a" as "top" in our planning environment, then we may upgrade the "put" action as follows:

```
action putOnTop(WHAT, ON_WHAT):
    preconditions:
        hold(WHAT) in {true, incons}, top(ON_WHAT).
    add:
        // our thing is now on sth...
        on(WHAT, ON_WHAT).
```

```
        top(WHAT).
    remove:
        // we are not holding it anymore...
        hold(WHAT).
        top(ON_WHAT).
end.
```

Using such an upgrade, planning algorithm will always keep a single tower top in planning status and therefore will know exactly where to put the next brick:

```
Plan found:
1. take(FROM_WHAT: table(literal), WHAT: b(literal))
2. putOnTop(WHAT: b(literal), ON_WHAT: a(literal))
3. take(FROM_WHAT: table(literal), WHAT: c(literal))
4. putOnTop(WHAT: c(literal), ON_WHAT: b(literal))
5. take(FROM_WHAT: table(literal), WHAT: d(literal))
6. putOnTop(WHAT: d(literal), ON_WHAT: c(literal))

Planning took 288 milliseconds.
```

Here we also get an optimal plan, without any heuristics enabled. Note that in practical applications one should give as much information about the problem to the planner as possible. If we know that we are building a tower, we can give that knowledge to an agent that will be planning how to do it.

8.3. Scenario 3 - defusing a bomb in ACTLOG

The scenario is available in the file "example14.4ql" in the inter4QL's package.

This is the most sophisticated scenario in the chapter - this is why we will not be providing a PDDL comparison for it. As we saw in previous two scenarios, PDDL sometimes cannot cope with problems in a way we would like it to. Moreover, we deal with partial and potentially inconsistent beliefs being outside of the scope of PDDL.

The scenario presents the following situation: we have a building with doors and paths between rooms. Doors may be opened or closed and paths may be blocked or not (we can assume that this is a war zone where some parts of the building have collapsed). What is important here, in the building some evil person has planted a bomb which may cause far bigger damage when exploded - this is why it has to be defused.

For achieving the goal we have a robot with two arms and two cameras capable of reaching the bomb and defusing it. However, the robot was organized in a quick manner (bomb is ticking) so the quality of its camera components is not the highest possible - the cameras often report different readings for the same places.

To defuse the bomb, the robot has to get to the room where it is located. There the robot will see that the bomb has 3 cables (red, green and blue) where one or many has to be cut to defuse the bomb. For the purposes of the simulation, the combination defusing the bomb consists of red and green cables cut at the same time. The bomb type is known, so the combination is also known during the problem specification.

Figure 8.1 shows the overview of building's plan with doors and paths visible. One can see that the building has 3 doors and 3 paths with following statuses:

- *door1* - both cameras report different statuses (door "open" status is inconsistent);

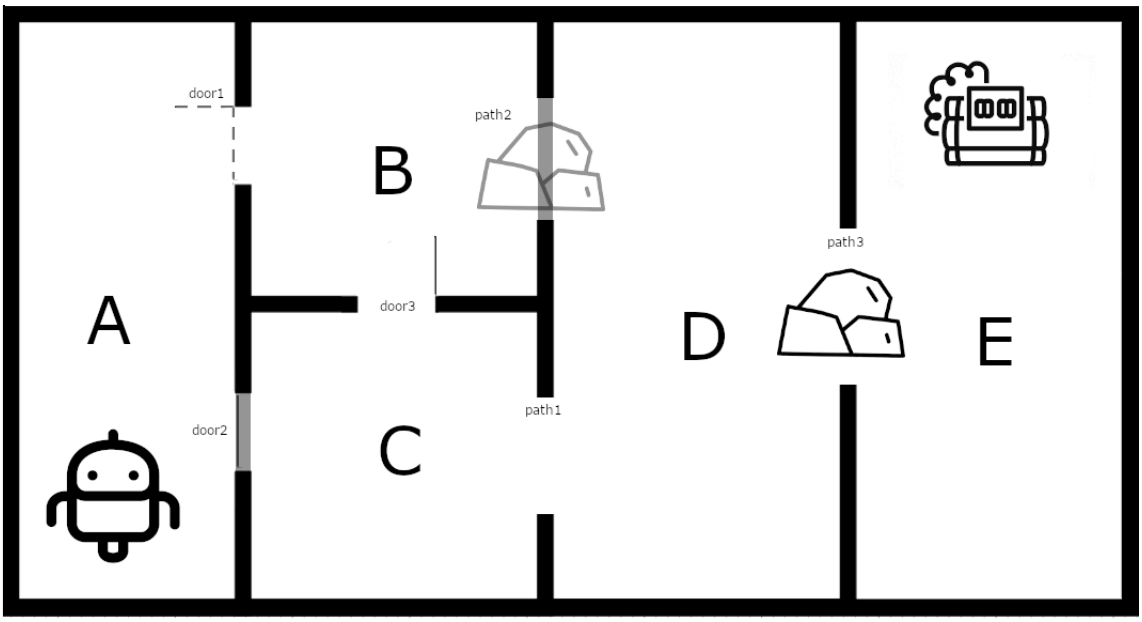

Figure 8.1: Building overview with reading errors visible.

- *door2* - one camera reports closed door and the other reports no door at all;
- *door3* - both cameras report open door;
- *path1* - both cameras report unblocked path;
- *path2* - one camera reports blocked path and the other does not report the path at all;
- *path3* - both cameras report the path as blocked.

One can notice, that reaching the goal in the building defined as in Figure 8.1 is not possible - the path from “D” to “E” is blocked. The robot has a belief not to move through blocked paths but when being close to the goal it can shadow it and try to ram the obstacle to get to the bomb (finalizing the mission is so important that robot damages do not count as much). Similarly, even though the robot may have inconsistent data about doors, it will still try to go through them. This is how it can get from room “A” to “B” even when both cameras report different information and effectively do not allow for deducing a consistent belief about the door’s state.

Representation in ACTLOG

Usually in robotic systems, we have an external sensor driver which is responsible for loading the data to the planning application. As we did not try to run this on an actual robot (yet), we need some 4QL modules to emulate the cameras. Here is the first camera:

```
module camera1:
    domains:
        literal place.
        literal id_door.
        literal id_path.
    relations:
        path(place, place, id_path).
```

```
        door(place, place, id_door).
        closed(id_door).
        blocked(id_path).
        bombAt(place).
    rules:
        door(X, Y, ID) :- door(Y, X, ID).
        path(X, Y, ID) :- path(Y, X, ID).
    facts:
        door(a, b, door1).
        door(b, c, door3).
        path(c, d, path1).
        path(b, d, path2).
        path(d, e, path3).
        -closed(door1).
        -closed(door3).
        -blocked(path1).
        blocked(path2).
        blocked(path3).
        bombAt(e).
end.
```

The second camera (truncated as domains, relations and rules are exactly the same) is:

```
module camera2:
    ...
    facts:
        door(a, b, door1).
        door(a, c, door2).
        door(b, c, door3).
        path(c, d, path1).
        path(d, e, path3).
        closed(door1).
        closed(door2).
        -closed(door3).
        -blocked(path1).
        blocked(path3).
        bombAt(e).
end.
```

Now we need to combine cameras into a single belief base to be then able to query for robot's beliefs about the world:

```
beliefs sensors:
    worlds:
        camera1.
        camera2.
end.
```

The camera modules alone do not include any inconsistency, but when combined into one belief base, they introduce quite chaotic image of the world:

```
# Bel[sensors](blocked(X)).
```

```
== Results ==
X: path2 = true
X: path3 = true
X: path1 = false
# Bel[sensors](closed(X)).
== Results ==
X: door1 = inconsistent
X: door3 = false
X: door2 = true
```

Please note, that even though camera 2 does not see “path2”, its existence is still reported due to the combination of inputs from both cameras.

Sensors are outside of our main planning route - they only provide data to the robot. We still need a belief base which will contain information about robot’s position and other important elements of the environment, like the bomb. Let us start with the world representation itself:

```
module world:
    ...
    relations:
        at(place).
        visited(place).
    ...
    facts:
        at(a).
end.
```

One can see that we have two main information chunks here: where the robot currently is and where it was in the past. The second information corresponds directly to the approach shared during analysis of the Scenario 2 stating, that if we know about some characteristics of a problem, we should give it to the planning engine to use. This is the case here: we know that robot does not have to return to the rooms it has already visited so we are explicitly marking those with a “visited” flag. This improves the planning performance substantially.

Now, as we are able to store the position of the robot, we should also have a place to keep information about the bomb:

```
module bomb:
    domains:
        literal color.
        literal place.
        literal object.
    relations:
        cut(color).
        ticking(object).
        defused(object).
    rules:
        -cut(X) :- color(X).
    facts:
        ticking(bomb).
        color(red).
        color(green).
```

```
        color(blue).
end.
```

As described in the scenario, we have three cables which can be cut to defuse the bomb. By default, all cables are not cut (their status is known so we close the world locally here). The question remains how to store the combination of the cables to be cut to defuse the bomb. It turns out, that together with creating a belief base for planning purposes, we can find the combination inside it:

```
beliefs environment:
    constraints:
        rigid:
            world.at(X) -> world.visited(X) = unknown.
            bomb.cut(red) = incons -> bomb.cut(green) = incons,
                    -bomb.cut(blue).
            bomb.cut(green) = incons -> bomb.cut(red) = incons,
                    -bomb.cut(blue).
            -bomb.cut(blue).
    worlds:
        bomb.
        world.
end.
```

There are several interesting aspects here. First of all one can see that the belief base contains two modules which provide a view on two complete different parts of the world. This has been already stated in Section 2.2 that worlds in a belief base can either provide complementary views on the same world (like above) or alternative views on the same world (like "sensors" belief base). This scenario provides both examples. Secondly, the rigid constraints of the belief base are here used to disallow the robot to return to already visited nodes. Please note, that this happens automatically, we do not need to provide any explicit rules. Finally, the bomb's defuse combination is also stored in the constraints here which will cause failure on each planning path where robot cuts other cables that red and green (what is important - at the same time).

There is still one environmental module left to be presented. As stated in scenario's description, the robot can ignore potential blockade on a path if it is close enough to the planning goal. To achieve this, we need a module that will provide altered (shadowed) information about statuses of paths:

```
module ignoreBlocked:
    domains:
        literal id_path.
    relations:
        blocked(id_path).
    facts:
        -blocked(path3).
        -blocked(path2).
        -blocked(path1).
end.
```

Using the above module the robot can shadow its belief about blockade status of path and try to force its way through an blocked path:

```
# Bel[sensors as ignoreBlocked](blocked(X)).
== Results ==
```

```
X: path1 = false
X: path3 = false
X: path2 = false
```

Now, having the environmental modules done, we may proceed to the actual actions - starting with the ones used for moving around the building:

```
action take_path(FROM_WHERE, TO_WHERE):
    preconditions:
        at(FROM_WHERE),
            Bel[sensors](path(FROM_WHERE, TO_WHERE, ID)),
            Bel[sensors](blocked(ID)) = false.
    add:
        at(TO_WHERE).
        visited(FROM_WHERE).
    remove:
        at(FROM_WHERE).
end.

action go_through_door(FROM_WHERE, TO_WHERE):
    preconditions:
        at(FROM_WHERE),
            Bel[sensors](door(FROM_WHERE, TO_WHERE, ID)),
            Bel[sensors](closed(ID)) = false.
    add:
        at(TO_WHERE).
        visited(FROM_WHERE).
    remove:
        at(FROM_WHERE).
end.
```

The actions state that the robot can go through the path if the path is believed to exist and not to be blocked. Of course it can take the path only if it is in a room "connected" with the path. Similarly, the robot can go through the door if the door is believed to exist and be opened.

Unfortunately, due to camera flaws there are not many useful doors and paths in the sense of the above actions. The door can either not exist or its status can be reported in an inconsistent way. In the situation of inconsistent status of existing door, the robot is instructed to try to go through the door:

```
action force_through_door(FROM_WHERE, TO_WHERE):
    preconditions:
        at(FROM_WHERE),
            Bel[sensors](door(FROM_WHERE, TO_WHERE, ID)),
            Bel[sensors](closed(ID)) = incons.
    add:
        at(TO_WHERE).
        visited(FROM_WHERE).
    remove:
        at(FROM_WHERE).
end.
```

As the door is usually constructed from soft materials, a robot (made of metal) can force through it without much harm. The situation with blocked paths is a bit different as the blockade is usually made out of rocks and bricks which may damage the outer armor of the robot. However, the robot is instructed to care more about the mission success than its own safety only if the goal is really close. This is why the robot is equipped with following action:

```
action force_through_path_to_goal(FROM_WHERE, TO_WHERE):
    preconditions:
        at(FROM_WHERE),
            Bel[sensors](bombAt(TO_WHERE),
                    path(FROM_WHERE, TO_WHERE, ID)),
            Bel[sensors as ignoreBlocked](blocked(ID)) = false.
    add:
        at(TO_WHERE).
        visited(FROM_WHERE).
    remove:
        at(FROM_WHERE).
end.
```

Please note, that robot's true beliefs about the environment are not changed - they are just shadowed for a while to allow it to ram through blocked path. This may happen only if the bomb is in the destination room - otherwise, the robot will try to find another way.

Finally, when the robot is in the room together with a bomb, it can proceed to defusing it. It has a following set of actions to be used here:

```
action cutCable(COLOR):
    preconditions:
        Bel[sensors](bombAt(X)),
            world.at(X), ticking(bomb),
            -cut(COLOR).
    add:
        cut(COLOR).
end.

action cutTwoCables():
    composite:
        cutCable(COLOR1) || cutCable(COLOR2)
end.
```

The "cutTwoCables" action is, just like in case of Scenario 2, just a template function which suggests the robot to try to cut two cables at the same time - it does not say anything about the colors (in fact COLOR1 can be equal to COLOR2, the action is not disallowing it).

Finally, after the job done the robot should notify the remote crew about the success (they will notice the failure by themselves):

```
action notifyDefuse():
    preconditions:
        cut(green) = incons, cut(red) = incons, -cut(blue).
    add:
        defused(bomb).
    remove:
```

```
        ticking(bomb).
end.
```

Please note, that the status after the defusing is in fact inconsistent. It is due to the fact, that the world was locally closed here and then we have actually cut the cables - by fusing true with false inconsistency appeared. It can be easily seen that this inconsistency is rather artificial. Indeed, it is kept here to emphasize that inconsistent goals of planning are nothing unusual and the ACTLOG planner can deal with them without any issues. The "notifyDefuse" action is added here only because it is always nice to report a success. The precondition of the action could be placed as planning goal in problem's definition and the planning will also work without any problems.

Finally, we need a planning problem:

```
problem defuse_bomb:
    beliefs:
        environment.

    actions:
        take_path.
        go_through_door.
        force_through_door.
        force_through_path_to_goal.
        cutCable.
        cutTwoCables.
        notifyDefuse.

    goal:
        defused(bomb)

    max_depth:
        10

    heuristics:
        none.
end.
```

The planning problem finalizes the "implementation" of the scenario in ACTLOG. Let us now look how it works.

Heuristics in the scenario

Running a planning process for a problem defined exactly as in previous subsection creates a correct, but inefficient plan:

```
Plan found:
1. cutTwoCables():
#>    cutCable() - failed preconditions
#>    cutCable() - failed preconditions
2. cutTwoCables():
#>    cutCable() - failed preconditions
#>    cutCable() - failed preconditions
```

```
3. cutTwoCables():
#>   cutCable() - failed preconditions
#>   cutCable() - failed preconditions
4. cutTwoCables():
#>   cutCable() - failed preconditions
#>   cutCable() - failed preconditions
5. force_through_door(FROM_WHERE: a (literal),
       TO_WHERE: b (literal))
6. go_through_door(FROM_WHERE: b (literal),
       TO_WHERE: c (literal))
7. take_path(FROM_WHERE: c (literal), TO_WHERE: d (literal))
8. force_through_path_to_goal(FROM_WHERE: d (literal),
       TO_WHERE: e (literal), )
9. cutTwoCables():
#>   cutCable(COLOR: red (literal))
#>   cutCable(COLOR: green (literal))
10. notifyDefuse()

Planning took 58909 milliseconds.
```

Also the time of planning is long (even for a rather slow laptop). Let us see whether heuristics will improve the situation. We will omit the "random" method as it already proved itself not to provide good results.

"composite_first"

As the final optimal plan uses composite actions at the very end of the planning, using this heuristic here causes a result completely opposite to the desired one. Just to indicate how bad this scenario with "composite_first" heuristic is, let us state, that the produced plan is exactly the same as in case of no heuristics at all but planning takes around 140 seconds.

"goal_probing"

Unluckily, in the case of such big scenarios which contain so isolated planning goal, the goal does not really reflect the statuses obtained at the beginning (or even in the middle) of the planning process. This is why goal probing does not have any chance here as most of the time it has no data to use while ordering the actions and valuations (goal rating is usually equal to 0). This makes the heuristic basically very similar to no heuristics at all (except from application of the last action, of course, but it is preceded by a huge tree of state searching built completely blindly). The same applies to "increase_rating_only" and "strict_increase_rating_only". Therefore with this scenario these heuristics appear not helpful.

"disallow_failed_preconditions"

When looking at the plan, the only issue with it seems to be caused by allowed empty preconditions for parallel actions. Therefore not surprisingly, disallowing that ends up in producing a correct and optimal plan in a reasonable time:

```
Plan found:
```

```
1. force_through_door(FROM_WHERE: a (literal),
       TO_WHERE: b (literal))
2. go_through_door(FROM_WHERE: b (literal),
       TO_WHERE: c (literal))
3. take_path(FROM_WHERE: c (literal), TO_WHERE: d (literal))
4. force_through_path_to_goal(FROM_WHERE: d (literal),
       TO_WHERE: e (literal), )
5. cutTwoCables():
#>   cutCable(COLOR: red (literal))
#>   cutCable(COLOR: green (literal))
6. notifyDefuse()

Planning took 1647 milliseconds.
```

In fact the "disallow_failed_preconditions" heuristic fixes all the issues with the action definitions. The planning problem is generally well defined and does not need additional support from e.g. goal probing. Please note, that the "visited" relation is directing the navigation around the building and actions have well-defined preconditions - all except from the "cutTwoCables" action. Please note that adding a simple precondition to it stating that this action is allowed to be applied only in rooms with a bomb will fix the problem as well. This shows the great responsibility of a user which can either work with a planning engine by specifying actions based on the understanding of planning principles or can work against the engine and leave big parts of the state space not guarded in any way. These unguarded spaces are where the planner can spend long time on searching for (possibly inefficient) solutions. Minimizing the state space with some heuristics is a good approach but there is no heuristic working with all planning problems as clearly shown in this chapter.

Chapter 9

Summary and conclusions

The thesis presents the results of four years of both theoretical and implementational work. All of the theoretical results achieved (and a part of a doxastic practical ones) have been published in four articles [18, 19, 21, 22] and received a positive feedback from a research community. The planner part was left as an original contribution to be presented in the thesis together with a set of planning experiments and problems.

Let us now briefly summarize all the results presented in the thesis.

9.1. Original contributions of the thesis' author

First of all, theoretical results included in papers [18, 19, 21, 22] have been obtained as an effect of cooperation with other co-authors. The thesis author's contribution to the papers depends on participating in discussions leading to the published concepts and their formalization as well as providing the published experimental results based on the author's implementation. Therefore the author's significant involvement into the key results of the research allows him to include the articles into this PhD thesis with an appropriate explanatory note.

The whole implementation and experimental work, including most of the scenarios and examples included in [19, 21, 22] as well as in the inter4QL interpreter are the original contributions of the thesis' author. Of course, there was a continuous interaction between implementation-oriented and theoretical work. In particular, the final syntax of proposed constructs is a good example of such an interaction.

9.2. Theoretical results achieved

When the work presented in the thesis has started, the logic underlying the inter4QL interpreter (version 3.2 back then) contained the full support for paraconsistent and paracomplete reasoning. At that same time an articles about belief bases, belief structures and epistemic profiles have been published [42, 43, 45, 90] (the author of the thesis was not involved in that work). This made a good opportunity to set the next steps: extending 4QL with the $\mathrm{Bel}()$ operator, proposing belief shadowing, defining the ACTLOG language and developing a planner.

To achieve the expected goals, first the logic had to be extended to allow reasoning and querying of belief bases which resulted in an 4QL's extension, $4QL^{Bel}$ [18]. With this part ready, the first version of ACTLOG, based on the $4QL^{Bel}$, has been proposed with atomic actions only [19]. Then, in 2018,

belief shadowing was introduced together with constraints for belief bases enclosed into 4QL$^{\text{Bel+}}$ language [21]. The 4QL$^{\text{Bel+}}$ language was then a base for extending the ACTLOG language with parallel actions in 2019 [22] which became a final logical background for this PhD thesis.

Of course throughout all of the articles the tractability of the reasoning and querying of the belief bases was underlined as this is a very important feature of the solutions. Tractability of quite advanced features is an important motivation of the presented approach.

9.3. The practical results

The development of the inter4QL interpreter matches strongly the article publishing milestones from the previous section. When 4QL$^{\text{Bel+}}$ language was published, the implementation for version 4.0 has been finalized. The version included a support for belief bases, $\mathrm{Bel}()$ operator, belief shadowing and constraints for both modules and belief bases. Belief bases were not changing module's well-supported models, they were aggregating them. Also the lightweight belief shadowing was not changing the knowledge base but rather provided a support for smart queries to the appropriate belief bases/modules. On top of that a condition for constraints had to be added to querying methods and the full coverage for 4QL$^{\text{Bel+}}$ could be provided.

Next, when the improved ACTLOG was published, the second phase of the implementation was started. As the articles did not say anything about the actual planning algorithms (except from an overview of action application method), several design decisions had to be made balancing the complexity of the implementation and the usability of the solution. As a result, in December 2019 the inter4QL 5.0 has been proposed as a support for composite and atomic actions. Soon it became clear that the blind planning used in this first implementation, although providing correct plans, took a significant amount of time and could produce inefficient plans in the cases when planning problems were really broad. This is why five months later versions 5.1 and 5.2 were proposed containing a set of planning heuristics which improve the planner's performance.

9.4. Further development possibilities

Currently the planner covers all of the research requirements set in the articles gathered in this thesis. When the thesis is finalized, the planner is half a year old but even now can approach problems from planning contests. The list of the possible improvements include:

- unifying the memory management in ModuleProblem.cc and the rest of the planner;
- introduction of multi-threading to the planner;
- implementation of other planning algorithms - like A* or BFS.

The planning performance is what makes a planning engine usable in practice. The performance area of improvements is very broad here. First of all, the planner (just like the original inter4QL) is a single-thread application which in the current state of technology prevents it from being used in a larger scale applications. Also, sacrificing some of the simplicity in storing of the planning state, some of the more efficient state-search algorithms could be used instead of the DFS. E.g., BFS could be used instead. Though it uses more memory during computations and needs to keep copies of whole databases as a planning state, it will always find an optimal solution wrt the restricted plan's size.

The area of planning heuristics is also worth further investigation, for example allowing for an automatic heuristic selection. Right now the provider of the planning problem needs to understand it and select the heuristic matching the issues observed during the planning. Possibly this process can be somehow automated.

Also, the underlying logic itself has a broad potential for further expansion. The set of composite actions can be expanded, e.g., with finite loops. The planning itself is also not considering full belief structures now - possibly this is a path that can be checked.

Definitely, there is a room for further development of the 4QL language family and inter4QL itself.

Bibliography

[1] Abe, J., Nakamatsu, K.: Manipulating paraconsistent knowledge in multi-agent systems. In: Agent and Multi-Agent Systems: Technologies and Applications. pp. 159–168. Springer Berlin Heidelberg (07 2007)

[2] Abe, J., Nakamatsu, K.: Multi-agent systems and paraconsistent knowledge. Studies in Computational Intelligence 170, 101–121 (12 2008)

[3] Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley Pub. Co. (1996)

[4] Alchourrón, C.E., Gärdenfors, P., Makinson, D.: On the logic of theory change: Partial meet contraction and revision functions. Journal of Symbolic Logic 50(2), 510–530 (1985)

[5] de Amo, S., Pais, M.: A paraconsistent logic approach for querying inconsistent databases. International Journal of Approximate Reasoning 46, 366–386 (2007)

[6] Angelotti, E., Scalabrin, E., Ávila, B.: PANDORA: a multi-agent system using paraconsistent logic. In: Proceedings Fourth International Conference on Computational Intelligence and Multimedia Applications. ICCIMA 2001. pp. 352–356 (02 2001)

[7] Aucher, G.: Generalizing AGM to a multi-agent setting. Logic Journal of the IGPL 18(4), 530–558 (2010)

[8] Bacchus, F., Petrick, R.P.A.: Modeling an agent's incomplete knowledge during planning and during execution. In: Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning. p. 432–443. KR'98, Morgan Kaufmann Publishers Inc. (1998)

[9] Baioletti, M., Milani, A., Poggioni, V., Suriani, S.: A multivalued logic model of planning. Frontiers in Artificial Intelligence and Applications 141, 575–579 (01 2006)

[10] Balzer, R.: Tolerating inconsistency. In: Proceedings of the 13th International Conference on Software Engineering. p. 158–165. ICSE '91, IEEE Computer Society Press, Washington, DC, USA (1991)

[11] Bazzotti, G., Gerevini, A.E., Lipovetzky, N., Percassi, F., Saetti, A., Serina, I.: Iterative Width Search for Multi Agent Privacy-Preserving Planning: XVIIth International Conference of the Italian Association for Artificial Intelligence, Trento, Italy, November 20–23, 2018, Proceedings, pp. 431–444 (11 2018)

[12] Belnap, N.: A useful four-valued logic. In: Eptein, G., Dunn, J. (eds.) Modern Uses of Many Valued Logic. pp. 8–37. Reidel (1977)

[13] Bertoli, P., Cimatti, A., Roveri, M., Traverso, P.: Planning in nondeterministic domains under partial observability via symbolic model checking. IJCAI International Joint Conference on Artificial Intelligence (05 2001)

[14] Bertossi, L.E., Hunter, A., Schaub, T. (eds.): Inconsistency Tolerance, LNCS, vol. 3300. Springer, Heidelberg, New York (2005)

[15] Bertossi, L.E., Hunter, A., Schaub, T.: Introduction to inconsistency tolerance. In: Inconsistency Tolerance [14], pp. 1–14

[16] Béziau, J.Y., Carnielli, W., Gabbay, D. (eds.): Handbook of Paraconsistency. College Publications (2007)

[17] Białek, Ł.: Development of an interpreter of a rule-based query language 4QL. M.Sc. thesis, University of Warsaw (2013)

[18] Białek, Ł., Dunin-Kęplicz, B., Szałas, A.: Rule-based reasoning with belief structures. In: Kryszkiewicz, M., et.al. (eds.) Proc. ISMIS'2017. LNAI, vol. 10352, pp. 229–239. Springer (2017)

[19] Białek, Ł., Dunin-Kęplicz, B., Szałas, A.: Towards a paraconsistent approach to actions in distributed information-rich environments. In: Ivanović, M., Bădică, C., Dix, J., Jovanović, Z., Malgeri, M., Savić, M. (eds.) Proc. IDC - Intelligent Distributed Computing XI. Studies in Computational Intelligence, vol. 737, pp. 49–60. Springer (2017)

[20] Białek, Ł., Szałas, A., Borkowski, A., Gnatowski, M., Borkowska, M.M., Dunin-Kęplicz, B., Szklarski, J.: Coordinating multiple rescue robots. Prace Naukowe Politechniki Warszawskiej. Elektronika z. 194, t. 1, 185–194 (2014)

[21] Białek, Ł., Dunin-Kęplicz, B., Szałas, A.: Belief shadowing. In: EMAS@AAMAS (2018)

[22] Białek, Ł., Dunin-Kęplicz, B., Szałas, A.: A paraconsistent approach to actions in informationally complex environments. Annals of Mathematics and Artificial Intelligence (05 2019)

[23] Białek, Ł., Szklarski, J., Borkowska, M., Gnatowski, M.: Reasoning with Four-Valued Logic in Multi-robotic Search-and-Rescue Problem, vol. 440, pp. 483–499. Springer International Publishing, Cham (01 2016)

[24] Bochman, A.: A Logical Theory of Nonmonotonic Inference and Belief Change. Springer (2001)

[25] Bonet, B., Geffner, H.: Planning with incomplete information as heuristic search in belief space. Proceedings of the 6th International Conference on Artificial Intelligence in Planning Systems (AIPS) pp. 52–61 (01 2000)

[26] Brenner, M.: A multiagent planning language. In: In Proc. of ICAPS'03 Workshop on PDDL (06 2003)

[27] Bułanowski, A.: Implementation of indeterministic belif structures: paraconsistent approach. M.Sc. thesis, University of Warsaw (2015)

[28] Chellas, B.F.: Modal Logic: An Introduction. Cambridge University Press (1980)

[29] Cholvy, L., Hunter, A.: Information fusion in logic: A brief overview. In: Gabbay, D., Kruse, R., Nonnengart, A., Ohlbach, H. (eds.) Qualitative and Quantitative Practical Reasoning, Proc. ECSQARU-FAPR'97. LNCS, vol. 1244, pp. 86–95. Springer (1997)

[30] Cholvy, L., Hunter, A.: Merging requirements from a set of ranked agents. Knowledge-Based Systems 16(2), 113–126 (2003)

[31] daCosta, N., Bueno, O.: Belief change and inconsistency. Logique & Analyse 41(161-163), 31–56 (1998)

[32] De Silva, L., Padgham, L.: Planning on demand in bdi systems. In: International Conference on Automated Planning and Scheduling. University of Southern California (2005)

[33] Dechter, R.: Constraint Processing. Morgan Kaufmann (2003)

[34] Dixon, S.E.: Belief revision: A computational approach. PhD thesis, University of Sydney (1994)

[35] Doherty, P., Kvarnström, J.: TALplanner: A temporal logic based forward chaining planner. Annals of Mathematics and Artificial Intelligence 30, 119–169 (2001)

[36] Doherty, P., Kvarnström, J.: TALplanner: A temporal logic-based planner. AI Magazine 22(3), 95–102 (2001)

[37] Doherty, P., Kvarnström, J.: Temporal action logics. In: Lifschitz, V., van Harmelen, F., Porter, F. (eds.) The Handbook of Knowledge Representation. pp. 709–757. Elsevier (2008)

[38] Doherty, P., Kvarnström, J., Szałas, A.: Temporal composite actions with constraints. In: Brewka, G., Eiter, T., McIlraith, S. (eds.) Proc. 13th Int. Conf. KR: Principles of Knowledge Representation and Reasoning. pp. 478–488. AAAI Press (2012)

[39] Doherty, P., Szałas, A.: Stability, supportedness, minimality and Kleene Answer Set Programs. In: Eiter, T., Strass, H., Truszczyński, M., Woltran, S. (eds.) Advances in Knowledge Representation, Logic Programming, and Abstract Argumentation, LNCS, vol. 9060, pp. 125–140. Springer International Publishing (2015)

[40] Dunin-Kęplicz, B., Strachocka, A.: Tractable inquiry in information-rich environments. In: Proc. 24th IJCAI. pp. 53–60 (2015)

[41] Dunin-Kęplicz, B., Strachocka, A.: Paraconsistent argumentation schemes. Web Intelligence 14, 43–65 (2016)

[42] Dunin-Kęplicz, B., Szałas, A.: Epistemic profiles and belief structures. In: Proc. KES-AMSTA 2012: Agents and Multi-agent Systems: Technologies and Applications. LNCS, vol. 7327, pp. 360–369. Springer (2012)

[43] Dunin-Kęplicz, B., Szałas, A.: Taming complex beliefs. Transactions on Computational Collective Intelligence XI LNCS 8065, 1–21 (2013)

[44] Dunin-Kęplicz, B., Szałas, A.: Indeterministic belief structures. In: Jezic, G., Kusek, M., Lovrek, I., J. Howlett, J., Lakhmi, J. (eds.) Agent and Multi-Agent Systems: Technologies and Applications: Proc. 8th Int. Conf. KES-AMSTA, pp. 57–66. Springer (2014)

[45] Dunin-Kęplicz, B., Szałas, A.: Indeterministic belief structures. In: Jezic, G., et.al. (eds.) Agent and Multi-Agent Systems: Technologies and Applications, Adv. in Int. Syst. Comp., vol. 296, pp. 57–66. Springer (2014)

[46] Dunin-Kęplicz, B., Szałas, A., Verbrugge, R.: Tractable reasoning about group beliefs. In: Dalpiaz, F., Dix, J., van Riemsdijk, M. (eds.) Engineering Multi-Agent Systems: 2nd Int. Workshop, EMAS 2014, pp. 328–350. Springer (2014)

[47] Dunin-Kęplicz, B., Verbrugge, R.: Teamwork in Multi-Agent Systems. A Formal Approach. John Wiley & Sons, Ltd. (2010)

[48] Edelkamp, S., Hoffmann, J.: PDDL2: The language for the classical part of the 4th international planning competition. Proceedings of the 4th International Planning Competition (01 2004)

[49] Eiter, T., Faber, W., Leone, N., Pfeifer, G., Polleres, A.: Planning under incomplete knowledge. In: Lloyd, J., Dahl, V., Furbach, U., Kerber, M., Lau, K.K., Palamidessi, C., Pereira, L., Sagiv, Y., Stuckey, P. (eds.) Proc. Computational Logic: 1st Int. Conf. pp. 807–821. Springer (2000)

[50] Fagin, R., Halpern, J., Moses, Y., Vardi, M.: Reasoning About Knowledge. MIT Press (2003)

[51] Fermé, E.: On the Logic of Theory Change : Extending the AGM Model. PhD thesis, KTH Royal Institute of Technology (2011)

[52] Fermé, E., Hansson, S.O.: AGM 25 years: Twenty-five years of research in belief change. J. Philosophical Logic 40(2), 295–331 (2011)

[53] Ferraris, P., Lifschitz, V.: On the minimality of stable models. In: Balduccini, M., Son, T. (eds.) Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning. LNCS, vol. 6565, pp. 64–73. Springer (2011)

[54] Fikes, R.E., Nilsson, N.J.: STRIPS: A new approach to the application of theorem proving to problem solving. In: Proceedings of the 2Nd International Joint Conference on Artificial Intelligence. pp. 608–620. IJCAI'71, Morgan Kaufmann Publishers Inc. (1971)

[55] Flouris, G., Plexousakis, D., Antoniou, G.: On generalizing the AGM postulates. In: Proceedings of the 2006 Conference on STAIRS 2006: Proceedings of the Third Starting AI Researchers' Symposium. pp. 132–143. IOS Press (2006)

[56] Frank, J., Jónsson, A.: Constraint-based attribute and interval planning. Constraints 8(4), 339–364 (2003)

[57] Gärdenfors, P.: Conditionals and changes of belief. Acta Philosophica Fennica 30, 381–404 (1978)

[58] Gelfond, M., Lifschitz, V.: Action languages. ETAI 3 (04 1999)

[59] Gerevini, A., Long, D.: Plan constraints and preferences in PDDL3 - the language of the fifth international planning competition. Tech. rep. (2005)

[60] Hadley, R.F.: The many uses of 'belief' in AI. Minds and Machines 1(1), 55–73 (1991)

[61] Hansson, S.O.: Taking belief bases seriously. In: Prawitz, D., Westerståhl, D. (eds.) Logic and Phil. of Sci. in Uppsala, pp. 13–28. Springer (1994)

[62] Hansson, S.O.: Revision of belief sets and belief bases. In: Dubois, D., Prade, H. (eds.) Belief Change, pp. 17–75. Springer (1998)

[63] Hansson, S.O.: A Textbook of Belief Dynamics. Theory Change and Database Updating. Kluwer Academic Publishers (1999)

[64] van Harmelen, F., Lifschitz, V., Porter, B.: Handbook of Knowledge Representation. Elsevier (2007)

[65] Helmert, M.: Changes in PDDL 3.1. Unpublished summary, IPC-2008 website (2008)

[66] Herzig, A., Rifi, O.: Propositional belief base update and minimal change. Artificial Intelligence 115(1), 107 – 138 (1999)

[67] Hewitt, C.: Formalizing common sense for scalable inconsistency-robust information integration using Direct Logic reasoning and the actor model. arXiv:0812.4852 (2008)

[68] Hewitt, C., Woods, J. (eds.): Inconsistency Robustness. College Pub. (2015)

[69] van Hoeve, W.J., Katriel, I.: Global constraints. Foundations of AI 2, 169 – 208 (2006)

[70] Huber, F.: Formal representations of belief. In: Zalta, E.N. (ed.) The Stanford Enc. of Philosophy. Stanford University, Spring 2016 edn. (2016)

[71] Hunter, A., Nuseibeh, B.: Managing inconsistent specifications: Reasoning, analysis and action. ACM Transactions on Software Engineering and Methodology 7 (08 1996)

[72] Jayakumar, B.: Handling Inconsistency in Knowledge Bases. PhD thesis, Georgia State University (2017)

[73] Kakas, A., Miller, R., Toni, F.: Planning with incomplete information. CoRR cs.AI/0003049 (03 2000)

[74] Katarzyniak, R., Pieczyńska, A.: The outline of the strategy for solving knowledge inconsistencies in a process of agents' opinions integration. In: Proceedings of the 6th International Conference on Computational Science - Volume Part III. p. 891–894. ICCS'06, Springer-Verlag (2006)

[75] Kim, J.H.: Convince: A Conversational Inference Consolidation Engine. PhD thesis, University of California, Los Angeles (1983)

[76] Konieczny, S., Lang, J., Marquis, P.: Da^2 merging operators. Artif. Intell. 157(1-2), 49–79 (2004)

[77] Konieczny, S., Pino Pérez, R.: Merging with integrity constraints. In: Hunter, A., Parsons, S. (eds.) Proc. ECSQARU'99 Conf.: Symbolic and Quantitative Approaches to Reasoning and Uncertainty. LNCS, vol. 1638, pp. 233–244. Springer (1999)

[78] Konieczny, S., Pino Pérez, R.: Merging information under constraints: A logical framework. J. Log. Comput. 12(5), 773–808 (2002)

[79] Kovacs, D.L.: BNF definition of PDDL3.1: completely corrected, without comments. Unpublished manuscript, IPC-2011 website. (2011)

[80] Kovacs, D.L.: BNF definition of PDDL3.1: partially corrected, with comments/explanations. Unpublished manuscript, IPC-2011 website. (2011)

[81] Kushmerick, N., Hanks, S., Weld, D.: An algorithm for probabilistic planning. Artificial Intelligence 76 (03 2002)

[82] Lang, J.: Belief update revisited. In: Proc. 20th IJCAI. pp. 2517–2522. Morgan Kaufmann (2007)

[83] Lever, J., Richards, B.: parcPlan: a planning architecture with parallel actions, resources and constraints. In: Raś, Z.W., Zemankova, M. (eds.) Methodologies for Intelligent Systems. pp. 213–222. Springer Berlin Heidelberg, Berlin, Heidelberg (1994)

[84] Liberatore, P.: The complexity of belief update. Artificial Intelligence 119(1), 141 – 190 (2000)

[85] Liberatore, P.: A framework for belief update. In: Ojeda-Aciego, M., de Guzmán, I.P., Brewka, G., Pereira, L.M. (eds.) Proc. JELIA 2000, pp. 361–375. Springer Berlin Heidelberg, Berlin, Heidelberg (2000)

[86] Liu, C., Kuehlmann, A., Moskewicz, M.: CAMA: a multi-valued satisfiability solver. IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers pp. 326– 333 (12 2003)

[87] Łukaszewicz, W.: Non-Monotonic Reasoning: Formalization of Commonsense Reasoning. Ellis Horwood Series in Artificial Intelligence, Ellis Horwood Limited (1990)

[88] Małuszyński, J., Szałas, A.: Living with inconsistency and taming nonmonotonicity. In: de Moor, O., Gottlob, G., Furche, T., Sellers, A. (eds.) Datalog 2.0. LNCS, vol. 6702, pp. 384–398. Springer-Verlag (2011)

[89] Małuszyński, J., Szałas, A.: Logical foundations and complexity of 4QL, a query language with unrestricted negation. Journal of Applied Non-Classical Logics 21(2), 211–232 (2011)

[90] Małuszyński, J., Szałas, A.: Partiality and inconsistency in agents' belief bases. In: Barbucha, D., Le, M., Howlett, R., Jain, L. (eds.) KES-AMSTA. Frontiers in Artificial Intelligence and Applications, vol. 252, pp. 3–17. IOS Press (2013)

[91] Maluszynski, J., Szalas, A.: Logical foundations and complexity of 4ql, a query language with unrestricted negation. Journal of Applied Non-Classical Logics 21 (11 2010)

[92] Marchi, J., Bittencourt, G., Perrussel, L.: A syntactical approach to belief update. In: Gelbukh, A., et.al. (eds.) Proc. MICAI 2005, pp. 142–151. Springer, Berlin, Heidelberg (2005)

[93] McCann, H., Bratman, M.: Intention, plans, and practical reason. Noûs 25, 230 (04 1991)

[94] McCarthy, J., Hayes, P.J.: Some philosophical problems from the standpoint of artificial intelligence. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence 4, pp. 463–502. Edinburgh University Press (1969), reprinted in McC90

[95] McDermott, D.M.: The 1998 AI planning systems competition. AI Magazine 21(2), 35 (Jun 2000)

[96] Mcilraith, S., Fadel, R.: Planning with complex actions. In: In Proc. NMR'02. pp. 356–364 (2002)

[97] Meseguer, P., Rossi, F., Schiex, T.: Soft constraints. In: Rossi et al. [115], pp. 281 – 328

[98] Meyer, J.J.C., van der Hoek, W.: Epistemic Logic for AI and Theoretical Computer Science. Cambridge University Press (1995)

[99] Mueller, E.: Commonsense Reasoning. Morgan Kaufmann (2006)

[100] Newell, A., Simon, H.A.: Computer simulation of human thinking. Science 134(3495), 2011–2017 (1961)

[101] Nguyen, N.T.: Processing inconsistency of knowledge in determining knowledge of a collective. Cybernetics and Systems 40(8), 670–688 (2009)

[102] Pearl, J.: Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1988)

[103] Pednault, E.P.D.: ADL: Exploring the middle ground between STRIPS and the Situation Calculus. In: Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning. p. 324–332. Morgan Kaufmann Publishers Inc. (1989)

[104] Peot, M.A., Smith, D.E.: Conditional nonlinear planning. In: Hendler, J. (ed.) Artificial Intelligence Planning Systems, pp. 189 – 197. Morgan Kaufmann, San Francisco (CA) (1992)

[105] Peppas, P.: Belief revision. In: van Harmelen, F., Lifschitz, V., Porter, B. (eds.) Handbook of KR, pp. 317–359. Elsevier (2008)

[106] Perlis, D.: The Role(s) of Belief in AI, p. 361–374. Kluwer Academic Publishers (2000)

[107] Petrick, R., Bacchus, F.: A knowledge-based approach to planning with incomplete information and sensing. Proc. AIPS 2002 (04 2002)

[108] Petrick, R., Bacchus, F.: Extending the knowledge-based approach to planning with incomplete information and sensing. Proc. ICAPS 2004 pp. 613–622 (01 2004)

[109] Pigozzi, G.: Belief merging and judgment aggregation. In: Zalta, E. (ed.) The Stanford Enc. of Philosophy. Stanford University, winter 2016 edn. (2016)

[110] Priest, G.: Special issue on impossible worlds. Notre Dame Journal of Formal Logic 38(4), 481–660 (1997)

[111] Priest, G.: Paraconsistent belief revision. Theoria 67(3), 214–228 (2001)

[112] Pryor, L., Collins, G.: Planning for contingencies: A decision-based approach. J. Artif. Int. Res. 4(1), 287–339 (May 1996)

[113] Regnier, P., Fade, B.: Complete determination of parallel actions and temporal optimization in linear plans of action. In: European Workshop on Planning. pp. 100–111. Springer Berlin Heidelberg, Berlin, Heidelberg (1991)

[114] Roos, N.: A logic for reasoning with inconsistent knowledge. Artificial Intelligence 57, 69–103 (09 1992)

[115] Rossi, F., van Beek, P., Walsh, T. (eds.): Handbook of Constraint Programming, Foundations of AI, vol. 2, Supplement C. Elsevier (2006)

[116] Russell, Norvig, S.: Artificial intelligence: A modern approach. Prentice Hall, Englewood Cliffs, NJ (01 2010)

[117] Sakama, C., Inoue, K.: An alternative approach to the semantics of disjunctive logic programs and deductive databases. J. Autom. Reasoning 13(1), 145–172 (1994)

[118] Santos, Y.D., Ribeiro, M.M., Wassermann, R.: Between belief bases and belief sets: Partial meet contraction. In: Proc. 2015 Int. Conf. on Defeasible and Ampliative Reasoning. DARe'15, vol. 1423, pp. 50–56. CEUR-WS.org (2015)

[119] Shachter, R.D.: Evaluating influence diagrams. Operations Research 34(6), 871–882 (1986)

[120] Shepherdson, J.: Negation in logic programming. In: Minker, J. (ed.) Foundations of Deductive Databases and Logic Programming, pp. 19–88. Morgan Kaufmann (1988)

[121] Smith, D., Weld, D.: Conformant graphplan. Proceedings of the National Conference on Artificial Intelligence pp. 889–896 (01 1998)

[122] Soininen, T., Niemelä, I.: Developing a declarative rule language for applications in product configuration. In: Gupta, G. (ed.) Proc. PADL'99. LNCS, vol. 1551, pp. 305–319. Springer (1999)

[123] Spanily, P.: Interpreter for four-valued rule-based query language 4QL. M.Sc. thesis, University of Warsaw (2011)

[124] Szałas, A.: How an agent might think. Logic Journal of the IGPL 21(3), 515–535 (2013)

[125] Szklarski, J., Białek, Ł., Szałas, A.: Paraconsistent reasoning in cops and robber game with uncertain information: A simulation-based analysis. International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems 27(03), 429–455 (2019)

[126] Tanaka, K.: The AGM theory and inconsistent belief change. Logique & Analyse 48(189-192), 113–150 (2005)

[127] Testa, R.R., Coniglio, M.E., Ribeiro, M.M.: Paraconsistent belief revision based on a formal consistency operator. CLE E-Prints 15(8), 01–11 (2015)

[128] Testa, R.R., Coniglio, M.E., Ribeiro, M.M.: AGM-like paraconsistent belief change. Logic Journal of the IGPL 25(4), 632–672 (2017)

[129] Thrun, S., Burgard, W., Fox, D.: Probabilistic Robotics (Intelligent Robotics and Autonomous Agents). MIT Press (2005)

[130] Weld, D., Anderson, C., Smith, D.: Extending graphplan to handle uncertainty & sensing actions. Proc. AAAI-98 (07 1998)

[131] Wilkins, D.E.: Domain-independent planning representation and plan generation. Artificial Intelligence 22(3), 269 – 301 (1984)

[132] Wilkins, D.E., Myers, K.L., Lowrance, J.D., Wesley, L.P.: Planning and reacting in uncertain and dynamic environments. Journal of Experimental & Theoretical Artificial Intelligence 7(1), 121–152 (1995)

[133] Winner, E., Veloso, M.M.: Automatically acquiring planning templates from example plans. Proceedings of AIPS'02 Workshop on Exploring Real-World Planning p. 69–74

[134] Wolverton, M.: Prioritizing planning decisions in real-world plan authoring. Proceedings of the ICAPS-04 Workshop on Connecting Planning Theory with Practice (2004)

[135] Wooldridge, M.: Reasoning About Rational Agents. MIT Press (2000)

[136] Younes, H.L.S., Littman, M.L.: PPDDL1.0: An extension to PDDL for expressing planning domains with probabilistic effects. Tech. rep. (04 2004)

[137] Zadeh, L.: Fuzzy sets. Information and Control 8, 333–353 (1965)

[138] Zhang, D.: Inconsistency in multi-agent systems. Advances in Intelligent and Soft Computing 122 (12 2011)