Calculating End Game Databases for General Game Playing

Arsen Kostenko
Master thesis
European Master in Computational Logic
Fakultät Informatik, Technische Universität Dresden
Facultad de Informática, Universidad Politécnica de Madrid

October 22, 2007

Part I
Authorship

Author:	Arsen Kostenko
Student number:	3341380
Title:	Calculating End Game Databases for General Game Playing
Dergee:	Master of Science
Date of submission:	23.10.2007

Hereby I do confirm that the thesis was prepared by me independently. I do also confirm that only the references and auxiliary means indicated in the thesis were used.

Signature of the author

Part II
Preface

The work presented by current paper was performed within scope of European Master program in Computational Logic (EMCL), sponsored by European Commission.

During research on the given topic efforts were focused on both theory and practice. The theoretical aspect of the work is presented in writing that follows. Whereas the practical implementation of the theoretical findings could be found on the enclosed CD, or inside the FLUXPlayer source repository, upon permission of FLUXPlayer maintainers.

Part III
Acknowledgements

The work presented in the current paper is a concluding link in a two-year long chain of events.

Since studies, as well as any other sane human activity, do not happen in isolation , there are number of persons/organizations that helped the event chain to happen. The contributors came from both academical and non-academical spheres.

This part of paper mentions just a small part of all those who helped (deliberately or not) the work to come into being. The list has neither explicit nor implicit order or priority, just the sequence in which appropriate contributors arose in the authors mind. The author would like to thank to:

European Commission for giving him a valuable chance to participate in the EMCL program.

Prof. Steffen Hölldobler and *Prof. Luís Pereira* for launching the program itself.

Prof. Michael Thielscher and *Stephan Schiffel* for guiding the work and filling the missing pieces of puzzle.

Susana Muñoz Hernández for being a coach during the first steps within the program and a friend through all of the program.

Prof. Manuel Carro Liñares for showing that prolog-programming could be real fun and supporting/assisting in numerous ways.

Julia Koppenhagen for empowering the internals of the program at TU-Dresden and making unofficial part of the studies a real pleasure.

Sylvia Epp for coordinating the program in TU-Dresden and bearing all the inconveniences of running the international program.

Daniel Rubio Bonilla for being an open-minded interlocutor and open-hearted friend as well as cautious driver.

Iván Pérez Domínguez for all the provocative, yet inspiring ideas and coffee-time conversations.

Anna Bonilla and *Jesus Rubio* for their generousness, kindness, and hospitality.

Oleksandra Gnatush for her critical judgements and rational thinking.

Atif Iqbal for expanding the cultural horizons as well as being an understanding and helping class-/flat- mate.

Novak Novaković for his assistance in fighting LaTeX and complexity, hidden in the notations of the scientific papers.

Cédric Phillipe for proving that being clever does not imply being boring.

His sister, *Vira*, and mother, *Iryna Kostenko*, for the patience, condolence and understanding during those two years.

His father, *Anatoliy Kostenko*, for remaining his silent listener and providing the critical point of view.

His classmates from the first and the second year of studies, for all the unforgettable moments they spent together.

Oleksiy Kukhar and *the Ukrainian students community* in Dresden for all the support exerted from the very first day.

Iryna Sabadash for understanding.

Contents

List of Tables

Abstract

The issue of end game database construction is known and well-studied in various specific gaming domains like the Chess and the Checkers. However with arrival of General Game Playing (GGP) challenge the automatic construction of end game databases attracted focus again. The current paper presents the sound and complete algorithm for automated construction of end game databases within the GGP setting. The theoretical base for the algorithm stands on the regression introduced in the Situation Calculus. Along with the theoretical base a sample implementation constructed on top of the FLUX (a Fluent Calculus implementation) is presented. Discussion of current open issues and future challenges concludes the writing.

Part IV
The Work

1 Introduction

The idea of computers playing games emerged even earlier than the term of Artificial Intelligence (AI) as it was coined by John McCarthy in 1956. For instance, some of the checkers-playing programs date back to 1951, when they were created in University of Manchester. Unlike in case with an original Imitation Game as part of Turing test, it took much longer for AI to become mature in this field. Right now most of the popular games (Chess, Checkers, Othello, etc.) already have world-champion class playing programs, which employ various domain-specific AI techniques.

In any case games do remain a certain guideline for AI research. The reason why games are interesting for a human (and therefore for AI) is twofold. On one hand games involve evaluation and planning, which originally had no predefined algorithms or methods and had to be discovered by experience. On the other hand, games do remain quite complex tasks, so neither computer nor human can fit the whole game tree into the memory. Due to the implicit nature of the gaming in general the borderline between to sides is somehow blurred. Next two paragraphs try to draw it as explicitly as possible.

First of all, what makes games really interesting for humans? It is the possibility of both positive and negative outcomes. Since only positive outcomes are desirable, a plan for achieving them (or one of them) has to be constructed. The interesting part lies in discovering properties of the game and/or game states that would assist in creating a successful plan. Why AI touched games and incorporated games as one of the specific research areas? Because of the idea, that AI-system could act in a human-like way, solve such issues as discovering state properties, assessing game states according to those properties, and even become superior in that. So far most of the benefit AI received from direct transfer of human game-playing experience. Therefore issue of automated discovery of state properties is still to be solved. In any case though, if AI systems are really capable of playing games successfully, it would imply they are mature enough to handle no less complex problems with guaranteed success. Using this notion AI was presented a series of tasks, through out the history. One of the recent gains in this field is the victory of Deep Blue over the world chess champion in 1997.

The second point of interest is somehow summoned around “computers vs. humans” confrontation, like in G.Kasparov vs Deep Blue match. Agitation about the confrontation in its turn is based on the fact that humans are especially skilled in pattern recognition, comparison, and generalization, while poor in calculations (both speed and accuracy) and memorizing. It appears that the weaknesses of the human brain are exactly the strong sides of silicon computers. Despite modern computers seem to have gained enough speed and memory to outcast the humans from the field, the humans show quite good results and therefore keep the excitement growing. Given this fact, games become a really special tool in the AI setting for assessment and comparison of human and computational intelligences.

It is not by mistake that intelligence is divided into two parts, or even two different kinds of intelligence. According to [26] they are not going to become the same under any scenario. Though there are ways to imitate human brain activity in a direct way (neural networks for instance), this is outside the scope of this work and will be omitted.

Yet another reason to talk about games and their the role in AI research, is absence of simplification. Even now simplified domains are employed by AI researchers for reasoning about domain properties, provability of certain theorems etc. By their nature games are usually complex with a high branching factor of the corresponding game trees[1]. It is possible to reduce some of the branches due to game board symmetry [3] or by considering some special stages of the game, like "End-Game" [27]. However, solutions to these simplified cases do not produce an overall winning strategy for the whole game, neither to they simplify the rules of a particular game. In it's turn, this means that the complexity of a game is not reduced by focusing on a special case of the game.

As it became clear that human and computational intelligences are two poles of the same globe, let us think of the next challenge for AI game-systems. But to make the idea more clear the strong sides of computational intelligence are highlighted and briefly discussed.

Brute-force search As it was already mentioned, computers are specially good at quick and accurate calculations, which allow them to compute some numerical value and assign it to a particular state in the game. Taking high speed of such calculations into consideration evaluation of big amounts of game states is performed relatively easy and fast. As reported by [13] the Deep Blue program was evaluating about 200 million states a second during the match in 1997, while the world champion was evaluating only two. Such computational effort is relatively similar to analyzing each possible turn at least 12 ply[2] or 6 moves ahead. Today "brute-force search" became a usual tool for an AI researcher.

Large memory Manufacturing of computer memory is becoming cheaper. The direct implication of this fact is higher amount of memory installed into computer systems. Since more memory is available much more information about any particular game could be stored at once, without any overhead. This is illustrated by the Chinook checkers playing program, which contains the "End-Game" database of more than 6 gigabytes, which fit directly into random access memory. This allows Chinook to play ideally (win a winning position and not to lose a drawn one) once the database entry is reached.

Implicit knowledge This point talks about numbers being exact but still unclear to a human mind. Since reasoning on pure numbers is not the nature of human mind, everything that was generated by a computer seems to make sense only to the computer. For instance, an evaluation function based on simulations and fuzzy evaluation with automatic parameter adjustments usually produces a set of numbers associated with one or more

[1] For instance the branching factor of Checkers is approximately 7. For Chess this value is increased to 35, while the Shogi games lifts it up to 80. And for game of Go the branching factor jumps to over 250, which is a specific case and will be discussed a bit more later.

[2] In a standardized chess terminology a "ply" is a turn of one player. One move consists of one turn by each of the two players.

states. Despite that meaning of those numbers are completely unclear for a human reading them, a computer system could easily estimate which move is given preference in a particular state.

Simulation Most of the games regarded by AI research so far are variations of zero-sum games with complete information. What happens if arbitrary games with incomplete information are considered as well? A human player would observe the strategy of an opponent to make corresponding conclusions about the type of the game. Computers in their turn are poor in generalizing experiences, but are very efficient in calculating probabilities. A computer program would initiate some matches of a given game with itself just to calculate the probabilities distributions. For example, a Bridge playing program would pick the card to play by simulating the play of the hand. In other words it would start an "imaginary" game and deal cards to opponents in the "imaginary" game so that they are consistent with the actual game and play starting with every card some considerable number of times. The card which leads to a successful end in most of the cases is picked then.

Once the strong sides of currently available computer systems are identified, we can move on to defining an appropriate challenge for future game-playing AI systems. A real world example of a game, where computer systems are still not maintaining supremacy, is the game of Go. [2] points several issues which make Go an extremely complex game for a computer system. First of all, the branching factor here is 300 in average (compare to 35 in average for Chess). Moreover, the evaluation of a game state could be several times more complex than in any other known game. Finally the game includes lots of inherent knowledge, which despite being a strong side of computer-bases game-playing systems, requires numerous simulations and parameter adjustments. In other words making a kind of off-line pre-evaluation of Go states is not beneficial enough, therefore some alternative approaches have to be searched.

In order to make the task even more generic, we can outline those properties where computer programs are still not sophisticated enough to compete with human players.

On-line execution Most of the current domain-oriented game playing programs benefit from the knowledge of the domain, which is given to them beforehand. This behavior is named "off-line execution", since end-game databases, as well as heuristics and strategies are developed in a training phase and before actual play. On the other hand "on-line execution" stands for development of those mechanisms during the actual course of the game, or alternatively right before it, without any pauses or delays.

Generalization Current game playing programs show lack of generalization. For instance a chess playing program, is not only unable to play checkers, but a different chess flavor would also cause considerable problems. "Generalization" stand for concluding some common properties of various domains, which leads to benefits in playing game from any of those domains. There are a number of applications for such "generalization", for instance, one might conclude that playing chess on two usual boards simultaneously, is quite similar to playing two separate games in turn, or that

"Towers of Hanoi" puzzle is solvable in general case, since a sub-problem with 3 pieces is solvable as well.

Pattern recognition There are a number of games, for which pattern recognition would be quite useful. For instance, it is widely known, that a "stable formation" in Go game is a kind of visual pattern, which once recognized could be constructed relatively easy. One should also note, that "pattern recognition" here stands not only for the visual patterns (though these are probably the most complex to recognize), but also for any pattern that may occur in a winning combinations of moves (i.e. like in the "Towers of Hanoi" game), partially successful state or opponents strategy.

Semantics understanding Some of the games could be played by computer without actual understanding of the game's plot. Recent studies [14] show that even crosswords game could be successfully played by computer without deep understanding of words meaning. However, "understanding semantics" of some game descriptions is crucial to successful play if we are talking about any arbitrary game. For instance, understanding that any board is a two-argument function and vice-versa could simplify the whole reasoning process for many board games.

All of these properties are featured in the General Game Playing initiative launched by Stanford University Logic Group. The overall challenge talks about computers being able to play an arbitrary game after looking at the game's rules. Rules of a game are described in a standardized language named Game Description Language or GDL in short. As it was motivated above, GGP is likely to become a logical successor of the game-playing programs challenge, which is getting resolved with the time. The first championship in GGP was held in 2005 and has been repeated every year since then. Games that are presented to participants are practically limited only by restrictions of GDL, which are few. In particular, the current version of GDL does not allow descriptions of games with incomplete information, as well as games with infinite number of game states. Other flavors of games are allowed and present within the games collection at [31], i.e. any combination of the following turn-taking/simultaneous, single-/multi-player, non-/zero sum is a valid GDL game. In authors humble opinion, extending GDL to cover games with incomplete information is a matter of time.

1.1 Motivation

There are two main reasons, which make GGP challenge so specific. Each of them puts computers in a situation quite different from the one commonly used in classical computer gaming. The first reason stands for the absence of a particular strategy known so far. Since the variety of games available is so vast, some of them could be solved by simple planning or brute breadth-first search, others should probably rely on some heuristics, etc. This claim is supported by two facts.

- The current world champion in GGP, was relying on Monte Carlo method application, not on fine-tuned heuristics. Which is a relatively new approach in game theory. Some bits of the Monte Carlo method were used

for playing Go [2], but generally the Monte Carlo method is not adopted, as to the authors knowledge.

- Moreover the world champion of 2006, FLUXPlayer, from TU Dresden appeared to be more efficient in playing general games than parallel cluster. It should be noted at this point that FLUXPlayer is a single-threaded program written in Prolog. This drops some shadow on supremacy of computers like Deep Blue, which is a parallel computer with 32 IBM SP-2 processors on board [13], in the GGP setting.

The second reason concerns the "off-line execution". According to the rules of GGP, the computer has no prior information about the game it is going to play. There is some time for pre-processing game rules. However, this time is usually limited up to one hour, which is magnitude less than what is needed for constructing a solid End-Game database, or simulating a single round of something more complex than Tictactoe game. For instance, the End-Game database used by Chinook, was taken from Colossus, another checkers playing program. It took more than a year for the Colossus developer to construct the database, and then over six months for Chinook to verify it [28]. Of course, such time odds are not given to GGP-programs. Due to this reason completely different demands are imposed on the preprocessing stage.

One can speculate, that GGP brings computers in more human-like setting. Speaking about the odds, those were given to computers through out the history due to their nature (lack of generalization and recognition). Within the GGP setting any odds given to the computers before are considerably undermined, so the both sides of intelligence are brought closer together.

The purpose of the current paper is to study the construction of End-Game database in GGP setting and prove its correctness w.r.t. the obtained results. One must note that at this very point some ambiguity is already introduced. The term of End-Game is somewhat different in classical gaming programs, which treat an entry of the End-Game database (EGDB) as complete description of the state. While what is meant by EGDB in GGP is more similar to "pattern database" used by Culberson and Schaeffer [5]. This issue will be brought up and explained in more detail later in the paper. The paper is build around the actual implementation of the EGDB construction mechanism developed for FLUXPlayer.

1.2 Conventions

Current work combines several different fields, each with a separate notation and common conventions. Following all of them at a time would make the paper opaque and complex for understanding. Thus, for the sake of readability and ease of comprehension some general conventions are established and maintained through out the paper.

The first convention rule relates to the formulas. Unless explicit convention is given any upper case Latin letters, like X, Y or Z, stand for variables. The special variable S usually belongs to one of the sorts: *situation* or *state*. Consequently vector sign above variables, like $\vec{X}$, $\vec{Y}$ and $\vec{Z}$, stands for the array/vector of variables that might belong to same sort as well as to different ones.

Greek letters, both upper and lower case ones, are used to denote meta-formulas or formula sets.

The true type font, like `match_EGDB`, indicates inclusions of program source code.

Due to the fact, that single-player games are just a special case of planning domain, the words "player" and "actor" are used interchangeably.

1.3 Structure

The rest of the paper is structured as follows. Section 2 provides basic theoretical background needed for understanding of the GDL and goal regression. In particular it touches the frame problem as central issue for any action calculus and then two popular flavors of action calculus, i.e. the Situation Calculus and the Fluent Calculus. At the end of the section some basics of GDL along with $Datalog^{\neg}$ language (as basis for GDL) are introduced.

Section 3 highlights previous research, which contributed to the current work. There are two separate parts. The first one talks about the goal regression, which was introduced to the Situation Calculus as tool for systematic, backward-reasoning. The second one is addressing pattern databases as successful novel approach to in the field of gaming, which is based on techniques similar to the ones used for goal regression.

Section 4 actually provides the main contribution of the current work, by binding all the prerequisites and related work together and proving that resulting regression technique is sound and complete. At the end of the section an implementation-related passage is presented, that focuses on structure of resulting database.

Section 5, as stated in the title of the section, discusses possible further development of the present work. Four main directions are identified: model construction simplification and state invariant identification for impoving the regression algorithm along with disjoint pattern databases, and decision trees as two improvements for the EGDB access time.

The whole paper is concluded by small Section 6 summing up all the work and resulting benefits.

All of the examples given in Section 4 are based on two domains: the Tictactoe game and the Maze game. Tictactoe game is what some might know as "Xs and Os": two players (one playing with "x" and the other with "o"), a 3x3 cell game field and the objective is to form a 3-element line of appropriate marks. The Maze game in its turn is a single-player game, where an actor (a robot) located in a maze (a cycle of rooms) can move through the maze clockwise and also pick up whatever it spots in the room or drop whatever it holds in manipulators. The objective of the Maze game is to relocate a piece of gold from one definite position to another predefined position.

Due to novelty of the approach there is neither dedicated field nor corresponding publications which are directly related to regression in arbitrary multi-agent environment. Thus the work mainly consisted of discovering relevant pieces of information and binding them in a coherent fashion. That is why the paper also contains plenty of introductory material, which is later referred to.

2 Preliminaries

This paper is extensively using action calculus so some prior introduction into the the subject seems appropriate. Generally speaking, any action calculus is made to describe dynamic domains. The word "dynamic" comes from the changes introduced to domain (also referred to as "domain world" or "world"). The changes are expected to be outcomes of the actions performed by some predefined actor. A typical description of a domain in an action calculus includes description of the state of the world and actions available in the world along with their outcomes. The actor has to plan/perform actions in such a sequence that world changes to fit her/his/its suppositions about the ideal world or goal state.

Why action calculus is so widely used? There are two major reasons for this:

1. As it will be shown later in the paper, GDL is based on a flavor of action calculus called the Situation Calculus.

2. Implementation of the technique presented in the paper was performed using realization of the Fluent Calculus, FLUX, which in turn is an instance of action calculus.

In order to move from generic action calculus towards its concrete instances, let us first address one more issue, the frame problem, a fundamental problem of any action calculus being developed. It was first formulated by McCarthy and Hayes in 1969 [21], they also proposed a logic formalism named "Situation Calculus" to solve the problem. Later on Pednault revised the Situation Calculus as solution to the frame problem.

2.1 Frame problem

As it was noted above the frame problem seems to give birth to the Situation Calculus, which is deeply rooted in GDL. Also the frame problem has impact on regression, so in order to have complete understanding of the topic, some comprehension of the frame problem is required as well. The frame problem is a problem of formulating dynamic domains by some logical syntax without explicit specification of those conditions that are not affected by an action. Let us take a more precise look at this definition.

An action in a dynamic domain is regarded as such only in case there are some conditions changed by it. The nature of those changes could be either positive or negative. Disregarding of the change type, it has to be specified in a logical syntax. Rules that actually specify the change produced by an action are called *effect axioms*.

In their turn, positive changes are those that make some condition hold after the action is performed. Let $a(\vec{X})$ denote some action and r stand for some condition, then the corresponding *positive effect axiom* would look like:

$$\pi_a(\vec{X}, S) \wedge \epsilon_R^+(\vec{X}, \vec{Y}, S) \supset r(\vec{Y}, do(a(\vec{X}), S)).$$

Here S stands for some state and $do(a(\vec{X}), S)$ for some successor state once the action $a(\vec{X})$ was performed in state S. The meta-formula $\pi_a(\vec{X}, S)$ holds all the prerequisites that must be satisfied so that the execution of action $a(\vec{X})$

is possible. Prerequisites of this type are commonly referred to as *action precondition axioms*. An example of an action precondition axiom for a domain with one actor and a table could sound like: "in order to put anything on a table an actor must have something in his hands/manipulators". In such case the meta-formula $\pi_a(\vec{X}, S)$ would entail something like $\neg hands\_empty(S)$ or $has\_in\_hands(X, S) \wedge X \neq nothing$, where *hands_empty* predicate symbol encodes the fact that in state S hands/manipulators of an actor are empty (similarly for other predicates). The concrete syntax is not important at this moment. However an important point to note is that these formulas do not depend on condition r, but only on action a itself[3]. Condition r of the state, is also referred to as a *property of the state* or a *fluent*, due to the nature of the condition that changes in state over time. Meta-formula $\epsilon_r^+(\vec{X}, \vec{Y}, S)$ holds state prerequisites under which the action $a(\vec{X})$ is performed. State prerequisites are commonly referred as *fluent preconditions*. Logical notion for fluent preconditions could be expressed as follows: if action $a(\vec{X})$ is performed under $\epsilon_r^+(\vec{X}, \vec{Y}, S)$, then fluent r becomes true for $\vec{Y}$ in the successor state $do(a(\vec{X}), S)$.

If some fluent becomes true after certain action being performed, it could logically become false after the execution of some contrary action. For instance, if putting some object on the table causes an object actually to appear on the table, then picking that object from the table would cease the fluent, which describes presence of the object on the table. Moreover, it might be the same action that, once performed causes some fluents to become true and others to become false. In case with the table-actor domain discussed above, one could imagine that picking an object ceases the fluent which describes presence of an object on the table and simultaneously brings up the fluent which describes presence of an object in the hand/manipulator of an actor. Therefore, there should exist a separate group of *negative effect axioms*, which would represent the ceasing of fluents after execution of an action. The meta-syntax of negative effect axioms is similar to the one for positive effect axioms:

$$\pi_a(\vec{X}, S) \wedge \epsilon_r^-(\vec{X}, \vec{Y}, S) \supset \neg r(\vec{Y}, do(a(\vec{X}), S)).$$

Again, $\pi_a(\vec{X}, S)$ stands for the action precondition axioms, $\epsilon_r^-(\vec{X}, \vec{Y}, S)$ stands for the fluent preconditions. The overall meaning of the formula entails: "if action $a(\vec{X})$ is performed under fluent precondition $\epsilon_r^-(\vec{X}, \vec{Y}, S)$ then fluent r becomes false".

But if the concerned domain is a bit more complex, for instance, if there are several objects on a table and an actor can pick up and hold just one at a time. What would happen to the other object? It is exactly the case when there are some conditions (fluents) not affected by an action. The problem of defining only the relevant actions, without explicit specification of all action/fluent effect axioms is actually the core of the frame problem.

The initial solution [21] included a separate axiom for each action/fluent combination in both positive and negative forms. A *positive frame axiom* has form

$$\pi_a(\vec{X}, S) \wedge \phi_r^+(\vec{X}, \vec{Y}, S) \wedge r(\vec{Y}, S) \supset r(\vec{Y}, do(a(\vec{X}), S)).$$

[3]The actual formal definition of the action precondition axiom which is used for the paper is given later by formula (4)

In case of table domain with two objects, discussed above, a positive frame axiom for an object that remains untouched could entail following formula:

Example 2.1 (Positive frame axiom for table domain)

$$hands\_empty(S) \wedge on\_table(X,S) \wedge X \neq Y \wedge on\_table(Y,X) \supset on\_table(Y, do(pickup(X), S))$$

Here $hands\_empty(S) \wedge on\_table(X,S)$ is an action precondition axiom, $X \neq Y$ is the frame axiom prerequisite, while $on\_table(Y,S)$ and $on\_table(Y, do(pickup(X), S))$ is the same fluent, which remains true after action $pickup(X)$ is executed.

Similarly there might be fluents which are negative (not true) before and after the action execution. Therefore a *negative frame axiom* is needed to describe them.

$$\pi_a(\vec{X}, S) \wedge \phi_r^-(\vec{X}, \vec{Y}, S) \wedge \neg r(\vec{Y}, S) \supset \neg r(\vec{Y}, do(a(\vec{X}), S)).$$

To give example for negative frame axiom, suppose that one of the objects is painted white, while the other is not, and there is a fluent $white(X,S)$, which is true in case of one object and false in case of the other. Example 2.2 shows a possible negative frame axiom for fluent $\neg white(Y,S)$ and action $pickup(X)$:

Example 2.2 (Negative frame axiom for table domain)

$$hands\_empty(S) \wedge on\_table(X,S) \wedge \neg white(Y,S) \supset \neg white(Y, do(pickup(X), S))$$

The issue about the solution given by McCarthy and Hayes was that the frame axioms where expected to be given explicitly along with effect axioms in a domain description. The proposal by Pednault [23] first touched the issue of automatic construction of frame axioms via syntactical transformations. The idea employs the *completeness assumption* for fluent preconditions. According to the completeness assumption, the fluent precondition $\epsilon_r^+(\vec{X}, \vec{Y}, S)$ specifies all the conditions under which action a, if performed, will lead to the truth of r for $\vec{Y}$ in a's successor state. Similarly $\epsilon_r^-(\vec{X}, \vec{Y}, S)$ specifies all the conditions under which action a, if performed, will lead to the falsity of r for $\vec{Y}$ in a's successor state.

By the completeness assumption, one can reason as follows: Suppose that precondition $\pi_a(\vec{X}, S)$, for some action a holds, as well as $r(\vec{Y}, S)$ and $\neg r(\vec{Y}, do(a(\vec{X}), S))$ do. If that is the case, r changed its value from true in state S to false in successor state $do(a(\vec{X}), S)$ due to action a. By the completeness assumption, the only way r could become false is when $\epsilon_r^-(\vec{X}, \vec{Y}, S)$ were true. This reasoning could be expressed as the following meta-formula:

$$\pi_a(\vec{X}, S) \wedge r(\vec{Y}, S) \wedge \neg r(\vec{Y}, do(a(\vec{X}), S)) \supset \epsilon_r^-(\vec{X}, \vec{Y}, S)$$

Which could be rewritten to:

$$\pi_a(\vec{X}, S) \wedge r(\vec{Y}, S) \wedge \neg \epsilon_r^-(\vec{X}, \vec{Y}, S) \supset r(\vec{Y}, do(a(\vec{X}), S))$$

Conversely, similar transformation could be applied to positive effect axioms to produce the following formula:

$$\pi_a(\vec{X}, S) \wedge \neg r(\vec{Y}, S) \wedge \neg \epsilon_r^+(\vec{X}, \vec{Y}, S) \supset \neg r(\vec{Y}, do(a(\vec{X}), S))$$

Concluding what is stated above, there is a systematic way to obtain the frame axioms from the effect axioms, given that completeness assumption holds. In order to illustrate the construction of the frame axioms, let us extend the table domain by action $paint\_white(X)$, which causes an object in the hand /manipulator become of white color. Evidently, for the action to be possible there must be an object in the hand/manipulator. Here is the positive effect axiom for *white* fluent and action *paint_white*.

Example 2.3 (Positive effect axiom for $white/paint\_white$**)**

$$has\_in\_hands(X, S) \wedge X \neq nothing \wedge X = Y \supset white(Y, do(paint\_white(X), S))$$

In this case the completeness assumption entails that the only precondition for an object Y to become white after painting X is that Y must be equal to X, which in its turn must be an object in hand/manipulator of an actor. Once this assumption is adopted, the negative frame axiom for combination $white/paint\_white$ is:

Example 2.4 (Negative frame axiom for $white/paint\_white$**)**

$$has\_in\_hands(X, S) \wedge X \neq nothing \wedge \neg white(Y, S) \wedge X \neq Y \supset \neg white(Y, do(paint\_white(X), S))$$

As common sense hints all the combination of fluents and actions must be processed this way, which definitely includes some void combinations. In a void combination fluent is not affected by action in any case. Consider a negative frame axiom given in Example 2.2. To construct it in the automatic way one has to look at the positive effect axiom of following form:

Example 2.5 (Positive effect axiom for $white/pickup$**)**

$$hands\_empty(S) \wedge on\_table(X, S) \wedge false \supset white(Y, do(pickup(X), S))$$

The automatically constructed equivalent of the formula given in Example 2.2, would look like:

Example 2.6 (Negative frame axiom for $white/pickup$**)**

$$hands\_empty(S) \wedge on\_table(X, S) \wedge \neg white(Y, S) \supset \neg white(Y, do(pickup(X), S))$$

At this point it becomes clear that such an approach entails an issue - to construct all the frame axioms all fluent/action pairs have to be considered even void ones like *white/pickup*, which amounts to enumerating all the frame axioms directly. The number of axioms obtained this way is equal to

$$2 \times \mathcal{A} \times \mathcal{F} \tag{1}$$

where $\mathcal{A}$ stands for the amount of actions and $\mathcal{F}$ for the amount of fluents in domain. A common problem that arises from (1) is that there might be too many frame axioms to be automatically considered or/and generated. Therefore a different approach has to be taken to treat the frame problem.

2.2 Situation calculus basics

A major refinement to the situation calculus was introduced by Reiter [25]. According to his proposal one could avoid declaring the frame axioms explicitly by making the completeness assumption and introducing a $poss(A, S)$ predicate, which would entail all the possible action precondition axioms for every possible action in the domain. Imagine that there is one more way for making objects white - wrapping them into white cover. However, this time they should be small enough to fit into it. Here is a usual positive effect axiom for such definition:

Example 2.7 (Transformation) *Definition of positive effects of two different actions*

$$has\_in\_hands(X, S) \wedge X \neq nothing \wedge X = Y \supset white(Y, do(paint\_white(X), S))$$
$$has\_in\_hands(X, S) \wedge X \neq nothing \wedge small(X) \wedge X = Y \supset white(Y, do(wrap(X), S))$$

could be rewritten to

$$\{[has\_in\_hands(X, S) \wedge X \neq nothing \wedge X = Y \wedge A = paint\_white(X)] \vee [has\_in\_hands(X, S) \wedge A \neq nothing \wedge small(X) \wedge X = Y \wedge A = wrap(X)]\} \supset white(Y, do(A, S))$$

The later formula could be represented in a shorter way if there was a predicate qualifying that action A is possible in a situation S.

Example 2.8 (Shortened) *Following formula represents the same positive effect axiom as above:*

$$poss(A, S) \wedge [(\exists X).A = paint\_white(X) \wedge X = Y \vee (\exists X).A = wrap(X) \wedge X = Y \wedge small(X)] \supset white(Y, do(A, S)) \tag{2}$$

$$has\_in\_hands(X, S) \wedge x \neq nothing \supset poss(paint\_white(X), S)$$
$$has\_in\_hands(x, S) \wedge X \neq nothing \supset poss(wrap(X), S)$$

If generalized the axiom (2) takes the following form

$$poss(A, S) \wedge [(\exists \vec{X_1}).A = a_1(\vec{X_1}) \wedge \epsilon^+_{r,a_1}(\vec{X_1}, \vec{Y}, S) \vee .. \vee (\exists \vec{X_n}).A = a_n(\vec{X_n}) \wedge \epsilon^+_{r,a_n}(\vec{X_n}, \vec{Y}, S)] \supset r(\vec{Y}, do(A, S)) \tag{3}$$

In case $poss(A, S)$ predicate is introduced in a systematic way, one can refer to the following completeness assumption: "The axiom (3) characterizes all the conditions under which action A leads to Y being true under condition r (being white in this particular case)".

Such a compact representation gives more flexibility in reasoning and describing fluent changes. Suppose $poss(A, S)$ is true along with $\neg white(Y, S)$ and $white(Y, do(A, S))$. Then value of Y changed due to formula

$$(\exists X).A = paint\_white(X) \wedge X = Y \vee (\exists X)A = wrap(X) \wedge X = Y \wedge small(X)$$

being true. Such notion could be formulated as follows:

$$poss(A, S) \wedge \neg white(Y, S) \wedge white(Y, do(A, S))$$
$$\supset (\exists X).A = paint\_white(X) \wedge X = Y \vee (\exists X).A = wrap(X) \wedge X = Y \wedge small(X)$$

Similar formulation is possible in opposite direction - from positive fluent in starting situation to negative fluent in successor situation. In general this approach is similar to the approach with *explanation closure axioms* proposed by Clark [4].

Another concept required before the discussion can proceed is the concept of *simple formula w.r.t. to s*.

Definition 2.9 (Simple formulas) *A formula $\int$ is said to be "simple w.r.t. s" if it mentions only domain predicate symbols, whose fluents do not mention the function symbol do, which do not quantify over variable of sort situation, and which have at most one free variable S of sort situation.*

Simplicity of the formula is required in order to speculate about the nature of underlying predicates. In case there is only one situation variable no transition to other situations could be made within the formula, thus one can conclude that the formula is reasoning entirely on the base of current situation. Moreover, the only transition available from one situation to another is through successor state axioms. This point will be recalled once again while proving soundness and completeness of regression.

The changes made to the Situation Calculus with introduction of $poss(A, S)$ predicate could be outline now.

Action precondition axioms An action precondition axiom is a formula of the following form:

$$(\forall \vec{X}, S)[\Pi_a \supset poss(a(\vec{X}), S)] \tag{4}$$

where a is an n-ary action function and Π_a is a formula that is simple w.r.t. s and whose free variables are among $\vec{X}, S$ (recall that according to notational conventions $\vec{X}$ stands for $X_1, ..., X_n$). In order to gather all the action precondition axioms under same name, let us consider a formula

$$(\exists \vec{X}).A = a_1(\vec{X}) \wedge \Pi_{a_1} \vee ... \vee (\exists \vec{Z}).A = a_n(\vec{Z}) \wedge \Pi_{a_n} \tag{5}$$

The only free variables in the formula above are action A and situation S. If one assigns a name to the formula specified it would act as a generic $poss(A, S)$ predicate for any action available. The name which is assigned by [25] is $\mathcal{D}_{\mathcal{F}}$.

Successor state axioms Based on the same completeness assumption as above, one could rewrite effect axioms to *successor state axioms*:

$$poss(A, S) \supset [r(\vec{Y}, do(A, S)) \equiv \gamma_r^+(\vec{Y}, A, S) \vee r(\vec{Y}, S) \wedge \neg\gamma_r^-(\vec{Y}, A, S)]$$

A plain explanation of the formula could sound like: "if action A is possible in a situation S, then executing A in S would lead to r becoming true in successor situation only in one of the cases. Either it was action A that made particular fluent true; or the fluent r was true in parent situation S and execution of action A did not cease the truth value of the fluent"

The general scheme of successor state axiom could be simplified even further, if one adopts convention, that no fluents come from previous situation, but all of them are re-created based on the current situation and the action applied.

$$(\forall A, S)(\forall \vec{X}).poss(A, S) \supset f(\vec{X}, do(A, S)) \tag{6}$$

where f is a fluent with arity $(n + 1)$ last argument of which is supposed to be of sort *situation* for convenience of notation. [25] also provides a generic name for such formula, Φ_f. Based on the form of the successor situation formula, one can state that Φ_f is also simple w.r.t. S and all the free variables of it are among $A, S, \vec{X}$

There is one more point missing in the discussion about $poss(A, S)$. In particular, in order for $poss(A, S)$ to work correctly *unique name assumption* (UNA) has to be adopted as well. In this particular case UNA relates to actions and situations. *Unique names assumption for actions* situations, that two actions are really identical if and only if they have same name and arguments.

$$a \neq a' \supset a(\vec{x}) \neq a'(\vec{x})$$
$$a(x_1, ..., x_n) = a(y_1, ..., y_n) \supset x_1 = y_1 \; land... \wedge x_n = y_n$$

The rationale behind UNA for actions is that $poss(A, S)$ predicate abstracts over actions. Thus there has to be a way to differentiate between two action either on the basis of arguments or on the basis of action name. Otherwise matching of some formula to action precondition axioms of the formula would fail.

The second part of UNA, *unique names assumption for situations* imposes following condition: two situations are identical if and only if they have the same origin, i.e. the sequence of actions that lead to the situations under concern is the same.

$$s_0 \neq do(A, S)$$
$$do(A, S) = do(A', S') \supset A = A' \wedge S = S'$$

This part of UNA (UNA for situations) is more relevant to tasks of regression. According to UNA for situations any situation is described by sequence of actions that to such a situation. This notion simplifies the regression since any situation actually stores part and therefore could be checked for executability and correctness.

Let us discuss what a situation calculus domain description is according to Reiter's notation. The following statement is taken from [24]:

$\mathcal{L}_{sitcalc}$ is a second order language with equality. It has three disjoint sorts: *action* for actions, *situation* for situations, and a catch-all sort *object* for everything else depending on the domain of application.

This basically breaks the domain description into three major parts:

Static facts are those eternally holding statements about the domain. In particular, for the table domain they could describe the amount of objects available, size of each object, etc.

Rules are used for reasoning about the current situation, conditions (fluents) that hold in a state are one example of those. Rules take just one situation variable and no action variables to reason on the current situation.

Dynamic rules are employed to make a transition from one situation to another. These are subdivided into two parts [18]:

Action Precondition Axioms (APA) This represents all definitions of the $poss(A, S)$ predicate. Action precondition axiom for the table domain may look like:

Example 2.10 (Some APA for the table domain)

$$has\_in\_hands(X, S) \wedge X \neq nothing \supset poss(paint\_white(X), S)$$
$$has\_in\_hands(X, S) \wedge X \neq nothing \supset poss(wrap(X), S)$$

Note also that as in the case with rules, precondition for executability of actions are determined only by the current situation.

Successor State Axioms (SSA) Formulas of this part describe fluents that hold in a successor situation, once an action was performed, i.e. they take exactly one variable of action and situation sorts. This ensures that the truth value of fluents in successor situation $do(A, S)$ is determined only by the current situation S. Successor state axioms for the table domain could look like:

Example 2.11 (A SSA for the table domain)

$$white(X, do(A, S)) \equiv \{A = paint\_white(X)\} \vee \{A = wrap(X) \wedge small(X)\} \vee \{white(X, S)\}$$

Section 2.4 shows how the Situation Calculus domain description is employed in GDL.

2.3 Fluent calculus basics

The first research on fluent calculus were introduced in [11]. However the name itself was first used in [1]. The central issue of fluent calculus is trying to solve the inferential frame problem, i.e. efficiently handle computing the non-effects of the actions. Therefore, dealing with the Fluent Calculus one adopts the following idea, situations are identical to their successor situations except for those fluents changed by an action that lead to successor situation. Evidently there is a tradeoff, between supposed efficiency and generality of the approach.

For instance, in the planning domain like the Maze game, the Fluent Calculus would appear quite beneficial, since only a small part of fluents is changed at a time, while the rest of them are simply preserved from the previous state. Conversely, in a game of Go some situations would cause more then one piece to appear on/disappear from the board. If the number of those pieces is close to the number of pieces present on the board the overhead for computing the modification would increase.

Except theoretical foundations (see below) for representing this point formally some changes to the notation have to be introduced. Firstly, the focus of the method is moved from *situation* to *state*. As it was mentioned above, a situation in the Situation Calculus is more similar to a history. On the other hand, a state in the Fluent Calculus can be viewed as snapshot of all the conditions/fluents in a particular domain at a moment of time. Consequently, there could be two syntactically different situations, which are represented by the same state. Conversely, if states are different their situations are different as well. A function $state(S)$ actually relates a situation S to particular state of the world [33].

The world states are, in their turn, described with the help of associative and commutative $\circ$ operation. The operation (written in the infix notation) joins those fluents that are known to hold in a particular state. Following example describes a state in the table domain, where two objects are on the table, one of them is white and hands/manipulators of an actor are empty.

Example 2.12 (State definition in the Fluent Calculus.)

$$\begin{aligned}(\exists X, Y, Z)state(S) \quad = \quad & on\_table(X) \circ on\_table(Y) \circ white(X) \circ Z \wedge \\ & X \neq Y \wedge \\ & (\forall Z', X')[Z \neq has\_in\_hands(X') \circ Z' \\ & \qquad \vee X' = nothing]\end{aligned}$$

Composition of states is performed via the same $\circ$ operator. For instance, assertion that some fluent a holds in a situation S is formalized as

$$(\exists Z).state(S) = a \circ Z \tag{7}$$

In order to abbreviate such a long notation *holds* predicate symbol is introduced as follows:

$$\begin{aligned}holds(A, Z) &\stackrel{def}{\equiv} (\exists Z').Z = A \circ Z' \\ holds(A, S) &\stackrel{def}{\equiv} holds(A, state(S))\end{aligned} \tag{8}$$

Where the variable Z is of *state* sort, whereas the variable S is of sort *situation*. Note also that depending on whether state supplied to *holds* is ground or not it acts correspondingly as checker or constructor of the state.

Contrary to the Situation Calculus in the Fluent Calculus, two states are different if they are composed of different fluents.

The Definition 2.13 gathers all all the properties stated above under the common name, “foundational axioms” [37].

Definition 2.13 (Foundational axioms Σ_{sstate} of the Fluent Calculus) *Consider fluents x, y, z, f, and g, as well as variables X and Y of sort state. Then all of the following rules form the set of foundational axioms for the Fluent Calculus:*

Associativity, commutativity

$$\begin{aligned}(x \circ y) \circ z &= x(y \circ z)\\ x \circ y &= y \circ x\end{aligned} \tag{9}$$

Empty state axiom

$$\neg holds(f, \emptyset) \tag{10}$$

Irreducibility

$$holds(f, g) \supset f = g \tag{11}$$

Decomposition

$$holds(f, X \circ Y) \supset holds(f, X) \vee holds(f, Y) \tag{12}$$

State equality

$$(\forall f).(holds(f, X) \equiv holds(f, Y)) \supset X = Y \tag{13}$$

Associativity and commutativity axioms hint that the order in which fluents occur in a state does not matter. The empty state axiom says that no fluent a member of the special state $\emptyset$. The decomposition and the irreducibility axioms encode idea, that a state could be broken into pieces, but only until those pieces are same as fluents. In other words, the fluents are the fines decomposition of the state. The last axiom actually states the uniqueness of states composed of the same fluents.

Given the foundational axioms a domain description in the Fluent Calculus could be described as stated in Definition 2.14.

Definition 2.14 (Domain axiomatization in Fluent Calculus) *Consider a fluent calculus signature with action functions. A domain axiomatization (or domain description) in fluent calculus is a finite set of axioms:*

$$\Sigma = \Sigma_{dc} \cup \Sigma_{sua} \cup \Sigma_{aux} \cup \Sigma_{init} \cup \Sigma_{sstate}$$

where:

Σ_{dc} *is the set of domain constraints;*

Σ_{poss} *is the set of precondition axioms, one for each action;*

Σ_{sua} *is the set of state update axioms, one for each action;*

Σ_{aux} *is the set of auxiliary axioms, with no occurrence of states except for fluents, no occurrence of situations, and no occurrence of $poss(A, S)$ predicate;*

Σ_{init} *by definition is $\{state(s_0 = \tau_0)\}$, where τ_0 is a ground state;*

Σ_{sstate} *are foundational axioms of the Fluent Calculus.*

The most interesting element at this point is the set of *state update axioms* (SUA), Σ_{sua}. The general form of a state update axiom is

$$\Delta(S) \supset \Gamma[state(do(A, S)), state(S)] \tag{14}$$

where $\Delta(S)$ entails the conditions on S, or more likely on the state corresponding to situation S, under which Γ defines how the successor state $state(do(A, S))$ is obtained via modifications on the current state $state(S)$. As an example, let us consider the effect of the $pickup(X)$ action.

Example 2.15 (State update axiom for the $pickup(x)$ action.)

$$\begin{aligned} &poss(pickup(X), S) \supset \\ &(state(do(pickup(X), S)) = state(S) + has\_in\_hands(X) - on\_table(X)) \end{aligned} \tag{15}$$

Given in words, this formula states that if action $pickup(X)$ is possible, then executing it in state $state(S)$ would produce a new state. The old state is equal to the new one if appended by $has\_in\_hands(X)$ fluent and subtracted by $on\_table(X)$ fluent. Thus, $on\_table(X)$ is the only negative and $has\_in\_hands(X)$ is the only positive effect of the action $pickup(X)$.

Intuitively one can see that appending is another shorthand for concatenation via $\circ$ operator, but subtracting requires a more involved macro-definition [37].

Definition 2.16 (Minus operator, $-$) *Suppose that f, $f_1, .., f_n$ are all fluents, z, z_1, and z_2 are ground state definition respectively. Then subtraction operation is defined inductively,as follows:*

$$\begin{aligned} &z_1 - \emptyset = z_2 \stackrel{def}{\equiv} z_2 = z_1 \\ &z_1 - f = z_2 \stackrel{def}{\equiv} (z_2 = z_1 \vee z_2 \circ f = z_1) \wedge \neg holds(f, z_2) \\ &z_1 - (f_1 \circ f_2 \circ ... \circ f_n) = z_2 \stackrel{def}{\equiv} \\ &\qquad (\exists z)(z = z_1 - f \wedge z_2 = z - (f_2 \circ ... \circ f_n)) \end{aligned} \tag{16}$$

Important point is covered in second formula. It actually provides case distinction depending whether the fluent f was true in a state z_1 or not. In case it was not, two states z_1 and z_2 are simply equal, otherwise relation between z_2 and z_1 is expressed by formula:

$$z_2 \circ f = z_1 \wedge \neg holds(f, z_2)$$

Along with difference in axiomatization, fluent calculus offers a different way of constructing states. State update axioms were mentioned above, now we will go through them in detail. The formula (14) gives a generic form of state update axiom. In practice, $\Delta(S)$ usually combines a $poss(A, S)$ predicate with some assertions over fluents in current state $state(S)$. While the form of Γ varies depending on action under concern. Three cases are outlined in [33]:

Simple form This case relates to deterministic actions with direct and closed effects. Under closed effects, one should understand that action does not have potentially infinitely many effects. Example 2.15 shows one of the simple state update axioms. Here Γ takes the form of an equation relating the current state to the successor state. Generally for action A and some positive effect f of the action this could be formulated as $state(do(A, S)) = state(S) \circ f$. In other words, new state is obtained from old one by appending all the direct positive effects. If the effect f is negative, then it has to be withdrawn from the new state, so that old state is a combination of new state and negative effects: $state(do(A, S)) = state(S) - f$. Combination of these two formulas leads to generic form of state update axioms for deterministic actions with direct and closed effects (recall the completeness assumption from Section 2.1):

$$\Delta(S) \supset state(do(A, S)) = state(S) + \vartheta^{+} - \vartheta^{-}$$

Here ϑ^{-} and ϑ^{+} stand for finitely many fluents connected by $\circ$ operator. There is a mechanical and systematic way of obtaining formulas of this form from Situation Calculus-like effect axioms. For such a mechanical derivation all the effect axioms are assumed to provide a complete account of the effects relevant to an action [33].

For the particular case of regression within GGP setting simple form of SUA is sufficient.

Disjunctive form If disjunction is introduced into Γ then state update axioms become suitable for specifying nondeterministic actions. In this case each disjunct in Γ would describe a possible effect of an action. Take a look at the following example:

Example 2.17 (SUA for nondeterministic actions.)
$poss(shoot, S) \supset$
$state(do(shoot, S)) \circ has\_arrow = state(S)$
$\vee$
$state(do(shoot, S)) \circ has\_arrow = state(s) \circ hit\_the\_target$

As one may have noticed this is a simple axiomatization of shooting and arrow in some arbitrary domain. According to what is given in SUA, shooting an arrow can either result into hitting the target or not, but causes loosing the arrow in any case. Therefore fluent *hit_the_target* may or may not become true after the action execution.

Disjunctive form could be employed for regression of simultaneous games, however it would relate to one-sided perspective on the game. For instance, if an actor is trying to plan own actions considering the actions of other players as source of nondeterministic outcome. The corresponding bipartile game graph would in this case entail nondeterminicity in the same way as implied by the formula in Example 2.17.

Ramification According to [9] the Ramification Problem deals with handling indirect effects of actions. For an example of indirect effect consider a version of table domain axiomatization, where both fluents $has\_in\_hands(x)$ and *hands_empty* exist. It is logical to conclude that:

$$holds(hands\_empty, S) \supset holds(has\_in\_hands(nothing), S)$$

or even

$$holds(has\_in\_hands(X), S) \wedge X \neq nothing \supset \neg holds(hands\_empty, S)$$

It is clear that picking an object from the table should affect both of the fluents. In particular, if both *hands_empty* and *has_in_hands*(*nothing*) hold in a state, where action *pickup*(X) is performed, which causes $\neg$*has_in_hands*(*nothing*), then $\neg$*hands_empty* must be caused as well. Moreover, further indirect effect could be accounted by successive application of *causal relationships*[4] [32], [34]. Therefore a separate type of state update axioms has to be used. State update axioms that take indirect effects into account are of the form:

$$\Delta(s) \supset Z \circ \vartheta^- = state(S) \circ \vartheta^+ \supset ramify(Z, \vartheta^- \circ \vartheta^+, state(do(A, S))$$

where ϑ^- and ϑ^+ are negative and positive effects respectively, and $ramify(S, E, S_{+1})$ stands for successive application of casual relationships (zero or more) to state *state* and effects $effects$ resulting into state *new_state*.

Ramification case is practically of no use for the purpose of regression. Thus it is mentioned here entirely for the sake of keeping the introduction to the Fluent Calculus consistent.

It was shown [12] that verifying conclusions about action sequences in fluent calculus is computationally more beneficial than in situation calculus. In its turn this computational benefit is dependent on an efficient treatment of equality. Since straightforward addition of equality could harm theorem proving, efficient constraint solving algorithms were developed to satisfy equational theory required by "$\circ$" function, [22].

Finally, it is worth mentioning that the version of the Fluent Calculus described here is commonly known as Special Fluent Calculus, which is also indicated in the name of foundational axioms Σ_{sstate}, additional "s" stands for "special". The Special Fluent Calculus is oriented towards complete information domains, which is exactly the case with GGP. The next chapter provides more information on this matter.

This rounds out the discussion on action calculi and their usage for domain descriptions. The following sections introduce GDL and its formal basis, *Datalog*$^\neg$.

2.4 *Datalog*$^\neg$ and GDL syntax

Datalog is a restricted version of Prolog, i.e. Horn-clause programs without functional symbols, which became popular as part of logical databases [8]. Due to computational benefits presented by *Datalog* in comparison to Prolog, attention to this language spread in other areas as well. This paper mentions *Datalog*, or rather *Datalog* with negation (*Datalog*$^\neg$), because it is used as base for GDL – language for description of general games.

[4]Causal relationships are formalized by expression of the form $condition_1$ `causes` $effect$ `if` $condition_2$, which actually operate on state/effects pairs (S, E), where S is current state and E is collection of all effects (both, direct and indirect) considered so far.

2.4.1 *Datalog*$^\neg$

As it was stated above *Datalog*$^\neg$ is an extension of Horn-clause programs without functional symbols by a negation operator $\neg$. Since negation itself could harm computational efficiency of *Datalog* some conventions were introduced to preserve the benefits to the extent possible.

Before going into that let us describe the syntax of *Datalog*$^\neg$. As any Horn-clause based language, *Datalog*$^\neg$ has *atomic sentences*, which consist of n-ary relation symbols applied to n terms. A *term* in its turn is a variable or an object constant. A *literal* is an atomic sentence or its negation, i.e. $p(a)$, $q(2, X)$ and $\neg r(Y)$ are all literals. A *Datalog*$^\neg$ rule is an implication, where head of the implication is an atomic sentence. The body of the rule is conjunction of zero or more literals. Restrictions for negation usage are discussed later. In general, a rule has the following form:

$$head \Leftarrow literal_1 \wedge ... \wedge literal_n$$

where *head* and $literal_i$ are the head of a rule and a literal in the body of a rule respectively. For safety reasons the following convention on negation is adopted: "if a variable appears in the head or in a negative literal within the body it must appear in a positive literal in the body of the rule."

This restriction worth giving it some more attention. It exists to deal with negation while keeping *Datalog*$^\neg$ in decidable domain. Since *Datalog*$^\neg$ employs "negation as failure" technique to define negation, which can cause computational problems [30], [4], negation in *Datalog*$^\neg$ has to be stratified. A set of rules is said to be *stratified* if starting from any relation constant p one can never reach $\neg p$ by following the rules backwards. The first rule in following example is stratified, the second one is not:

Example 2.18 (Stratified and not stratified rules) *The following two formulas could be stratified:*

$$s(X) \Leftarrow t(X) \wedge r(X)$$
$$r(X) \Leftarrow s(X) \wedge f(X, Y)$$

whereas next three could not:

$$p(Y) \Leftarrow g(Y)$$
$$g(Y) \Leftarrow q(Y)$$
$$g(Y) \Leftarrow \neg p(Y)$$

[20] contains a nice explanation on why stratification is so important. Consider a rule that has a variable in the head (or in a negative literal) which does not occur in a positive literal of the body, as provided by Example 2.19

Example 2.19 (Invalid *Datalog*$^\neg$ rules)

$$p(X, Y) \Leftarrow q(X)$$
$$r \Leftarrow \neg q(X)$$

Suppose that $q(a)$ is true, then $p(a, Y)$ should also be true, but commonly for *Datalog*$^\neg$ the vocabulary is assumed to include infinitely many object constants. Therefore, there are infinitely many outcomes of $q(a)$.

In the second case, the violation of the stratification rule leads to ambiguity in interpretation. One can understand that $\neg q(X)$ is true once there is some particular value of X for which q is false. Alternatively, the second statement could be interpreted as, "r is true, when q is false for its every ground instance."

The constraints on *Datalog* remove this ambiguity by asserting that every variable in head or negative literal has to occur in positive literal. This way first all the positive literals are evaluated to make all the negative literals ground before evaluating them with negation as failure.

Finally, *Datalog* also includes two more assumptions: unique name assumption (UNA) as discussed in Section 2.2 and *domain closure assumption* (DCA), which says that the only objects in the world are those named by ground terms.

In order to sum up the current section and move on to the discussion of the GDL itself, one should keep in mind that *Datalog*$^\neg$ is a syntactical base. It was chosen thanks to considerable expressive power and relatively low computational overhead.

2.4.2 GDL

A *game description language* (GDL) is the binding point for all the previous sections in this chapter. GDL is a language for describing finite-state machines based on *Datalog*$^\neg$ but contains some modifications that deal with equality, inclusion of function constants and finiteness of recursion, due to presence of function constants. Each of the modification is discussed in turn.

Due to UNA *Datalog* treats two terms equally if those are exactly the same term. *Datalog* also allows testing for equality ($=$) as well as for inequality ($\neq$). In its turn GDL accepts only inequality testing via `distinct/2` predicate. The idea behind such choice is that equality could be achieved when instead of writing $X = Y$, X is always used in place of Y or vice versa. On the other hand encoding a relation for all unequal terms in domain without a dedicated predicate could be troublesome.

For the sake of readability and compactness of game descriptions GDL also allows disjunction in the bodies of the rules. A rule of the form

$$p \Leftarrow a \wedge (b \vee c)$$

is much more compact then its DNF representation. Surely the same effect could be achieved by introducing a new rule bc:

$$\begin{aligned} p &\Leftarrow a \wedge bc \\ bc &\Leftarrow b \\ bc &\Leftarrow c \end{aligned}$$

In any case, introduction of disjunction allows writing more compact game descriptions and saves from obligation to write auxiliary clauses every time.

The next modification to consider is recursion in combination with functional constants. In general *Datalog*$^\neg$ queries with recursion and functional symbols are formally undecidable. It is true that avoiding any of these features would bring queries back to decidable side. However such solution would make some

game descriptions excessively complex. [20] offers a minor syntactic restriction on combination of recursion and function symbols, which is sufficient to preserve finiteness and decidability in all cases. The formulation of the restriction is based on dependency graph abstraction.

Definition 2.20 (Dependency graph) *If Δ is a set of rules then the dependency graph for Δ is constructed as follows:*

- *predicate symbols are used for vertices;*
- *an edge from a vertex r_1 to vertex r_2 is introduced if there is a rule with r_1 in the head and r_2 in the body.*

Now, everything is set to provide the formal definition of the recursion rule.

Definition 2.21 (Recursion Rule) *Let Δ be a set of rules and let G be the dependency graph of Δ. Suppose that Δ contains the rule*

$$p(t_1, ..., t_n) \Leftarrow b_1 \wedge ... \wedge q(v_1, ..., v_k) \wedge ... \wedge b_m$$

where q appears in a cycle with p in G. Then $\forall j \in \{1, ..., k\}$, either v_j is ground, $v_j \in \{t_1, ..., t_n\}$ or $\exists i_{\in\{1,...,m\}}.b_i = r(..., v_j, ...)$, where r does not appear in a cycle with p.

This definition ensures that terms can not grow to unbounded size via means of recursion, since they may be passed along a recursive cycle without functional symbols, and those terms introduced inside recursive cycles have to be grounded by a finite set of base relation sentences.

To conclude with the *Datalog*$^\neg$ part of the GDL let us summarize the modifications introduced.

- The vocabulary is extended by functional constants with associated arity, e.g. cell, room, gold.
- The terms are extended by functional symbols with arity n applied to n terms, e.g. $a(X, 1, b(2, Y))$.
- The body of a rule can consist of literals or disjunction of literals. However, after converting all rules with disjunction in the body into a set of rules without disjunction a general *Datalog* constraint must hold: "if a variable appears in the head or in a negative literal it must appear in a positive literal in the body". Meanwhile, the `distinct/2` predicate is treated as a negative literal.
- The Recursion Rule given in Definition 2.21 must hold as well.

Whereas *Datalog*$^\neg$ forms the syntactical base of GDL, GDL descriptions follow the APA/SSA framework of the Situation Calculus. The rest of this section is dedicated to the discussion on this point.

As GDL is a language for describing finite state machines, it describes the initial state as well as transition function. The way the transition function is presented, resembles the Situation Calculus with its action precondition axioms and successor state axioms. What GDL uses for defining action preconditions ($poss(A, S)$ predicate in Situation Calculus) is called a `legal` predicate of arity

2. It takes a role name (also referred to as "player" or "actor") and an action as argument. If a particular action is possible in the current state, then corresponding `legal` predicate clause evaluates to truth. The question of supplying different states to the `legal` predicate, in order to expand the game tree to higher depth for example, is delegated to GGP-software developers.

In order to decouple moves which are legal in a state from those actually played one more predicate is introduced. `does` or arity 2 denotes the moves which were taken by players in the course of the game.

Successor state axioms are represented by the `next` predicate which follows the pattern described in (6). If some fluent must hold in the successor state, then corresponding clause of the `next` predicate evaluates to truth. No convention is made concerning fluents being transferred automatically to the new state. In other words the new state consists entirely of the fluents, which where introduced by successful evaluation of the `next` predicate.

Whereas `next` describes the successor state, everything that holds in the current state is defined via `true` predicate. Such decomposition imposes some more syntactic restrictions, i.e. `true` and `does` are never expected in the head of a rule, whereas `next` could never be used in the body of a rule.

The similarity between GDL and the Situation Calculus gave rise to the central idea of this paper - application of goal regression for constructing EGDB.

Essential to soundness and completeness proof, there is one more restriction imposed on domains described by GDL. In particular, the games/domains must be well-formed, where property of being well-formed is defined as follows.

Definition 2.22 (Well-formed game) *A game description in GDL is well-formed if it terminates and is both playable and weakly winnable.*

This definition relies on three other definitions provided below.

Definition 2.23 (Weak winnability) *A game description is weakly winnable if and only if, for every role (also referred to as "actor" within the paper), there is a sequence of joint moves of all roles (also referred to as "combined action", see Section 4.1.2) that leads to a terminal state, where that role's goal value is maximal.*

Definition 2.24 (Termination) *A game description in GDL terminates if all infinite sequences of legal moves from the initial state of the game reach a terminal state after a finite number of steps.*

Definition 2.25 (Playability) *A game description in GDL is playable if and only if every role has at least one legal move in every non-terminal state reachable from the initial state.*

As presented in the Section 2.3, one of the changes introduced by the Fluent Calculus is the representation of a state as collection of fluents, not as situation or history [35]. In this sense a game described in GDL really resembles a Fluent Calculus domain description, since GDL also relies on particular fluents being described via `true` (checks whether a fluent holds in current state) and `init` (defines the initial state) statements. Despite that such change could potentially allow descriptions with incomplete information [36], GDL, at least according to the current specification, describes only complete information games.

The discussion of GDL is concluded by some implementation-related points. The predicate symbols `does` and `next` will appear in Section 4 with some minor modifications. In case of the `does` predicate two more arguments are appended: the third argument represents a combined action (see Definition 4.6) performed by all the actors at a given game step and the fourth argument classically represents a state. `next` has only one argument appended representing the successor state. Such changes are forced by the implicit nature of `does` and `next` given in [20]. Both of them are expected to reason about a single current state, whereas the implementation considers several states at a time, for instance, during the search procedure. Presence of combined action as the third argument of `does` is motivated by the same rationale. Moreover, since FLUX (FLuent EXecutor, realization of the Fluent Calculus) is the base-system of choice for implementation of the current work, the `true` predicate symbol is systematically replaced by the `holds` predicate, native to the Fluent Calculus.

3 Related work

The work presented in this paper forms a relatively new research area. However, there are several existing fields which contribute to actual research. Two of them are discussed here as most prominent: "regression" as defined by R.Reiter in [25], and "pattern databases" introduced by J.Schaeffer [5].

3.1 Regression in situation calculus

In [25] regression is introduced as a tool for systematic, backward-reasoning style proof theory. Before similar idea also appeared in [38]. The overall objective is to find some situation S that satisfies

$$\mathcal{F} \models (\exists S) g(S) \wedge ex(S) \tag{17}$$

Where F is a suitable axiomatization of a dynamic domain, $g(S)$ is a certain goal and $ex(S)$ is a predicate, which ensures *executability* or reachability of situation S. Presence of $ex(S)$ may seem excessive, in order to motivate its presence in the formula, let us discuss how regression is presented in the Situation Calculus. For a table domain introduced above some axiomatization may entail

Example 3.1 (A ghost situation in situation calculus)
$$\mathcal{F} \models white(A, do(paint\_white(A), s_0))$$

By retracting situation $do(paint\_white(A), s_0)$ one obtains a plan to make A of white color. But this plan has to be executable, i.e. the action precondition for $paint\_white(A)$ has to follow from $\mathcal{F}$. Since in initial situation all the objects reside on the table and hands/manipulators are empty this situation is not reachable by any executable sequence of actions. Such situations are called *ghost situations* in [25].

In order to exclude such situations from possible answer set (possible plans for achieving goal) $ex(S)$ predicate is introduced.

Definition 3.2 (Predicate that ensures executability of situation)

$$ex(S) \equiv S = s_0 \vee (\exists A, S') S = do(A, S') \wedge poss(A, S') \wedge ex(S'). \tag{18}$$

The recursive definition of $ex(S)$ states that $poss(A, S)$ has to be satisfied at every stage of a plan until initial situation is reached. Consequently satisfaction of $ex(S)$ for some situation S proves its executability in general.

At this point it should be mentioned that $ex(S)$ predicate is not affordable for regression in GGP. The rationale for this is given later.

Once the overall view of goal regression was given, let us move to particular definitions used for goal regression in the Situation Calculus.

3.1.1 Regression operator

Regression operator is a proof-theoretic tool used on one particular formula to reduce the depth of nested *do* function symbols in the fluents of the formula.

Let $\mathcal{L}$ be a first-order language with two sorts *action* and *situation* used for axiomatization of dynamic domains in the Situation Calculus [25], and $\Theta \subseteq \mathcal{L}$ contain one successor state axiom for each distinct fluent of that language, then the *regression operator* could be introduces as given in Definition 3.3

Definition 3.3 (Regression operator) *The regression operator, $\mathcal{R}_\Theta$, is defined by case distinction as follows:*

- *If A is not a fluent atom (including equality atoms and atoms with predicate symbols poss and ex;*
  If A is a fluent atom, where situation argument is represented by a variable or initial situation constant s_0:

$$\mathcal{R}_\Theta[A] = A$$

- *If F is a fluent whose successor state axiom in Θ is*

$$(\forall A, S)(\forall X_1, ..., X_n)poss(A, S) \supset F(X_1, ..., X_n, do(A, S)) \equiv \Phi_F$$

  then

$$\mathcal{R}_\Theta[F(t_1, ..., t_n, do(\alpha, \sigma))] = \Phi_F|^{X_1,...,X_n,A,S}_{t_1,...,t_n,\alpha,\sigma}$$

- *If W is a formula:*

$$\begin{aligned}\mathcal{R}_\Theta[\neg W] &= \neg\mathcal{R}_\Theta[W],\\ \mathcal{R}_\Theta[(\forall V)W] &= (\forall V)\mathcal{R}_\Theta[W],\\ \mathcal{R}_\Theta[(\exists V)W] &= (\exists V)\mathcal{R}_\Theta[W].\end{aligned}$$

- *Whenever W_1 and W_2 are formulas:*

$$\begin{aligned}\mathcal{R}_\Theta[W_1 \wedge W_2] &= \mathcal{R}_\Theta[W_1] \wedge \mathcal{R}_\Theta[W_2],\\ \mathcal{R}_\Theta[W_1 \vee W_2] &= \mathcal{R}_\Theta[W_1] \vee \mathcal{R}_\Theta[W_2],\\ \mathcal{R}_\Theta[W_1 \supset W_2] &= \mathcal{R}_\Theta[W_1] \supset \mathcal{R}_\Theta[W_2],\\ \mathcal{R}_\Theta[W_1 \equiv W_2] &= \mathcal{R}_\Theta[W_1] \equiv \mathcal{R}_\Theta[W_2]\end{aligned}$$

$\mathcal{R}_\Theta[g]$ simply substitutes suitable instances of Φ_F in F's successor state axiom for each fluent of form $F(t_1, ..., t_n, do(\alpha, \sigma))$ that occurs in g.

3.1.2 Regression stage

Performing regression on a formula once does not usually produce interesting results, i.e. initial situation is seldom reached, and therefore the constructed plan is not of much use. On the other hand there is no prior notion on how many actions an actor has to perform to reach the goal. In many cases several successful plans exist, each with different amount of actions.

In order to deal with iterative regression of a formula, *regression stage* is introduced inductively as follows.

Definition 3.4 (Regression stage) *Let $g(S) \in \mathcal{L}$ have single situation variable. Suppose $\mathcal{F} \subset \mathcal{L}$ contains a successor situation axiom for each fluent of $\mathcal{L}$ and an action precondition axiom for each action function of $\mathcal{L}$:*

$$\Gamma_0(S) = g(S)$$
$$\Gamma_i(S) = (\exists A_i)\mathcal{R}_F[\Gamma_{i-1}(do(A_i, S))] \wedge \mathcal{D}_{\mathcal{F}}(A_i, S) i = 1, 2, ...$$

Where $D_{\mathcal{F}}(A_i, S)$ is the formula defined in Section 2.2.

The principle of goal regression as defined by R.Reiter [25] is formulated in the following theorem.

Theorem 3.5 (Regression Theorem) *Suppose $\mathcal{F} \subseteq \mathcal{L}$ contains the executability axiom (18), unique names axioms for situations, a successor state axiom for each fluent of $\mathcal{L}$. Suppose further that the remaining axioms of $\mathcal{F}$ are s-universal and mention only domain predicate symbols. Finally, suppose that $g(S) \in \mathcal{L}$ is simple w.r.t. S, and that the situation variable S is the only free variable of $g(S)$. Then each of the following holds:*

1. $\mathcal{F} \models (\exists S) g(S) \wedge ex(S)$

   if and only if for some n

   $$\mathcal{F}^- \models \Gamma_0(s_0) \vee ... \vee \Gamma_n(s_0)$$

   where $\mathcal{F} = \mathcal{F}^- \cup ex(S) \wedge \mathcal{F}^- \cap ex(S) = \emptyset$

2. *For every n, $\Gamma_0(s_0) \wedge ... \wedge \Gamma_n(s_0)$ mentions only domain predicate symbols.*

3. *For every n, s_0 is the only situation term mentioned by the fluent atoms of $\Gamma_0(s_0) \wedge ... \wedge \Gamma_n(s_0)$*

Let us dig into the definition to get the meaning of the theorem in a clear form. If one would like to get an iterative definition for Γ_n it would look like

$$\Gamma_n(s_0) = (\exists A_n)\mathcal{R}_{\mathcal{F}}[...(\exists A_1)\mathcal{R}_{\mathcal{F}}[g(do(A_n, ..., do(A_1, s_0)...))]...] \wedge$$
$$\mathcal{D}_{\mathcal{F}}(A_1, s_0)$$
$$\wedge ... \wedge$$
$$\mathcal{D}_{\mathcal{F}}(A_n, do(A_{n-1}, ...do(A_1, s_0)...))$$

If one assumes that plan $\vec{A_n}$ is a shorthand for $A_1, ...A_n$, $do(\vec{A_n}, s_0)$ is a corresponding syntactic shorthand for $do(A_n, ..., do(A_1, s_0)...)$ or execution of the plan, and $\mathcal{R}^n_{\mathcal{F}}$ represents successive application of regression operator n times, then the above formula could be represented in a compact form:

$$\Gamma_n(s_0) = (\exists \vec{A_n})\mathcal{R}^n_{\mathcal{F}}[g(do(\vec{A_n}, s_0))] \wedge$$
$$\mathcal{D}_{\mathcal{F}}(A_1, s_0)$$
$$\wedge ... \wedge$$
$$\mathcal{D}_{\mathcal{F}}(A_n, do(\vec{A}_{n-1}, s_0))$$

What the first part of this formula suggests is that the goal formula $g(S)$ applied to a situation $do(\vec{A_n}, s_0)$, which was synthesized by execution of plan $\vec{A_n}$ starting with the initial situation s_0, must be regressable to initial situation s_0 in n steps. Moreover every action A_i within the plan should be possible

in corresponding situation $do(A_{i-1}, S)$, which is ensured by a sequence of $\mathcal{D}_{\mathcal{F}}$ predicates in the second part of the formula.

Now let us return to the first statement of the Theorem 3.5. The sequence $\Gamma_0(s_0) \vee ... \vee \Gamma_n(s_0)$ stands for incremental application of regression stages to the initial situation. In other words, one increments the possible size of acceptable plan for reaching the goal from the initial situation. If the goal is reachable in principle, at some point the regression stage should evaluate to truth and make the whole formula valid. As it was said above, each regression stage includes the validation of the action by means of $\mathcal{D}_{\mathcal{F}}$. Since each action in the plan gets validated the need for $ex(S)$ disappears, so one can use $\mathcal{F}^-$ instead of $\mathcal{F}$.

Since formula $\Gamma_0(s_0) \vee ... \vee \Gamma_n(s_0)$ contains no action precondition axioms or successor state axioms, one might suppose that their role is already over, through contribution to regression. This notion is false. It turns out that not every plan synthesized this way will comply with the actual initial situation given. Here is an example from [25].

Example 3.6 (Incomplete goal regression)
Suppose $\mathcal{F}$ contains unique names axiom for situations together with the following two successor state axioms and single action precondition axiom:

$$poss(A, S) \supset p(do(A, S)) \equiv d(S)$$
$$poss(A, S) \supset q(do(A, S)) \equiv e(S)$$
$$true \supset poss(A, S) \equiv \mathcal{D}_{\mathcal{F}}$$

Suppose $\mathcal{F}$ also contains an initial situation axiom $e(s_0)$, and a general axiom $q(S) \supset p(S)$. Consider the goal $(\exists S)p(S) \wedge ex(S)$. Then

$$\Gamma_0(s_0) \vee \Gamma_1(s_0) = p(s_0) \vee ((\exists A_1)\mathcal{R}_{\mathcal{F}}[p(do(A_1, s_0))] \wedge \mathcal{D}_{\mathcal{F}}(A, s_0)) =$$
$$p(s_0) \vee (D(s_0) \wedge true) =$$
$$p(s_0) \vee D(s_0)$$

It is trivial to notice that $\mathcal{F} \models p(s_0) \vee d(s_0)$ holds, whereas:
$e(s_0), q(S) \supset p(S) \not\models p(s_0) \vee d(s_0)$.

So it appears that successor state axioms along with action precondition axioms could not always be discarded. An intuitive question would be: "Under what conditions could these two axiom sets be omitted without harming the completeness of goal regression?"

The next section briefly describes the prerequisites for discarding action precondition axioms and successor state axioms without loosing completeness of the goal regression, as provided in [25].

3.1.3 Completeness of Goal Regression

First of all let us provide some definitions used for the goal regression completeness theorem.

Definition 3.7 (s-Admissible sentences) *A sentence is s-admissible if and only if it mentions no situation variable at all, or it is of the form $(\forall S)w(S)$, where S is a situation variable, and $w(S)$ is simple w.r.t. S.*

In plain language, s-admissible sentences are general facts of the axiomatized domain, something that is true in any situation or is not related to situation at all like unique names axioms or $on\_table(A, s_0)$. Some more examples of s-admissible and non-s-admissible sentences follow

Example 3.8 (s-Admissible sentences)

$$(\forall X, S)has\_in\_hands(X, S) \supset \neg on\_table(X, S)$$
$$(\exists X)small(X)$$

Example 3.9 (Non-s-Admissible sentences)

$$(\exists X)small(X) \wedge \neg white(X, S)$$
$$(\forall S)on\_table(X, S) \supset (\forall S')white(X, S')$$

Definition 3.10 (Closure under Regression) *Suppose $\mathcal{F} \subseteq \mathcal{L}$ contains a successor state axiom for each fluent of $\mathcal{L}$. Suppose Ω is a set of s-admissible sentences of $\mathcal{L}$. Then Ω is said to be "closed under regression" w.r.t. $\mathcal{F}$ if and only if $(\forall S)w(S) \in \Omega$,*

$$\Omega \models (\forall S, A)\mathcal{D}_{\mathcal{F}}(A, S) \supset \mathcal{R}_{\mathcal{F}}[w(do(A, S))].$$

Theorem 3.11 (Soundness and completeness of Regression) *Suppose*

$$\mathcal{F} = \{exax\} \cup \mathcal{F}_{ss} \cup \mathcal{F}_{ap} \cup \mathcal{F}_{uns} \cup \mathcal{F}_{\forall s} \subseteq \mathcal{L} \tag{19}$$

where exax is the executability axiom, $\mathcal{F}_{ss}$ is a set of successor state axioms, one for each fluent of $\mathcal{L}$, $\mathcal{F}_{uns}$ is the set of unique names axioms for situations, and $\mathcal{F}_{\forall s}$ is a set of s-admissible sentences that is closed under regression w.r.t. $\mathcal{F}$. Suppose that $g(S) \in \mathcal{L}$ has as its only free variable the situation variable s, and that $g(S)$ is simple w.r.t. s. Then

$$\mathcal{F} \models (\exists S)g(S) \wedge ex(S)$$

if and only if for some n,

$$\mathcal{F}_{uns} \cup \mathcal{F}_{\forall s} \models \Gamma_0(s_0) \vee ... \vee \Gamma_n(s_0)$$

What Theorem 3.11 states in (19) is that a domain axiomatization consists of separate distinct sets. It also ensures that all the domain predicates that appear in the axiomatization (which can not be part of $\mathcal{F}_{ss}$, neither of $\mathcal{F}_{ap}$, nor of $\mathcal{F}_{uns}$) are s-admissible, as defined by $\mathcal{F}_{\forall s}$. The key point here is the requirement for the $\mathcal{F}_{\forall s}$ to be closed under regression. To understand what this requirement means consider the following sentence:

$$\mathcal{F} \models (\forall A, S)\mathcal{D}_{\mathcal{F}}(A, S) \supset \mathcal{R}_{\mathcal{F}}[w(do(A, S))] \tag{20}$$

where $(\forall S)w(do(A,S))$ is an element of $\mathcal{F}_{\forall s}$. (20) appears to be s-admissible itself, moreover it is provable disregarding whether $\mathcal{F}_{\forall s}$ is closed under regression or not. On the other hand, if $\mathcal{F}_{\forall s}$ is really closed under regression w.r.t. $\mathcal{F}$, then it entails the (20) itself. One may say that $\mathcal{F}_{\forall s}$, which is closed under regression w.r.t. $\mathcal{F}$, captures all the s-admissible facts about the domain.

There is another consequence of the Theorem 3.11

Corollary 3.12 (Consistency of domain axiomatization) *Suppose $\mathcal{F}$ satisfies the conditions of Theorem 3.11. Then $\mathcal{F}$ is satisfiable if and only if $\mathcal{F}_{uns} \cup \mathcal{F}_{\forall s}$ is.*

3.2 Pattern databases

The second major impact on current work was by the research initiative in computer gaming started by J.C. Culberson and J.Schaeffer [5]. In general the concept of pattern database relies on *state space abstraction*, as concisely introduced in [10]. State space abstraction is a mapping of complete state space to some partial representation. Before moving any further let us take a look at elements defining the state space.

According to [10], there are two of them: the initial state s_0 and a set of state operators O. Consequently the state space is recognized as the transitive closure of the operators on the initial state.

Given such definition, a state space abstraction is ruled by a functional mapping ϕ operating on labels of the domain as given:

$$\phi(l) = k \tag{21}$$

where l and k are both label sets. The choice of functional mapping differs upon the approach. In case of [10], ϕ is regarded as non-injective, i.e. $|k| < |l|$. Such approach, however, lacks some generality, since it is applicable only to certain permutation groups.

[7] proposes a different approach, based on grouping fluents into mutually exclusive groups (see Definition 3.13) and then mapping over those.

Definition 3.13 (Mutually exclusive groups of fluents) *Let F_{ground} be a set of grounded predicates and $g = \{g_1, ..., g_k\}$ with $g_i \subseteq F_{ground} \cup \{\texttt{true}\}$ for $i \in \{1, ..., k\}$, then g is said to contain mutually exclusive groups (mutex groups) if and only if $f_i \neq f_j$ for $f_i \in g_i \backslash \{\texttt{true}\}$ and $f_j \in g_j \backslash \{\texttt{true}\}$*

A state for a domain is then constructed from the set of mutex groups by picking an element from each group and joining them via conjunction. All states represented this way span the state space.

Following the approach the functional mapping extends original ground set F_{ground} by one more element, `true`. Then state space abstraction becomes what is specified by Definition 3.14

Definition 3.14 (State space abstraction) *Let F_{ground} be a set of ground elements, and $\mathcal{G}$ be a set of mutex groups. A state space abstraction is a mapping from F to $F \cup \texttt{true}$, defined by case distinction:*

$$(\forall f \in F_{ground}, g \in \mathcal{G}).\phi(f) = \begin{cases} f, & f \notin g \\ \texttt{true}, & f \in g \end{cases} \tag{22}$$

In order to illustrate the notions presented above some mutually exclusive groups for the table domain are presented.

Example 3.15 (Mutex groups for the table domain) *A possible partitioning on mutually exclusive groups could be based on fluent names, if those are known to be unique in the domain.*

$$\begin{aligned} g_1 &= \{has\_in\_hands(nothing), has\_in\_hands(a), has\_in\_hands(b)\} \\ g_2 &= \{on\_table(a), \texttt{true}\} \\ g_3 &= \{on\_table(b), \texttt{true}\} \end{aligned}$$

Evidently, not all the states generated through joining elements of the groups are reachable from initial state.

[7] also argues that two instances of the functional mapping (ϕ_{odd} and ϕ_{even}) appear to be especially useful. ϕ_{odd} maps all elements of groups with odd indices to `true`, conversely ϕ_{even} does the same to the elements of even indexed groups. Applied to a sample sentence $t \equiv \{has\_in\_hands(b), on\_table(a)\}$, both of the mappings would ignore the G_3 group, since it is not present in the description, and produce the following output $\phi_{even}(t) = \{has\_in\_hands(b)\}$, and $\phi_{odd}(t) = \{on\_table(a)\}$. Note that fluents are preserved/removed not on the basis of their indices in the sentence, but on the basis of the corresponding group indices, as given by the definition above.

At this point everything is ready to formally define the pattern databases

Definition 3.16 (Pattern databases) *Let $\mathcal{A}$ be an abstract state space with ϕ as a corresponding state abstraction and $\delta_\phi(s_1, s_2)$ be a graph-theoretical shortest path between two states. Furthermore, let s_0 be initial state and s_t be the set of goal states in state space. A pattern database (PDB) is a set of pairs, with the first component being an abstract state s and the second component being the minimal solution length in the abstract problem space, presented as follows:*

$$PDB(\phi) = \{(s, \delta_\phi(s, \phi(s_t)))|s \in \mathcal{A}\}$$

Represented this way, the abstraction is much smaller then the original state space, in fact the shrinking is exponential, e.g. if state size is n and $\alpha = |k|/|l|$ is the *abstraction quotient* of the mapping operator, then 2^n vectors of the original state space are shrunk to $2^{\alpha n}$ elements in the abstract state space, where $0 < \alpha < 1$. Moreover, once computed the PDB could be accessed via simple table lookup.

State space abstraction could be used in a number of ways to increase the performance of the gaming program. The approach picked in [6] argues that although pattern databases could not be used for actual search, they are useful for effective guiding of the directed search through the actual state space.

3.2.1 Application to A^* search

Efficiency of any A^* search depends on the efficiency of the underlying heuristics, which subsequently depends on the lower bound search cost estimation and it's quality. Since a state space abstraction is constructed out of the actual state

space it can not overestimate the amount of transition between the actual states, i.e. the heuristics based on the state space abstraction is admissible.

A state abstraction does not completely describe a state, thus one can not hope for reaching the goal immediately after reaching the corresponding state abstraction. Alternatively, a state abstraction could be viewed as a sub-goal on the path of reaching the actual goal. Moreover, elements of the state space abstraction are interconnected in the same way as states of the state space. This fact hints, that there exists some distance measured in the amount of transition between patterns. A quote from [6] brings all the points together:

> We adopt this idea[5] to the problem of finding optimal solution paths in a single-agent search by noting that knowing the minimum distance from a permutation to the nearest other permutation partially matching the goal is a lower bound on reaching the goal.

A very similar approach was used for successful solving combinatorially complex problems like 24-Puzzle, Rubik's Cube [15] as well as quite general problems of planning [7]. Both of the approaches proved to be computationally beneficial.

A problem with pattern databases resides in the size of the actual database constructed and consequently in the time for constructing one. If the shrinking quotient is close to 1, the pattern database is of little use itself. Conversely, if the shrinking quotient is too small, the pattern database is too small, since heuristics, based on it highly underestimates the cost of reaching the goal. An elegant solution to this issue is the introduction of *disjoint pattern databases* [16].

3.2.2 Disjoint pattern databases

Construction of a single PDB, even with high shrinking factor, becomes both computationally and physically (in terms of physical memory) expensive once the size of underlying domain is increased. In order to preserve the benefit of PDB usage in comparison to it's construction, one could think of separating the single PDB into several disjoint parts. So that answers to the question of shortest path length could be summed up.

This way one arrives at the notion of disjoint pattern databases.

Definition 3.17 (Disjoint pattern databases)
Two pattern databases ($\texttt{PDB}(\phi_1)$ *and* $\texttt{PDB}(\phi_2)$*) are said to be disjoint, if the sum of respective heuristic estimates always underestimates the overall solution length.*

Formally, if $\delta(a, b)$ *is a graph-theoretical shortest path between* a *and* b*,* ϕ_1*,* ϕ_2 *are state space abstraction and* $s, s_g \in \mathcal{P}$ *are a state and a goal state in state space respectively, then the following must hold for a state space abstraction to be disjoint:*

$$\delta(\phi_1(s), \phi_1(s_g)) + \delta(\phi_2(s), \phi_2(s_g)) \leq \delta(s, s_g), \forall s \in P$$

[5]The idea is that a sequence of sub-goals leads to the overall goal, where sub-goals are represented by state abstractions (also referred to as "permutations" or "patterns") in a state space abstraction.

The benefit from disjoint pattern databases is based on the point that it is computationally easier to construct (and later access) several small databases than one big, especially if combined together they also provide admissible lower bound.

However the way PDBs are introduced in Definition 3.16 does not always yield disjoint pattern databases, an example from [7] illustrates this point.

Example 3.18 *Imagine a case where some goal statement contains two predicates p_1 and p_2, that belong to different groups. And an operator o would make both of them* `true`*. Then distance in both abstractions is 1, due to admissibility of abstraction. But then their sum is bigger than actual distance to the goal.*

In order to address this issue two more definitions are supplied.

Definition 3.19 (Independent abstraction) *A set of group indices I is said to be independent if no operator affects both atoms in groups of I and atoms in groups outside I. The corresponding abstraction ϕ_I is also referred to as independent.*

Using the independent abstraction, one can define the sufficient condition for a pattern database being disjoint.

Definition 3.20 (Sufficient condition) *A partition of the groups into independent abstraction sets produces disjoint pattern databases.*

Note also that disjoint pattern databases would be touched once again in Section 5.3, as possible future development of End Game databases in GGP.

4 Contribution

This chapter reflects the main contribution of the current work, which is a sound and complete regression procedure for states or rather state descriptions within the GGP setting. Some practical results and measurements are presented at the end of the chapter.

4.1 Prerequisites

Due to theoretical similarity between Fluent calculus and GDL (see Section 2.3) and proven [29] practical efficiency of Fluent calculus utilization for GGP needs, this work adopts some notions common to Fluent calculus including the notions of state. Therefore, state is recognized as list of ground fluent instances combined by $\circ$ operator.

4.1.1 State description

However operating on states is not always beneficial, since the state space is expected to be too large to fit into the memory of a contemporary computer. In order to simplify operations on a group of states that exhibit similar properties one can utilize the notion of a *state description*. State description definition relies on the simple formula definition as given by Definition 2.9. The wording of the definition is repeated here for convenience purposes with some specifications, in order to fit precisely into the GGP setting.

> A formula f is said to be "simple w.r.t. s" if it mentions only domain predicate symbols, whose fluents do not mention the function symbol *do* (this transforms into occurrence of successor state axioms and action precondition axioms or *next* and *does* predicate symbols respectively), which do not quantify over variable of sort *situation*, and which have at most one free variable s of sort *situation* (no comparison in between states and the whole formula is speaking about single state at once).

Definition 4.1 (State description) *A state description $\mathcal{F}_{SD}$ is a conjunctive formula, which mentions only fluents, domain facts (GDL-predicates with empty body) along with in-/equality on fluent functional symbols, and is simple w.r.t. s.*

In other words, a state description presents flat version of a predicate describing a state.

Example 4.2 (In-/valid state description) *The next three formulas are not state descriptions:*

$$line(x, S) \wedge holds(control(xplayer), S)$$
$$legal(xplayer, A, S) \wedge A = mark(1, 2)$$
$$holds(cell(1, 2, b), S) \wedge next(cell(1, 2, x), S)$$

The first and second formulas here are invalid state description since they mention non-atomic predicate symbols. Practically, $legal/3$ could be an atomic

symbol, like in the Maze game, see (31) for example. But this is a domain-specific rule and is not generally applied. The third formula mentions the next predicate symbol, which is prohibited by definition of simple formulas.

Following are all valid state descriptions:

$$cell(A,1,b) = cell(B,1,b) \wedge holds(cell(B,1,b),S) \wedge holds(cell(B,2,b),S) \wedge holds(cell(A,2,x),S)$$
$$holds(step(A),S) \wedge succ(1,A)$$
$$\neg holds(cell(1,A,b),S)$$

The above example was taken from domain of the Tictactoe game. The fluent *cell* represents the game grid, with the first two arguments as coordinates and the third argument representing the state of the cell: b for "blank", x and o for being marked by player of X and player of O respectively.

It is easy to notice that none of the valid examples above mentions the complete grid and/or complete game state, which includes some other fluents as well. A state description does not necessarily describes the complete state, this way one can talk about *complete* and *incomplete* state descriptions. Both of them are valid.

Logically the state description could be sound or not. A *sound state description* does not contain any contradictions within, whereas an unsound state description does.

Example 4.3 (Not sound state descriptions) *Following are examples of possible valid but not sound state descriptions:*

$$holds(cell(1,A,x),S) \wedge x = b$$
$$holds(cell(1,B,b),S) \wedge 1 \neq 1$$

Moreover a state description could point to a state that is not reachable from the initial state by subsecutive application of action precondition axioms and successor state axioms, a so called *ghost state* (as introduced in Section 3.1). Such state descriptions are referred to as *ghost state description* within this paper.

Example 4.4 (Ghost state description for the Tictactoe game)
Example of ghost state description for Tictactoe game:

$$holds(cell(1,1,o),Z) \wedge holds(cell(1,2,o),Z) \wedge holds(cell(2,1,o),Z) \wedge holds(cell(2,3,o),Z) \wedge holds(cell(3,2,o),Z)$$

Here Z stands for some state of Tictactoe. Evidently, such a state description would never match to any actual state in the game unless the rules of the whole game are changed, i.e. the first turn is given to side playing O's. Note also that ghost state description does not necessarily satisfies the domain constraints, like single piece of gold in the Maze game.

Example 4.5 (Ghost state description for the Maze game)
The following formula is a valid, consistent and incomplete ghost state description.

$$holds(gold(a),S) \wedge holds(gold(b),S)$$

Such state descriptions could arise from regressive application of successor state axioms and action precondition axioms, that have incompatible bodies. For instance the above state description was produced by the following successor state and action precondition axioms:

$$does(r, grab, A, S) \wedge holds(cell(X), S) \wedge holds(gold(Y), S) \wedge X \neq Y \supset next(gold(Y)) \quad (23)$$

$$holds(cell(X), S) \wedge holds(gold(Y), S) \supset legal(r, grab, S) \quad (24)$$

The successor state axiom (23) says that if actor r performs action *grab* while residing in a cell different from the one where the gold is located, then the position of the gold remains unchanged. Such a successor state axiom is clearly valid. However if the action precondition axiom (24) is the only one that specifies action *grab*, then performing this action in a cell different from the one where gold is located is trivially illegal. Therefore, the successor state axiom (23) is never used.

Contrary to the conclusion given above, one can not identify unusable actions within the course of regression without some external knowledge, since both axioms are valid. For instance, some state invariants, i.e. single piece of gold in any state, could highlight inconsistency between those two axioms. Discovering state invariants would then be helpful to refine such descriptions as incorrect w.r.t. to game rules. This topic is fully discussed later in Section 5.

4.1.2 Combined action

A point that makes GGP different from planning domains described in [25] is the presence of multiple actors, which perform actions simultaneously and so have a combined impact on state change. In practice some of the actors could perform a so called *noop* action, which defaults to silent ignorance, and so make no impact on state change at all. For instance turn taking games are represented this way. But from a theoretical stand point, *in every state every player must perform an action*, which is valid w.r.t. game rules, in order to bring the game to a next state.

Therefore one talks about a *combined action* as the combination of actions of all the players.

Definition 4.6 (Combined action) *Suppose that Roles is the set of all players taking part in a game. Suppose also that* $Actions_{role}$ *is the set of all the possible actions that a particular player can take withing the game.*

Then a set $\mathcal{A}_{\forall r}$ *of tuples of the form* $(role, action)$ *is said to be a combined action if and only if both of following statements hold:*

- *Single action for a role:*

$$r \in Roles, a, a' \in Actions_{role}.(r, a) \in \mathcal{A}_{\forall r} \wedge (r, a') \in \mathcal{A}_{\forall r} \supset a = a'$$

- *Each role is represented within* $\mathcal{A}_{\forall r}$*:*

$$(\forall r)(\exists a) r \in Roles, a \in Actions_{role}.(r, a) \in \mathcal{A}_{\forall r}$$

Note that validity of $\mathcal{A}_{\forall r}$ deals only with syntactic correctness of the $\mathcal{A}_{\forall r}$ structure, and has nothing to do with the game rules. Example 4.7 shows some possible combined actions for games of Tictactoe and Maze.

Example 4.7 (Valid combined action) *Some possible combined actions for the Tictactoe game:*

$$[(xplayer, mark(1,2)), (oplayer, mark(1,1))]$$
$$[(xplayer, noop), (oplayer, mark(2,2))]$$

Combined action for the single-player game of Maze:

$$[(r, grab)]$$

4.2 Algorithm

The general approach to regression (also employed in [25]) is a two-stage procedure that first maps fluents against suitable successor state axioms and then maps actions in resulting formulas against suitable action precondition axioms.

4.2.1 Previous action description

Note that according to [25] the set Θ defining the regression operator $\mathcal{R}_\Theta$ contains exactly one successor state axiom for each fluent. In case of GGP, predefining all possible Θ-sets has little effect, since only some of them will actually take part in regression. Therefore, a different approach is applied. Each fluent is replaced with a set of suitable successor state axioms, then all possible combinations of action precondition axioms are generated, by picking a single axiom from every set.

The result of permutation is referred within this paper as *previous action description.*

Definition 4.8 (Previous action description) *Suppose that $\mathcal{F}_{SD}$ is a valid state description and $\mathcal{F}_{\mathcal{A}}$ is a conjunctive formula with elements among does statements, that has at least one conjunct and contains at most one does statement for each of the actors, taking part in a game. Then a previous action description is defined as*

$$\mathcal{F}_{PAD} \equiv \mathcal{F}_{SD} \wedge \mathcal{F}_{\mathcal{A}}$$

Such definition ensures that at least one action by one of the actors is known, so during the course of regression the state description is moved one turn backward. In other case one might get trapped by a situation when only frame axioms are applied and newly regressed state is defaulted to a copy of the current one, i.e. no regression is actually performed. The following example shows a previous action description, generated while regressing the goal statement of the Tictactoe game.

Example 4.9 (Valid previous action description) *Sample of previous action description generated in the course of goal regression for the Tictactoe game:*

$succ(1,2) \wedge cell(A,1,x) = cell(B,1,x) \wedge$

$holds(cell(B,1,b),S) \wedge succ(2,3) \wedge$
$cell(A,2,x) = cell(C,2,x) \wedge holds(cell(C,2,b),S) \wedge$
$cell(A,3,x) = cell(D,3,x) \wedge holds(cell(D,3,b),S) \wedge$
$does(xplayer, mark(C,2), \mathcal{A}_{\forall r}, S) \wedge$
$does(oplayer, noop, \mathcal{A}_{\forall r}, S)$

Clearly, due to the associative property of logical conjunction the *does* statements could also occur inside the formula, which is the case in practice. The main point to note now is that there must be at least one *does* statement present and at most one *does* statement for each of the actors, taking part in the game.

4.2.2 Predecessor state description

Within the next regression stage the actions that are expected to take place are matched against their preconditions to form the *predecessor state description*, which in turn could be used for further regression. This makes the definition of predecessor state description trivial.

Definition 4.10 (Predecessor state description) *The predecessor state description is a state description (as specified by Definition 4.1), that is obtained from regressing the original state description by one combined action (also referred as "game turn", or "step").*

Like original state descriptions the predecessor state description could be complete or incomplete as well as sound or not. Whereas both complete and incomplete predecessor state descriptions are of value for the purpose of the current work, one would like to avoid the unsound descriptions from playing role in further regression.

In order to separate sound descriptions from not sound ones, a model is constructed for every predecessor state description. The construction of the model involves some additional knowledge about the game description, i.e. identified variables domains. Construction of variable domains is discussed in [29].

Example 4.11 (Valid predecessor state description) *The following is a state description obtained from regressing a goal formula in the Tictactoe game by one step:*
$succ(1,2) \wedge cell(A,1,x) = cell(B,1,x) \wedge$
$holds(cell(B,1,b),S) \wedge holds(control(xplayer),S) \wedge$
$holds(cell(B,1,b),S) \wedge succ(2,3) \wedge$
$cell(A,2,x) = cell(C,2,x) \wedge holds(cell(C,2,x),S) \wedge$
$x \neq b \wedge cell(A,3,x) = cell(D,3,x) \wedge$
$holds(cell(D,3,x),S) \wedge x \neq b$

4.3 Algorithm complexity

As it was mentioned above there are three major parts in the whole regression procedure:

- **Fluents**. All of the fluents have to be regressed in order to produce the predecessor state description. Therefore the number of fluents defines the amount of successor state axioms that could be applied.

- **Successor state axioms**. There could be several successor state axioms applicable to particular fluent. Each variant produces a different regression direction and can not be discarded.

- **Action precondition axioms**. As in previous point, there might be more than one action precondition axiom that suites the action specified by a successor state axiom. Therefore the number of possible regressions is determined by number of action precondition axioms as well.

In the most general case the complexity for a single step of the regression procedure is is $|F| \times |SSA| \times |APA|$, where F stands for the set of fluents in the state description, SSA for the set of successor state axioms in the particular game, and APA stands for set of action precondition axioms in the game. $|A|$ denotes the power of the corresponding set.

Practically such an estimate is quite big, so one would like to refine each of the multipliers, in order to keep the product as small as possible. Thus, out of all successor state axioms only those that describe fluents in F are considered, which is denoted by SSA_{F_i}. The same speculation is applied to action precondition axioms, APA_{A_j}. The resulting complexity is

$$|F| \times (\prod_{F_i \in F} |SSA_{F_i}| \times (\prod_{A_j \in SSA_{F_i}} |APA_{A_j}|)) \tag{25}$$

The practical benefit from such refinements highly depend on the domain in question. However there is a strong doubt that the complexity of regression could be lowered any further.

Given the complexity for a single regression step, the complexity calculation for several iterative steps is trivial:

$$(|F| \times (\prod_{F_i \in F} |SSA_{F_i}| \times (\prod_{A_j \in SSA_{F_i}} |APA_{A_j}|)))^n, n \geq 0 \tag{26}$$

It should be noted as well that (26) represents optimistic complexity assessment, since state description tends to grow during the course of regression.

4.4 Soundness and Completeness

In order to prove soundness and completeness of regression within GGP setting, let us take a look at any GDL formula that does not belong neither to the successor state axioms (SSA) not to the action precondition axioms (APA). The general form was given in Section 2.4.1. Examining body conjuncts one may encounter either of four cases:

- `distinct` predicate
- domain facts
- `does` statement
- `true` statement

One may also speculate about arbitrary predicate symbols not predefined in [20]. However those could be flattened returning to four cases specified above.

Taking a closer look at those four cases, one notices that first to do not mention any state variables (according to [20]). The later two contain state variable implicitly. Moreover the implicit state variable is meant as the current state and must be the same one for any conjunct. Thus the overall outlook of a formula in GDL could be presented as follows:

$$a_1 \wedge .. \wedge a_n \wedge b_1(S) \wedge .. \wedge b_m(S) \tag{27}$$

where $a_1 \wedge .. \wedge a_n$ stand for all the occurrences of `distinct` predicate and domain facts within the flat representation of a rule and $b_1(S) \wedge .. \wedge b_m(S)$ represent `does` and `true` statements. According to Definition 2.9 formula (27) is simple.

Taking into account that GDL rules are implication (the head is true if and only if the body is), any rule not being part neither of SSA nor of APA has the following form:

$$(\forall S).w \equiv h(S) \Leftarrow a_1 \wedge .. \wedge a_n \wedge b_1(S) \wedge .. \wedge b_m(S) \tag{28}$$

where S is the only state variable.

Now it is trivial to see that sentence w is simple and formula (28) is s-admissible. Therefore any rule of GDL not being in SSA nor in APA is s-admissible.

Next step is to check the closure under the regression precondition. The definition of the closure under regression from Section 3.1.3 is briefly outlined here for the sake of convenience.

> A set of s-admissible sentences Ω is said to be *closed under regression* w.r.t. domain description $\mathcal{F}$ if and only if whenever $(\forall)w(S) \in \Omega$ the following holds:
>
> $$\Omega \models (\forall A, S).\mathcal{D}_{\mathcal{F}}(A, S) \supset \mathcal{R}_{\mathcal{F}}[w(do(A, S))]$$

Verbose formulation of later condition yields:

> If performing an action A in a situation S is possible then regression of the sentence w applied to successor situation $do(A, S)$ must entail the action precondition axiom for the action A in the state S.

The condition is achieved by the algorithm step outlined in Section 4.2.2.

Furthermore, the algorithm is applied to an s-admissible rule of GDL and produces set of s-admissible formulas which in their turn are closed under regression themselves (by Definition 3.7). This makes the set of GDL rules (those not belonging neither to SSA nor to APA) closed under regression and satisfies[6] the preconditions of the Theorem 3.11.

As mentioned in Section 2.4.2 any domain description by GDL is a finite state machine (so there exists only a finite amount of states) and allows only finite amount of transitions between initial and goal states. The latter statement satisfies the condition of Theorem 3.11.

Finally, by Theorem 3.11 the regression algorithm within GGP setting appears to be sound and complete.

[6]Note that GDL as given in Section 2.4.1 already contains unique names assumption

4.5 Refinements

As mentioned in the previous section, the gradual refinement of each of the multipliers is chosen as high complexity treatment approach. Moreover, via refinements some useless branches of regression could be trimmed as well.

4.5.1 Consistent combined action

One can talk about combined actions being consistent w.r.t. the state in which they are performed. However in case of regression the state is unknown beforehand, so the notion of consistency is changed a bit.

In case of regression, the combined action ($c\mathcal{A}_{\forall r}$) is *consistent* w.r.t. previous action description $\mathcal{F}_{\mathcal{PAD}}$ if each *does* statement of $\mathcal{F}_{\mathcal{PAD}}$ is represented in $\mathcal{A}_{\forall r}$.

Definition 4.12 (Consistent combined action) *Formally speaking, suppose $\mathcal{A}_{\forall r}$ is a combined action and $\mathcal{F}_{PAD}$ is a previous action description. $\mathcal{A}_{\forall r}$ is called "consistent w.r.t. $\mathcal{F}_{PAD}$" if and only if each element of $\mathcal{F}_{\mathcal{A}}$ (as part of $\mathcal{F}_{PAD}$) is injective mapped into some element of $\mathcal{A}_{\forall r}$.*

In other words, each *does* predicate symbol of $\mathcal{F}_{\mathcal{A}}$, must have a corresponding tuple in $\mathcal{A}_{\forall r}$ and no player must be mentioned more than once in $\mathcal{F}_{\mathcal{A}}$.

Evidently some combined action could be consistent w.r.t. to one state,

Example 4.13 (Valid consistent combined action) *Combined action $[(xplayer, mark(X, Y)), (oplayer, Z)]$ would be consistent w.r.t. to the following previous action description:*
$succ(1, 2) \wedge cell(A, 1, x) = cell(B, 1, x) \wedge$
$holds(cell(B, 1, b), S) \wedge succ(2, 3) \wedge$
$cell(A, 2, x) = cell(C, 2, x) \wedge holds(cell(C, 2, b), S) \wedge$
$cell(A, 3, x) = cell(D, 3, x) \wedge holds(cell(D, 3, b), S) \wedge$
$does(xplayer, mark(C, 2), \mathcal{A}_{\forall r}, S) \wedge$
$does(oplayer, noop, \mathcal{A}_{\forall r}, S)$

Example 4.14 (Inconsistent combined action) *Alternatively, the same combined action $[(xplayer, mark(X, Y)), (oplayer, Z)]$ would be inconsistent if the following previous action description is considered:*
$succ(1, 2) \wedge cell(A, 1, x) = cell(B, 1, x) \wedge$
$holds(cell(B, 1, b), S) \wedge succ(2, 3) \wedge$
$cell(A, 2, x) = cell(C, 2, x) \wedge holds(cell(C, 2, b), S) \wedge$
$cell(A, 3, x) = cell(D, 3, x) \wedge holds(cell(D, 3, b), S) \wedge$
$does(xplayer, noop, \mathcal{A}_{\forall r}, S) \wedge$
$does(oplayer, noop, \mathcal{A}_{\forall r}, S)$

Within the current work the following two conventions are adopted as first refinement rule:

- A previous action description is considered for regression if and only if there exists a combined action, that is consistent w.r.t. it.
- A state description is considered for regression if and only if one of its previous action descriptions is considered for regression.

4.5.2 Recognized structures

Importance of state invariants was already mentioned in Section 4.1.1, this section provides some ideas on how similar outcome could be achieved without actual identification of state invariants. The approach which is actually employed in the regression algorithm implementation relies on structures recognized inside of state.

It is clear that most of human-oriented games use some kind of intuitive (often visually-intuitive) representation of the game environment, like grids, boards, maze-maps and tracks. But the formal representation of such environments is usually not intuitive and may considerably vary among different axiomatizations of the same game. Following example shows two ways of representing the central cell of the Tictactoe-grid

Example 4.15 (Tictactoe axiomatization differences) *A common way to represent a cell for the Tictactoe-grid is a function symbol "cell" with arity 3, where first two arguments denote the "x" and "y" axis coordinates, and third one stands for the value of the cell.*

$$cell(2, 2, b)$$

Alternatively one can name each cell of the Tictactoe-grid with a number or a letter, and refer to it via function symbol with arity 2. In sample axiomatization given below, all the cells are named with letters from "a" to "b". The fifth letter in the alphabetic order would denote the central cell of the Tictactoe-grid.

$$cell(e, b)$$

Disregarding which axiomatization approach is chosen it is beneficial to recognize some common behavior of game rules regarding the functional symbols. An instance of such common behavior in case of the Tictactoe-game is the fact that there always exist exactly nine fluents and only one argument of of a fluent is changed during course of the game. The task is even made simpler if one knows what to look for, i.e. what behaviors are common.

Once such a behavior is identified within game rules, one can talk about a structure being *recognized*.

Definition 4.16 (Recognized structure) *A function symbol is called a recognized structure if there exists a single signature that identifies input and output arguments associated with it.*

Formally speaking, suppose that $m_{\mathcal{L}}$ is a mapping function for some domain axiomatization $\mathcal{L}$, which maps any function symbol within the axiomatization to set H of all the input/output signatures discovered for the function symbol. Then a function symbol f is referred as recognized structure if and only if:

$$m_{\mathcal{L}}(f, H) \supset \exists! h \in H$$

where $\exists!$ stands for "exists single".

Depending on the kind of signature associated with the function symbol in question, one could talk about 0-ary functions, n-ary functions and other relation types. The functions with arity 0 (constants), could appear at most

once in state description. The functions with arity different then 0 are treated differently. For instance, 1-ary functions could be viewed as mappings, maps or tracks, the 2-ary functions could be viewed as boards or grids, etc. The following example shows some structures that could be recognized withing the GGP-domain

Function symbol	Signature	Structure
control/1	$\emptyset$	constant
map/2	(1)	mapping, track
cell/3	(1,2)	2-ary function, board

Table 1: Sample recognized structures

Example 4.17 (Sample recognized structure)

The second refinement adopted for the current work relies on two following rules:

- A state description is sound if it contains at most one instance of fluent (function symbol) recognized as a constant.
- A state description is sound if there are no fluents recognized as n-ary functions with same input arguments and different output arguments.

Logically, in case some state description is identified as not sound, it is not considered for the rest of regression course.

An issue associated with the structure recognition procedure is that in order to obtain a complete solution either the whole state space must be traversed or the a theorem prover could be used. Since state space is expected to be bigger than memory of any contemporary computer, the traversal would take far too long. And approach of employing a theorem prover is known to suffer from undecidability of underlying first-order logic. An alternative, but formally incomplete solution would be to pick a considerable amount of random states and test candidates for recognized structures against those states. This approach was first pointed to by [17] and later also adopted in [29].

4.5.3 Formula sub-descriptions

As explained in Section 2.1, the frame axioms are used in order to describe all the non-effects of some action. But within goal regression setting, the frame axioms could act as "false friends" as well.

Imagine an action that does not change any fluents relevant to a state description. Then performing such action would not bring an actor any further to the goal, even though the action along with corresponding predecessor state description is sound.

The current work refers to such predecessor state descriptions as *formula sub-descriptions*, and treats them as undesired or illegal regression outcomes.

Definition 4.18 (Formula sub-description) *A predecessor state description is regarded a "formula sub-description" if and only if all the models satisfying*

the predecessor state description satisfy the state description being regressed as well.

Formally speaking, suppose that regression operator $\mathcal{R}_\Theta$ applied to formula F produces W as one of the predecessor state descriptions, and $\mathcal{M}_W$ is the set of all the models satisfying W. Then W is regarded as formula sub-description if and only if the following statement holds:

$$(\forall m \in \mathcal{M}_W).F(m)$$

where $F(m)$ stands for successful application of model m to state description F.

A drawback of such refinement rule is the fact that generation of all possible models of a formula depends on the domain sizes of variables in the formula, which could be computationally expensive in many cases.

Example 4.19 (Formula sub-descriptions)
Consider the Maze-game domain, there are actions which change the location of a robot, and actions that change the location of the gold. Thus relocating the robot does not influence the goal state description, if it specifies only the location of the gold.

Suppose the following goal state description and one of the suitable successor state axioms:

$$holds(gold(a), S) \tag{29}$$

$$holds(gold(X), S_{-1}) \wedge does(r, move, \mathcal{A}_{\forall r}, S_{-1}) \supset next(gold(X), S) \tag{30}$$

$$true \supset legal(r, move, S) \tag{31}$$

Regressive application of successor state axiom (30) to the goal state description (29), would produce a previous action description of the form:

$$holds(gold(a), S_{-1}) \wedge does(r, move, \mathcal{A}_{\forall r}, S_{-1})$$

which is turned into predecessor state description by (31):

$$holds(gold(a), S_{-1}) \wedge true$$

Disregarding the action was performed, the predecessor state description of this kind does not point to the previous action in a successful plan. And therefore, should be dismissed from regression strategy.

It must be noted however that this is not the nature of frame axioms that gives rise to formula sub-descriptions problem, but their specific combination in the course of regression. In a different setting a frame axiom could be the only correct way to regress the formula description.

Example 4.20 (Usage of frame axioms for Tictactoe)
The goal for the Tictactoe game is a line of three pieces of the same type. For instance, following formula could satisfy one of the goal definition:

$$holds(cell(X,1,x),S) \wedge holds(cell(X,2,x),S) \wedge holds(cell(X,3,x),S) \tag{32}$$

One of the regressions of this formula has following form:

$$holds(cell(X,1,b),S_{-1}) \wedge holds(cell(X,2,x),S_{-1}) \wedge holds(cell(X,3,x),S_{-1}) \tag{33}$$

However to regress on from this point, one needs an axiom that would preserve/produce the blank cell $holds(cell(X,1,b),S)$. According to the rules of the Tictactoe game, the only way to preserve/produce a blank cell is not to occupy it. Therefore following frame axiom should be used to keep regression on correct track:

$$\begin{aligned} &does(W,mark(J,K),\mathcal{A}_{\forall r},S_{-1}) \wedge \\ &holds(cell(M,N,b),S_{-1}) \wedge \\ &(distinct(M,J) \vee distinct(N,K)) \\ &\supset next(cell(M,N,b),S) \end{aligned} \tag{34}$$

4.6 Database structure

The main objective of the whole work is the automatic creation of the database that stores end- as well as pre-end states, or rather state descriptions. The database itself is written to a separate file which is later compiled and loaded to the general game player program.

An interesting point here is the structure of the database. In general, the database should exert the following properties:

Speed Taking into consideration big volume of data generated while regression, accessing the database should give minimum speed penalty to the player program.

Utilization Consequently, once the matching database entry is found, it should point to some action, that is prescribed by regression. So that there is practical use of the database construction.

An important issue related to database structure is the formula being regressed. In the most trivial case, such formula defaults to the GDL *goal* statement [20] with actor name universally quantified or rather its flattened equivalent. Storing/regressing such a goal statement is sufficient for single-player games, but has little impact in case of multi-player games, where each of the actors could have both contradicting and cooperative aims. The point is reinforced by notion that some sub-goal state for a player could appear to be a goal

state for an opponent. In this case spotting (or not) the difference could change the outcome of the game

The initial treatment idea for such a case was to regress not just a *goal* formula along with the negation of the *terminal* statement, as shown in (35).

$$goal(role, s) \wedge \neg terminal(s) \tag{35}$$

A common drawback of this approach is that in many cases definition of $terminal(S)$ coincides with definition of $goal(R, S)$ and vice versa. In other words all the terminal states a the goal states for some of the players. Thus test for logical satisfiability would often yield $\perp$, without providing any insights on the real state of affairs.

Thus, convention adopted by the current implementation states that the end game database stores the goal states with maximal values for all of the actors. In order to facilitate future database lookup it also includes corresponding actor name as an explicit argument. Partitioning the goals this way gives some advantage during regression and results in more compact state descriptions being produced.

Except performance gains such modification appears to be expressive enough to drive various conclusions about the current state of the game. For instance, regressing the formula (36) as a whole, would produce too big state descriptions even on the first step of regression for a simple game like Tictactoe.

$$goal(role, s) \wedge (\forall r \in Roles).(role \neq r \wedge \neg goal(r, s)) \tag{36}$$

With the implementation proposed such behavior could be emulated during lookup procedure as shown in formula (37).

$$\texttt{match\_EGDB}(role, s) \wedge (\forall r \in Roles).(role \neq r \wedge \neg \texttt{match\_EGDB}(r, s)) \tag{37}$$

The formula presented above corresponds to identification of winning state. Other combinations are also possible. For instance, a cooperative role in a particular state could be tested by the formula (38) and loosing state would be encoded as presented by the formula (39).

$$\texttt{match\_EGDB}(role, s) \wedge (role \neq r \wedge \texttt{match\_EGDB}(r, s)) \tag{38}$$

$$\neg\texttt{match\_EGDB}(role, s) \wedge (\exists r \in Roles).(role \neq r \wedge \texttt{match\_EGDB}(r, s)) \tag{39}$$

Quantifiers mentioned in the formulas above are not present implicitly in the implementation. The issue of handling them correctly is delegated to respective users of the end game database. It should be noted as well that given big amount of opponents formulas like (37) or (39) might increase the matching times.

A common query on a database is matching between some state (either current of searched one), and a state description corresponding to database entry. Therefore one might think of database as a binary relation `EGDB` between some state description and a combined action plan $\vec{\mathcal{A}}_{\forall r}$. There is a practical objection to this point, however. As it was mentioned in Section 4.1.1, a state description does not have state variable instantiated, thus binding a state to state description might take some time for iterating through all the conjuncts of

the state description that have a state variable. In order to avoid unnecessary coding and performance penalties, the end-game database format was chosen to be a quadruple with the actor name R, state description $\mathcal{F}_{SD}$, corresponding combined action plan $\vec{\mathcal{A}_{\forall r}}$ and a state variable S mentioned explicitly.

$$\texttt{EGDB}(R, S, \mathcal{F}_{SD}, \vec{\mathcal{A}_{\forall r}})$$

Given such structure of end-game database, one can supply instances of actor name and state as first and second arguments respectively and receive a state description with state variable bound to the state supplied. Then checking for a match between state description and the state defaults to successive execution of state description commands. In case of successful matching the combined action is returned. In case of failure the next database entry is taken by Prolog backtracking mechanism.

4.7 Performance

The expected optimistic complexity of the algorithm was already discussed in Section 4.3. This section presents some actual measurements for construction of an end-game database for two of the games. The games under consideration are simple syntactic modification of the Tictactoe game and the Maze game, which were discussed already. The modifications are limited to recursive definition of `row/2` rule in the Tictactoe game and definition of goal by contradiction to other states in the Maze game. These modifications were introduces entirely for the sake of testing the implementation and do not simplify any of the domains.

The implementation was tested in two aspects. First of all, the times for construction of the end game database where measured w.r.t. depth of the regression (Table 4, Table 5, and Table 6). Tables presented in this section denote the depth of regression in church numbers, i.e. 0 stands for no regression, s(0) indicates regression one step backwards, s(s(0)) denotes a two-step regression, etc.

Second aspect to be tested is access time (Table 3 and Table 2). For this case the databases were regressed by three steps (the maximum possible w.r.t. hardware limitations faced by the author). Matching a single state against the end game database appeared insufficient for any conclusions. Therefore a sequence of matchings was performed on the same database. Corresponding tables indicate the number of iterations in sequences and corresponding times for accessing the database. Note however that the state matched remained the same through each of the sequence, therefore some internal caching algorithms of ECLiPSe prolog could have facilitated the matching.

Amount of iterations	Access CPU Time
500	6.13s
1000	12.11s
5000	61.08s
10000	122.93s

Table 2: The Tictactoe game EGDB access time

Amount of iterations	Access CPU Time
500	0.08s
1000	0.12s
5000	0.44s
10000	0.84s

Table 3: The Maze game EGDB access time

Table 4 show some times measured for the Tictactoe-game. As one may notice, the exponential blow up really takes place, as predicted earlier.

Regression depth	CPU Time required
s(0)	0.125s
s(s(0))	0.37s
s(s(s(0)))	34.78s
s(s(s(s(0))))	68.47s

Table 4: Times for construction of the Tictactoe-game EGDB.

It is also worth mentioning that, the construction of database does not take operating system level penalties (like opening a file, writing to file, flushing the buffer, etc.) into consideration. Since these times highly depend on implementation and current state of underlying operating system and can radically change the overall time required for writing the database, only times for actual computation of end game database were considered. The same holds for construction of end game database for the Maze game presented in Table 6.

Regression depth	CPU Time required
s(0)	0.0s
s(s(0))	0.2s
s(s(s(0)))	0.2525s
s(s(s(s(0))))	1.36875s

Table 5: Times for construction of the Maze-game EGDB (STRIPS aware case)

There is a small twist about the Maze game. The actual description of the Maze game fits into STRIPS domain [23], which could facilitate the construction of the end game database for such game by order of magnitude. In order to stress this point, performance of the implementations was tested with STRIPS-aware tuning as well. Corresponding result could be found in Table 5.

As it was mentioned above, the matching of a state against the end game database was performed in sequences with the same state through out each of the sequence of matchings.

In case of the Maze game (see Table 3) the highest number of iterations does not seem rational, since it highly outnumbers the amount of possible states. Alternatively for the Tictactoe game 10,000 is almost half of all possible states, available in the game and therefore has to be considered for/before any practical

Regression depth	CPU Time required
s(0)	0.0s
s(s(0))	0.1s
s(s(s(0)))	3.49s

Table 6: Times for construction of the Maze-game EGDB

usage.

5 Future work

At this point it is clear that automatic end-game database construction for general games is far from being as efficient as for specific games. Evidently there are some aspects to improve. This section covers just some of them, those that seem to have the major impact on efficiency of EGDB construction.

5.1 Model construction

First of all, the construction of a model for a state description could default to consideration of the whole domain in the worst case. A treatment proposed by [19] was to shrink the domain so that it included only part of all the atoms sufficient for construction of a model.

Given a state description $\mathcal{F}_{SD}$ the minimal amount of atoms for a model construction would be a multi-sorted set, where sorts depend on the positions of the variables and constants inside the fluent of the description. Thus, amount of atoms of sort *sort*, would be defined by following formula:

$$|v^{sort}_{\mathcal{F}_{SD}}| + |c^{sort}_{\mathcal{F}_{SD}}|$$

where $v^{sort}_{\mathcal{F}_{SD}}$ stands for the set of variables of sort *sort* appearing in $\mathcal{F}_{SD}$, $c^{sort}_{\mathcal{F}_{SD}}$ stands for the set of constants of sort *sort* appearing in $\mathcal{F}_{SD}$ and $|A|$ denotes the power of a set A.

5.2 State invariants

Another improvement could be derived from the identification of *state invariants*. Some work in this direction was already presented in [19]. A state invariant is treated as some statement or formula, that once becoming true in a state remains so in any successor state. Identification of state invariants would assist in goal regression by trimming considerable amount of ghost states and their descriptions.

Definition 5.1 (State invariants) *A state invariant is a formula that once became valid in some state s remains valid in any successor state of s.*

Formally speaking, consider some formula W that mentions no successor state or action precondition axioms, i.e. no does, next, legal statements in case of GDL-domains, and is simple w.r.t. to s. Consider also some state s_1, and an action collection $\mathcal{A}_1...\mathcal{A}_n$, that is possible/legal in the state s_1 and corresponding successor states $s_2...s_n$, $n \geq 1$. Formula W is referred as state invariant for s if and only if

$$W(s) \supset W(next(\mathcal{A}_1...\mathcal{A}_n, s_1))$$

where $next(\mathcal{A}_1...\mathcal{A}_n, s_1)$ stands for successive application of possible actions in order to produce successor states $s_2...s_n$.

Examples of state invariants can vary from some specific statements about a concrete game match to general observation on game rules. Example 5.2 sketches some of the possible state invariants.

Example 5.2 (State invariants) *A state invariant could talk about some specific game match like the the match of the Tictactoe game, when the central cell is occupied.*

$$W \equiv holds(cell(2, 2, X), S) \wedge X \neq b$$

It could also mention some global game properties like, the fact that no pawns are allowed to move back in chess:
$W \equiv holds(cell(X, Y, wp), S) \wedge Y \geq 2$

At this point it should be noted that what is know as "state invariants" could be further specified into two categories: invariants and domain constraints

State invariants As defined above, a statement that became true in one state and remains true in all successor states.

Domain constraints A statement that is true in all the states, throughout any match of the game, in other words, state invariants for initial state s_0.

Both of these are of value, the domain constraints are more applicable to the case of goal regression however, since they should hold in any of the states, including goal and sub-goal ones. The following example provides some domain constraints for the Maze and the Tictactoe games

Example 5.3 (Domain constraints) *An observation about the Maze game that only a single piece of gold exists at any given moment of time:*

$$(\forall S)(\exists! X).gold(X) \in S$$

Moreover, this could be extended by the fact, that robot could not appear in more than one location at a time:

$$(\forall S)(\exists! X, Y).gold(X) \in S \wedge cell(Y) \in S$$

In case of classical Tictactoe game one can speculate that only one player has turn at a time:

$$(\forall S)(\exists! X).control(X) \in S$$

or that no cell could have more than one piece on it at a time:

$$(\forall S, X, Y, V, V').cell(X, Y, V) \wedge cell(X, Y, V') \supset V = V'$$

Identification of state invariants seems to be a complex and challenging task. Currently there already exist algorithms for identification of the state invariants (link to paper by F. Lin). However these apply only to simple planning domains. Being developed for the "simple theories", these algorithms cover only small set of games in GGP. Therefore some more general approaches are still about to emerge.

Another important point is that state invariants show interdependent relation between themselves, i.e. in order to prove some state invariants other proved state invariants are to be utilized.

5.3 Disjoint pattern databases

As it was mentioned in Section 3.2, pattern databases are more close to end-game databases for GGP then the classical notion of the end-game databases for specific games. A novel idea in the field of pattern databases talks about construction of so called *disjoint pattern databases*, i.e. scattered pieces of patterns, that do not interfere with each other. This way databases are made smaller with corresponding wins in matching procedure.

The key point for the construction of a disjoint pattern database are groups. The properties of the patterns are grouped so that interdependent properties do not go to different groups and are matched together as well.

In case of the Fluent calculus setting such a grouping could be performed on the basis of *fluent interdependencies*.

Definition 5.4 (Interdependent fluents) *Two fluents are considered interdependent in some formula (state description) if they share at least one variable.*

Formally speaking, a fluent f_1 with arity n and fluent f_2 with arity m are considered interdependent if and only if for some variable v within some formula $\mathcal{F}_{SD}$ it holds that $\mathcal{F}_{SD} \equiv ... \wedge holds(f_1(.., v, ...), S) \wedge ... \wedge holds(f_2(.., v, ...), S) \wedge ...$

Direct application would then group non inter-depended fluents into separate groups and regress them separately. Matching procedure would then look into all of the databases in turn and guide the search on the game state space.

Example 5.5 (Interdependent fluents) *Some possible grouping for the Tic-tactoe game:*

G_1 $holds(cell(X, 1, x), S) \wedge holds(cell(X, 2, x), S) \wedge holds(cell(X, 3, x), S)$
G_2 $holds(control(xplayer), S)$

An important difference between disjoint pattern databases and end game databases for GGP lies in the scope of space coverage. The pattern databases, disregarding whether they are disjoint or not, cover the whole state space. This is due to the fact that pattern databases are constructed from initial state onwards following the transitions. Conversely the end game databases are constructed by regressing the goal state, which covers only part of the whole state space.

Though constructed in different ways disjoint pattern databases and end game databases share some common issues. In particular the issue of evaluating some pattern (state description) w.r.t. opponent's moves that are not influences by the actor. Solution employed by the current work (see Section 4.6) is to combine values of state comparison from actor's and opponent's perspectives.

5.4 Decision trees

As it was recently proposed, decision trees could be used as a structure for storing actual state descriptions instead of plain recording each of them into database. The idea behind the decision trees captures the point, that some state descriptions may contain the same fluent but within opposite scopes, i.e. one could contain a fluent within negative scope, whereas the other could contain the same fluent but within the positive scope. Knowing this would be sufficient to separate two descriptions and never consider them together. Same process could

be then repeated on other elements of the descriptions separating the whole regression into small categories. Matching would then entail going through the state and comparing fluents to the tree nodes of decision tree. Arriving to the leaf of the tree (in the ideal case) one would obtain a concrete action or a plan of actions for reaching the goal. A worse case would hold a collection of reduced state descriptions on the leaves of the decision tree. Dealing with later ones would default to matching current state against each of them, which is similar to the behavior currently implemented.

Example 5.6 (State matching based on decision trees) *A possible implementation of decision tree, as underlying structure for end game database, could have a layout where an end game database record would contain the following fields:*

- *name of the actor for which particular regression was constructed,*
- *probably an explicit argument for a state to facilitate the unification*
- *the decision tree itself, with fluents and/or formulas as tree nodes, and actions as the tree leaves.*

The following listing provides a sample record of end game database implementing decision trees as underlying structure for state matching:

```
egdb(r, _S,
  tree(holds(gold(i), _S),
    tree(holds(cell(a), _S),
      leaf(drop),
      leaf(move)
    ),
    tree(holds(cell(a), _S),
      tree(holds(gold(a), _S),
        leaf(pickup),
        leaf(move)
      ),
      leaf(move)
    )
  )
).
```

A possible drawback of such method is that it implies state descriptions being quite polar, i.e. containing sufficient amount of common fluents under opposite scopes. In practice such behavior occurs rarely, since initial state descriptions for regression are obtained from the goal description, which in its turn tends to be coherent rather then polar. Moreover some games (both of the games addressed in the current work) have no negations within their descriptions. Therefore constructing decision trees out of those would result in multiple trees with low (if any) branching, which corresponds to single record per state, as used by the implementation.

6 Conclusions

General Game Playing appears to be a new type of challenge for computer gaming and AI. Though being a relatively new research area, it attracts considerable attention of scholars.

The current work contributes to the field by providing method for automated construction of end game databases suitable for GGP purposes. It was shown that roots of the method rest on regression introduced by the Situation Calculus, [25]. Section 4.4 proves that the method presented is sound and complete under GGP setting.

There are some items w.r.t. implementation which should be discussed, however. The topmost issue is the complexity of the algorithm. A hidden drawback of the approach taken within the work is that all the formulas are "flattened" before regressing them or applying them within the course of regression. In many cases the flattening of the formula is too expensive compared to time available in GGP setting. Moreover, as it is noted in Section 4.3 and Section 4.7, depending on the nature of domain description the regression procedure could experience exponential blowup. Therefore under time limit constraints only a couple of regressing steps are possible, mostly with quite simple single- or two- player games. As it was stated in Section 1, for the current work only the Tictactoe and the Maze games were used for benchmarking/testing.

The positive side of the medal lies in the soundness and completeness of the algorithm. Which provides the action plan (a sequence of actions) for reaching the goal starting with the state satisfying the state description.

One should note that action plan is not usually ground. In other words, it is usually impossible to execute the plan without simulation of the game (or without going through actual real game). This is due to the fact, that some actions are described with variables in place of action arguments. Which is done for the sake of generality. The action variables then are easily substantiated for actual values as the game (or the state space search) takes its place. Contrary,in the case of regression the generality is preserved to keep the number of state descriptions generated low. Thus actions of an end game database record can contain variables, which should be matched to real values among those in legal actions for the actual state in question.

It is natural that regression is taking its course without any impact from the initial state. Thus all the possible regressions (not only reachable) are computed, recall the ghost situations in Example 3.1. The direct outcome of this fact is that only part of end game database is of actual use. The rest of it could have been useful if initial state of a game was changed every time the game is played. Unfortunately this is not the case with GGP.

Talking about implementation one should note that at the moment of writing the goal regression database is not fully integrated into FLUXPlayer, i.e. one can regress goals and use the results, but it is not done by current search function. As it is clear by now, though being based on real games all the tests were manually supplied to the program.

Concerning possible improvements and further development, as noted in Section 5.2, state invariants, according to the author's persuasion, could have significantly improved the performance of the algorithm by keeping the $|\mathcal{F}|$ multiplier in formula (26) bound. Therefore discovering the state invariants could be nominated as the next most appealing challenge for GGP-researchers.

Another possible further application of regression is described in Section 5.3. After success of Chinook, checkers playing program, the pattern databases and later the disjoint pattern databases received a lot of attention from the side of scientific community. The authors opinion is that disjoint pattern databases could be effectively applied and constructed within GGP setting as well.

To conclude the writing some notes on style of the paper are given. The paper itself draws a coherent line from the basics of the Situation Calculus, used as theoretical background, through the Fluent Calculus, used for the sake of the method implementation, to current practical efficiency of the method. Some of the related fields are referenced as well. A small section discussing the implementation of the method rounds up the contribution. Outline of some aspects for further development and research in the area concludes the writing.

References

[1] S.E. Bornscheuer and M. Thielscher. Explicit and Implicit Indeterminism: Reasoning About Uncertain and Contradictory Specifications of Dynamic Systems. *Journal of Logic Programming*, 31(1):119–155, 1997.

[2] B. Bouzy and T. Cazenave. Computer Go: An AI oriented survey. *Artificial Intelligence*, 132(1):39–103, 2001.

[3] M. Buro. Logistello— A strong learning Othello program, 1997.

[4] KL Clark. Negation as failure. *Readings in nonmonotonic reasoning table of contents*, pages 311–325, 1987.

[5] J.C. Culberson and J. Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.

[6] J.C. Culberson and J. Schaeffer. Searching with pattern databases. *Lecture Notes in Computer Science*, 981:101–112, 2001.

[7] S. Edelkamp. Planning with pattern databases. *European Conference on Planning (ECP)*, pages 13–24, 2001.

[8] H. Gallaire, J. Minker, and J.M. Nicolas. Logic and Databases: A Deductive Approach. *ACM Computing Surveys (CSUR)*, 16(2):153–185, 1984.

[9] M. Ginsburg and D.E. Smith. Reasoning about action II: the qualification problem. *Artificial Intelligence*, 35(3):311–342, 1988.

[10] I.T. Hernadvolgyi and R.C. Holte. Experiments with automatically created memory-based heuristics. *Proceedings of the Symposium on Abstraction, Reformulation and Approximation (SARA-2000), Lecture Notes in Artificial Intelligence*, 1864:281–290, 2000.

[11] S. Hölldobler and J. Schneeberger. A new deductive approach to planning. *New Generation Computing*, 8(3):225–244, 1990.

[12] S. Holldobler and M. Thielscher. Computing change and specificity with equational logic programs. *Annals of Mathematics and Artificial Intelligence*, 14(1):99–133, 1995.

[13] F.H. Hsu. IBM's Deep Blue Chess grandmaster chips. *Micro, IEEE*, 19(2):70–81, 1999.

[14] G.A. Keim, N.M. Shazeer, and M.L. Littman. Proverb: The Probabilistic Cruciverbalist. *New York Times (NYT)*, 792(10):70, 1999.

[15] R.E. Korf. Finding optimal solutions to Rubik's Cube using pattern databases. *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 700–705, 1997.

[16] R.E. Korf and A. Felner. Disjoint pattern database heuristics. *Artificial Intelligence*, 134(1):9–22, 2002.

[17] G. Kuhlmann, K. Dresner, and P. Stone. Automatic heuristic construction in a complete general game player. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence*. AAAI, July 2006.

[18] H. Levesque, F. Pirri, and R. Reiter. Foundations for the situation calculus. *Linköping Electronic Articles in Computer and Information Science*, 3(1998)(18), 1998.

[19] F. Lin. Discovering State Invariants. *Proceedings of the Ninth International Conference on Principles of Knowledge Representation and Reasoning, KR*, 4:536–544, 2004.

[20] N. Love, Hinrichs T, and Genesereth M. *General Game Playing: Game Description Language Specification*. Stanford Logic Group, Stanford University, 353 Serra Mall, Stanford, CA 94305, 1 edition, April 2006.

[21] J. McCarthy and P. Hayes. *Some Philosophical Problems from the Standpoint of Artificial Intelligence*. Stanford University, 1968.

[22] L. Pacholski and A. Podelski. Set constraints: a pearl in research on constraints. *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming-CP97*, pages 549–561, 1997.

[23] E.P.D. Pednault. ADL: exploring the middle ground between STRIPS and the situation calculus. *Proceedings of the first international conference on Principles of knowledge representation and reasoning table of contents*, pages 324–332, 1989.

[24] F. Pirri and R. Reiter. Some contributions to the metatheory of the situation calculus. *Journal of the ACM (JACM)*, 46(3):325–361, 1999.

[25] R. Reiter. The frame problem in the situation calculus: a simple solution (sometimes) and a ccompleteness result for goal regression. *In Artificial Intelligence and Mathematic Theory of Computation: Papers in Honor of John McCarthy*, 1991.

[26] J. Schaeffer. The role of games in understanding computational intelligence. *IEEE Intelligent Systems Journal*, pages 10–11, 1999.

[27] J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu, and D. Szafron. A World Championship Caliber Checkers Program. *Artificial Intelligence*, 53(2-3):273–289, 1992.

[28] J. Schaeffer, R. Lake, P. Lu, and M. Bryant. CHINOOK: The World Man-Machine Checkers Champion. *AI Magazine*, 17(1):21–29, 1996.

[29] S. Schiffel and M. Thielscher. Fluxplayer: A successful general game player. In *AAAI*, pages 1191 – 1196. AAAI Press, 2007.

[30] J. Shepherdson. Negation as failure, completion and stratification. *Handbook of AI and LP*, 1990.

[31] Stanford Logic Group. http://games.stanford.edu. GGP Project web page.

[32] M. Thielscher. Ramification and causality. *Artificial Intelligence*, 89(1):317–364, 1997.

[33] M. Thielscher. *Introduction to the fluent calculus*. Linköping Univ. Electronic Press, 1998.

[34] M. Thielscher. Reasoning about actions: steady versus stabilizing state constraints. *Artificial Intelligence*, 104(1), 1998.

[35] M. Thielscher. From Situation Calculus to Fluent Calculus: State update axioms as a solution to the inferential Frame Problem. *Artificial Intelligence*, 111(1):277–299, 1999.

[36] M. THIELSCHER. FLUX: A logic programming method for reasoning agents. *Theory and Practice of Logic Programming*, 5(4-5):533–565, 2005.

[37] M. Thielscher. *Reasoning Robots: The Art and Science of Programming Robotic Agents*. Kluwer Academic Pub, 2005.

[38] R. Waldinger. *Achieving Several Goals Simultaneously*. Stanford Research Institute, 1975.