Master's thesis

on topic

Towards Automated Symbolic Dynamic Programming

by

Olga Skvortsova
Born on 25th October 1979 in Moscow

Technische Universität Dresden
Department of Computer Science

submitted on 14th April 2003

Supervisors: Prof. Steffen Hölldobler,
Dr. Axel Großmann

Abstract

Recent symbolic dynamic programming approaches in the situation as well as in the fluent calculus rely on the normalization of systems over abstract states. Such normalization is done by hand so far. This work presents an approach for mechanizing the normalization process involved in the symbolic dynamic programming framework within the fluent calculus (SDP&FC). Within the approach the normalization algorithms for two classes of systems: systems of *C-states* and systems of *CN-states*, are developed. The normalization algorithms transform the systems of abstract states into their normal forms, viz., into the systems that are free of redundant abstract states. The subsumption relation, a core of the normalization algorithms, that enables to detect which abstract state is redundant is devised. The presented normalization algorithms are proven to be correct, complete, and to terminate delivering a system in normal form. The (worst-case) complexity analysis shows that the normalization algorithms are NP-hard. The Prolog-implementation of the normalization algorithms is designed. In effect, our normalization algorithms enable to shrink the abstract state space by removing the redundant abstract states. This allows the computational requirements of the SDP&FC algorithms, e.g., first-order value iteration algorithm, to be considerably improved.

Keywords: Symbolic dynamic programming, fluent calculus, first-order value iteration algorithm.

Contents

Chapter 1

Introduction

Planning using decision-theoretic notions, or *decision-theoretic planning (DTP)*, has been recently addressed by many scientists in artificial intelligence (AI) [5, 14, 29, 12]. DTP is an attractive extension of the classical AI planning approach, because it allows to model problems in which actions are stochastic, an agent has an incomplete knowledge about the world, and where the goal state may be partially defined. Roughly, the aim of DTP is to construct plans, or policies, that have high expected utility.

A large class of planning problems can be modelled as Markov decision processes (MDPs). The recent research in DTP has adopted the MDP framework as an underlying model [3, 6, 7], allowing the adaptation of existing algorithms for solving MDPs to be applied for planning problems. In this case, *dynamic programming (DP)* methods can be applied for decision-theoretic planning. The term *dynamic programming* refers to a collection of algorithms that can be used to compute optimal plans given a model of the environment as an MDP.

Classical DP algorithms rely on the explicit state enumeration.[1] Consequently, these algorithms do not scale up to large state spaces. In other words, the requirement of explicit state enumeration dramatically limits the use of DP algorithms for solving complex tasks with large state spaces. This can be problematic in AI planning since state spaces grow exponentially with the number of domain individuals.

Recently, in [8], an approach that we refer to as of *symbolic dynamic programming within the situation calculus*, SDP&SC, for short, has been proposed. SDP&SC addresses the above scalability problem by giving *the situation calculus* [27, 28] formalization of DP.

As an alternative specification language the fluent calculus (FC) [15, 17] can be chosen and, as it was shown in [13], the FC formalization of the dynamic programming paradigm can be developed. Thus, the approach called *symbolic dynamic programming within the fluent calculus*, SDP&FC, for short, has been designed.

[1] We introduce here the notion of classical DP algorithms in order to distinguish these algorithms from symbolic DP algorithms [8, 13] that do not rely on the explicit state enumeration.

The symbolic dynamic programming within the FC is an approach for computing an optimal policy, or plan, within the FC. Based on an idea initially presented in [8], the optimal value function and a policy along with the major components of MDPs are logically represented.

The technique produces a logical description of the optimal value function and a policy by minimally partitioning the state space into partitions, called *abstract states*, that are described symbolically. Consequently, the symbolic dynamic programming approach within the FC does not require to enumerate the state or action spaces, thereby solving the scalability problem of the classical DP methods.[2]

More precisely, the symbolic DP framework proposes the logical description of the classical value iteration algorithm (VIA) [4], an instance of a classical DP method, by representing the stochastic actions, the value functions, and the optimality criterion for choosing the optimal value function within FC. The symbolic version of the classical VIA is called *first-order VIA*.

The operation, called *decision-theoretic regression (DTR)*, underlying the first-order VIA, is employed to partition the state space into abstract states and to compute the value function for each abstract state. A system of abstract states comprises the abstract states together with the respective value functions. One should observe that in the first-order VIA the value functions are computed for partitions, i.e., abstract states, rather than for states as in the classical VIA, thereby addressing the scalability problem of classical DP.

The systems obtained as a result of the one-step DTR may contain redundancies. If no normalization is performed, such redundancies are freely propagated to the next steps of DTR, thereby considerably increasing the computational effort of the first-order VIA. In order to simplify the iteration steps of the first-order VIA and lighten its computational requirements, we aim at developing an approach underlying the normalization process that is involved in DTR. Such normalization is done by hand so far.

Depending on the specification of a goal state space, preconditions and effects of actions, the abstract states may have different logical descriptions. So, every abstract state belongs to some class of abstract states. Therefore, first of all, various classes of abstract states are to be specified within FC. For each class of abstract states a relation that allows to determine which abstract state is redundant will be defined. Such a relation is supposed to be the core of the rewrite rules that should remove the redundant abstract states from a system.

Thus, the goal of this work is to mechanize the normalization process of the systems of abstract states. A normalization algorithm that delivers a system in normal form, i.e., a system that does not contain redundancies, will be devised. An addition, a Prolog-implementation of this normalization algorithm will be designed.

[2] We believe, however, that it is possible to develop a worst case example that would require to consider all states as in classical DP algorithms, e.g., in one presented in Section 2.3.1. But we claim that such examples hardly ever arise in practice.

The master's thesis is organized as follows.

Chapter 2 presents the symbolic dynamic programming approach within the FC and initiates a background and importance of the normalization process. In addition, a brief introduction into the theory of multisets underlying the fluent calculus is given.

Chapter 3 is divided into two similar sections, corresponding to two classes of abstract states: *C-states* and *CN-states*. Section 3.1 devises a normalization algorithm for *C-states* by defining the language of *C-states*, introducing the subsumption relation on them, and specifying the simplification rules. The chapter is closed by elaborating on the complexity results and proving such important properties of the normalization algorithm as correctness and termination. In Section 3.2, the language of *CN-states* is developed and the similar issues as in Section 3.1 are elaborated upon.

In Chapter 4, we discuss the crucial aspects of the implementation of the normalization algorithms developed in Chapter 3. Moreover, the preliminary analysis of the computational behavior of the implementation is performed.

Chapter 5 is dedicated to the related work.

Finally, in Chapter 6 we draw conclusions and propose the directions of future work.

Appendices A and B list the complete code of the implementation and some empirical results, respectively.

Acknowledgments First of all, I benefited from the comments and help of many people and it seems impossible to list them without forgetting the one or the other. I thank all of them and want them to know that I highly appreciate their accompany.

I am particular grateful to Steffen Hölldobler for inspiring and guiding the work presented in this thesis. I am indebted to him for attracting me with such interesting and promising approach as symbolic dynamic programming and helping to get acquainted with it. Most significantly, he has put constant pressure on me to write this master's thesis.

I want to thank Axel Großmann for helpful suggestions and stimulating discussions. He has introduced me with the main ideas of the reinforcement learning and has not let to be lost in the labyrinth of the dynamic programming algorithms.

Thanks to the secretary of our "Computational Logic" Programme, Sylvia Epp for being always ready to handle bureaucratic problems.

Moreover, I would like to acknowledge the support by a scholarship provided by German Academic Exchange Office (DAAD) for joining the "Computational Logic" Programme at the Dresden University of Technology.

Most importantly, is the love, support and understanding given to me by my mother and my husband Eldar.

Chapter 2

Problem Definition

The symbolic dynamic programming approach within FC relies on the version of the fluent calculus [15] that uses the data structure of multisets to describe states. Thus, before describing the fluent calculus we pay attention to the notion of multisets, main operations and relations on them, and some of their properties.

2.1 Multisets

Informally, the multisets are sets in which each element can occur more than once. Formally, let $\dot{\emptyset}$ denote the empty multiset and let the parentheses $\dot{\{}$ and $\dot{\}}$ be used to enclose the elements of a multiset. Analogously to the case of sets, the following operations and relations are defined. Let $\mathcal{M}$, $\mathcal{M}_1$, and $\mathcal{M}_2$ be finite multisets. It is vital to define the notion of an occurrence of an element in a multiset. We write $X \in_k \mathcal{M}$ to express that an element X occurs precisely k-times in $\mathcal{M}$, $k \geq 0$. We write $X \in \mathcal{M}$ iff $\exists k.\ X \in_k \mathcal{M}, k > 0$. We will use the standard notion $X \notin \mathcal{M}$ instead of $X \in_0 \mathcal{M}$.

For instance, if $\mathcal{M}$ is the multiset

$$\dot{\{}a, b, c, a, b, a\dot{\}}\ ,$$

then $a \in_3 \mathcal{M}$, $b \in_2 \mathcal{M}$, $c \in_1 \mathcal{M}$ and $d \in_0 \mathcal{M}$.

Definition 1 *The two multisets $\mathcal{M}_1$ and $\mathcal{M}_2$ are **equal**, written $\mathcal{M}_1 \doteq \mathcal{M}_2$, iff for all X we find that $X \in_k \mathcal{M}_1$ iff $X \in_k \mathcal{M}_2$.*

For instance,

$$\dot{\{}a, b, a\dot{\}} \doteq \dot{\{}a, a, b\dot{\}}\ .$$

Definition 2 *An X is a member of the **union** of two multisets $\mathcal{M}_1$, $\mathcal{M}_2$, written $X \in_m \mathcal{M}_1 \dot{\cup} \mathcal{M}_2$, iff there exist $k, l \geq 0$ such that $X \in_k \mathcal{M}_1$, $X \in_l \mathcal{M}_2$ and $m = k + l$.*

For instance, if

$$\mathcal{M}_1 \doteq \dot{\{}a, b, c\dot{\}}$$

and

$$\mathcal{M}_2 \doteq \dot{\{}a, b, a\dot{\}}$$

then

$$\mathcal{M}_1 \dot{\cup} \mathcal{M}_2 \doteq \dot{\{}a, b, c, a, b, a\dot{\}}\ .$$

Definition 3 *An X is a member of the* ***difference*** *of two multisets $\mathcal{M}_1$, $\mathcal{M}_2$, written $X \in_m \mathcal{M}_1 \dot{\setminus} \mathcal{M}_2$, iff there exist $k, l \geq 0$ such that either $X \in_k \mathcal{M}_1$, $X \in_l \mathcal{M}_2$, $k > l$ and $m = k - l$ or $X \in_k \mathcal{M}_1$, $X \in_l \mathcal{M}_2$, $k \leq l$ and $m = 0$.*

For instance, if $\mathcal{M}_1$, $\mathcal{M}_2$ are defined as above, then

$$\mathcal{M}_1 \dot{\setminus} \mathcal{M}_2 \doteq \dot{\{}c\dot{\}}$$

and

$$\mathcal{M}_2 \dot{\setminus} \mathcal{M}_1 \doteq \dot{\{}a\dot{\}}\ .$$

Definition 4 *An X is a member of the* ***intersection*** *of the two multisets $\mathcal{M}_1$, $\mathcal{M}_2$, written $X \in_m \mathcal{M}_1 \dot{\cap} \mathcal{M}_2$, iff there exist $k, l \geq 0$ such that $X \in_k \mathcal{M}_1$, $X \in_l \mathcal{M}_2$ and $m = min\{k, l\}$, where the function min returns a minimal element out of the set $\{k, l\}$.*

For instance, if $\mathcal{M}_1$ and $\mathcal{M}_2$ are as above, then

$$\mathcal{M}_1 \dot{\cap} \mathcal{M}_2 \doteq \dot{\{}a, b\dot{\}}\ .$$

Definition 5 *A multiset $\mathcal{M}_1$ is* ***a submultiset*** *of a multiset $\mathcal{M}_2$, written $\mathcal{M}_1 \dot{\subseteq} \mathcal{M}_2$, iff $\mathcal{M}_1 \dot{\cap} \mathcal{M}_2 \doteq \mathcal{M}_1$.*

For example,

$$\dot{\{}a, b, a\dot{\}} \dot{\subseteq} \dot{\{}a, b, c, a, b, a\dot{\}}\ .$$

There is a straightforward mapping from fluent terms to multisets of fluents and vice versa.

In order to elaborate on this we first need to give a definition of a fluent and a fluent term within the FC.

Formally, the fluent calculus is a sorted calculus with pre-defined sorts ACTION, FLUENT, and STATE[1] with the constraint that the sort FLUENT is a subsort of the sort STATE. The user is allowed to define his/her sorts too.

Fluents are terms of sort FLUENT. Along with the variables for the pre-defined sorts and the variables for the user-defined sorts each function symbol

[1] Only some pre-defined sorts are listed here. The complete description of the FC will be provided in Section 2.2.

in the language should be sorted as well. E.g., the pre-defined function symbols for states are sorted as:

$$
\begin{aligned}
&1 : \textsc{state} \\
&\circ : \textsc{state} \times \textsc{state} \to \textsc{state}
\end{aligned}
$$

The binary function symbol $\circ$ is associative, commutative, and admits a unit element 1, but is not idempotent.[2] In other words, the function symbol $\circ$ satisfies the associativity, commutativity, and unit element axioms, called $AC1$-axioms, that are foundational axioms of the fluent calculus together with the axioms of equality.

Fluent terms are defined inductively as follows.

Definition 6 *A **fluent term** is defined as*

- *1 is a fluent term;*
- *each fluent is a fluent term;*
- *if S and T are fluent terms then $S \circ T$ is a fluent term.*

A fluent term 1 is often called an empty fluent term.

We can now come back to a mapping from fluent terms to multisets of fluents that is denoted by $\cdot^I$ and defined as follows.

$$
T^I = \begin{cases} \dot{\emptyset} & \text{if } T = 1 \\ \dot{\{}T\dot{\}} & \text{if } T \text{ is a fluent} \\ U^I \dot{\cup} V^I & \text{if } T = U \circ V \end{cases}
$$

The inverse mapping $\cdot^{-I}$ from multisets of fluents to fluent terms exists and is defined as

$$
\mathcal{M}^{-I} = \begin{cases} 1 & \text{if } \mathcal{M} \doteq \dot{\emptyset} \\ S \circ \mathcal{N}^{-I} & \text{if } \mathcal{M} \doteq \dot{\{}S\dot{\}} \dot{\cup} \mathcal{N} \end{cases}
$$

It is known that for a fluent term T and a multiset $\mathcal{M}$ of fluents the following equations hold.

$$
T =_{AC1} (T^I)^{-I} \tag{2.1}
$$

$$
\mathcal{M} \doteq (\mathcal{M}^{-I})^I \tag{2.2}
$$

Next, we will define the size of a multiset $\mathcal{M}$.

Definition 7 *The **size** of a multiset $\mathcal{M}$, written $|\mathcal{M}|$, is defined as*

1. *If $\mathcal{M} \doteq \dot{\emptyset}$ then $|\mathcal{M}| = 0$*
2. *If $|\mathcal{M}| = n$ then $|\mathcal{M} \dot{\cup} \dot{\{}X\dot{\}}| = n + 1$.*

[2]Since we refer to states as for the multisets of fluents, we must disallow the idempotency property here.

An application of a substitution to a multiset of fluents means that this substitution is applied to all elements of a multiset as it is stated in Definition 8.

Definition 8 *Let $\mathcal{M} = \dot{\{}T_1, \ldots, T_n\dot{\}}$ be a multiset of fluents and σ a substitution. Then the application of σ to $\mathcal{M}$ is defined as*

$$\mathcal{M}\ \sigma = \dot{\{}T_1\ \sigma, \ldots, T_n\ \sigma\dot{\}}.$$

For instance, the application of the substitution $\sigma = \{X \mapsto b, Y \mapsto c\}$ to the fluent term $F = f(a, X) \circ g(X, Y)$, that represents the multiset $\mathcal{M} = \dot{\{}f(a, X), g(X, Y)\dot{\}}$, results in the fluent term $F' = f(a, b) \circ g(b, c)$ that represents the multiset $\mathcal{M}' = \dot{\{}f(a, b), g(b, c)\dot{\}}$.

Definition 9 ***An AC1-matching problem*** *[32] consists of a fluent term $T_1 \circ \ldots \circ T_n$, a fluent term $S_1 \circ \ldots \circ S_m$, and a variable X not occurring in T_i, $1 \le i \le n$, and is the question of whether there exists a substitution σ such that $(T_1 \circ \ldots \circ T_n \circ X)\sigma =_{AC1} S_1 \circ \ldots \circ S_m$.*

If a fluent term $S_1 \circ \ldots \circ S_m$ is ground then an $AC1$-matching problem is said to be ground. Otherwise, it is called a non-ground $AC1$-matching problem.

Definition 10 ***A submultiset matching problem*** *consists of two multisets S and T and is the question of whether there exists a substitution σ such that $S\sigma \dot{\subseteq} T$.*

Please note that if a substitution σ is a solution for an $AC1$-matching problem consisting of the fluent terms T, S and a variable X then a substitution $\sigma|_{\textsc{Var}(T)}$ is a solution for the submultiset matching problem consisting of T^I and S^I, and vice versa, where $\sigma|_{\textsc{Var}(T)}$ is a restriction of a substitution σ to a set of variables in a fluent term T, i.e., $\textsc{Var}(T)$.

Moreover, we find that in this case

$$(X\sigma)^I \doteq S^I \dot{\setminus} (T\sigma)^I\ .$$

For instance, the substitution $\sigma = \{R \mapsto a, X \mapsto c\}$ is a solution for the $AC1$-matching problem

$$\exists\sigma.\ (T \circ X)\sigma =_{AC1} S$$

consisting of the fluent terms $T = f(R, b)$, $S = f(a, b) \circ c$, and a variable X.

Moreover, we find that the submultiset matching problem of whether there exists a substitution θ such that

$$\dot{\{}f(R, b)\theta\dot{\}} \dot{\subseteq} \dot{\{}f(a, b), c\dot{\}}\ .$$

has a solution $\theta = \sigma|_{\textsc{Var}(T)} = \{R \mapsto a\}$.

2.2 The Fluent Calculus

The fluent calculus, or FC for short, much like the situation calculus, provides a methodology for specifying and reasoning about states, actions and causality. It was originally set up as a first-order logic program with equality using SLDE- or SLDENF-resolution as sole inference rule [15, 17]. It is based on the idea to consider the states as multisets of fluents and to represent such states on the term level. Such representation is done with the help of a binary function symbol $\circ$, which is associative, commutative, and has a constant 1 as unit element but is not idempotent. The fluent calculus specification given in this paper is similar to [16].

Formally, the fluent calculus is a sorted calculus with sorts ACTION, LIST-ACTION, FLUENT, and STATE with the constraint that the sort FLUENT is a sub-sort of the sort STATE. Additionally to the predefined sorts, the user can specify his/her own scenario-dependent sorts $\mathcal{S}_u$.

Given a finite, non-empty set $\mathcal{S}$ of sorts, we assume countably many variables for each sort $s \in \mathcal{S}$. The set of variables $\mathcal{V}$ is defined as the union of the following disjoint sets:

$$\mathcal{V} = \mathcal{V}_A \cup \mathcal{V}_L \cup \mathcal{V}_{Fl} \cup \mathcal{V}_{St} \cup \bigcup_{i=1}^{k} \mathcal{V}_{\mathcal{S}_i},$$

where $\mathcal{V}_A$, $\mathcal{V}_L$, $\mathcal{V}_{Fl}$, $\mathcal{V}_{St}$ are sets of variables for the pre-defined sorts and $\mathcal{V}_{\mathcal{S}_1} \ldots \mathcal{V}_{\mathcal{S}_k}$ are sets of variables for the user-defined sorts $\mathcal{S}_1 \ldots \mathcal{S}_k$, respectively. Please note, that $\mathcal{S}_u = \bigcup_{i=1}^{k} \mathcal{S}_i$.

Each function symbol of arity n is assigned a structure $s_1 \times \ldots \times s_n \rightarrow s$. Analogously, each predicate symbol of arity n is assigned a structure $s_1 \times \ldots \times s_n$. In other words, all function and predicate symbols should be sorted. The predefined function symbols are sorted as:

$[\,]$: LIST-ACTION	1 : STATE
$\mid$: ACTION $\times$ LIST-ACTION $\rightarrow$ LIST-ACTION	$\circ$: STATE $\times$ STATE $\rightarrow$ STATE

with the understanding of the sort LIST-ACTION as of a list of actions.

As an example consider the scenario where the racing cars are loaded onto and unloaded from the trucks and trucks are driven between the cities. This example scenario will be used throughout the paper.

Example 1 For describing this scenario we would need to define the sorts CARS, TRUCKS, and CITIES besides the predefined sorts. To specify the states we would need the following four fluents with the intended meaning.

rin : CARS $\times$ CITIES $\rightarrow$ FLUENT	racing car R is in city C
tin : TRUCKS $\times$ CITIES $\rightarrow$ FLUENT	truck T is in city C
on : CARS $\times$ TRUCKS $\rightarrow$ FLUENT	racing car R is on truck T
rain : FLUENT	it is raining

Using the $\circ$ operation, the state with informal semantics 'the racing car ferrari is on the truck honda which is in the city Monte-Carlo' can be formalized by the fluent term:

$$on\,(ferrari\,, honda\,) \circ tin\,(honda\,, monteCarlo\,)\ .$$

To formalize the actions we will have to define the following functions in the same manner as for the fluents:

$unload$: CARS $\times$ TRUCKS $\rightarrow$ ACTION	a racing car is unloaded from a truck
$load$: CARS $\times$ TRUCKS $\rightarrow$ ACTION	a racing car is loaded onto a truck
$drive$: TRUCKS $\times$ CITIES $\rightarrow$ ACTION	a truck is driven in a city

$\square$

So far, we have learned that the variables are sorted, too. Additionally, one should note that the variables may be instantiated by the terms of the right sort only. Recalling the Example 1 and assuming that *ferrari* and *mclaren* are the only terms of the sort CARS, in the fluent $on(R, honda)$ only the terms *ferrari* or *mclaren* may be substituted for the variable R. From this we can conclude that the term $on(honda, honda)$ is not well-sorted.

In general, user-defined functions from any of the sorts from $\mathcal{S}_u$ into the sorts FLUENT and ACTION, describing the fluents and the actions, correspondingly, and the functions over $\mathcal{S}_u$ have to be taken into account. Therefore, the set of function symbols $\mathcal{F}$ is defined as the union of the following disjoint sets:

$$\mathcal{F} = \mathcal{F}_A \cup \mathcal{F}_L \cup \mathcal{F}_{Fl} \cup \mathcal{F}_{St} \cup \bigcup_{i=1}^{k} \mathcal{F}_{\mathcal{S}_i},$$

where $\mathcal{F}_A$ is a set of function symbols denoting action names, $\mathcal{F}_L = \{[], |\}$ is a set of function symbols denoting lists of actions, $\mathcal{F}_{Fl}$ is a set of function symbols denoting fluent names, $\mathcal{F}_{St} = \{1, \circ\}$ is a set of function symbols denoting states, $\mathcal{F}_{\mathcal{S}_i}$ is a set of function symbols for functions over a user-defined sort $\mathcal{S}_i$.

Example 1 (continued) For our example we will have the following function symbols. $\mathcal{F}_A = \{unload, load, drive\}$, $\mathcal{F}_L = \{[], |\}$, $\mathcal{F}_{Fl} = \{rin, tin, on, rain\}$, $\mathcal{F}_{St} = \{1, \circ\}$, and $\mathcal{F}_{\mathcal{S}_i} = \{\}$ for all $i = 1 \ldots k$, since we have no functions defined over any of sorts $\mathcal{S}_i$.

$\square$

Coming back to the predicate symbols, they are divided onto the predefined predicate symbols, namely $=$, *action*, *causes* and user-defined predicate symbols over $\mathcal{S}_u$. The predefined predicate symbols should be sorted as:

$action$: STATE $\times$ ACTION $\times$ STATE
$causes$: STATE $\times$ LIST-ACTION $\times$ STATE.

The preconditions and effects of the actions are specified with the help of the predicate $action/3$, where the arguments encode the preconditions, the name, and the effects of an action. For every action, we will have the predicate *action* of the form $action(Pre, A, E)$ where Pre and E are of the sort STATE and A is of the sort ACTION.

Example 1 (continued) Consider the unloading of a car R from a truck T: If a car R is loaded on T and a truck T is in a certain city C, then after unloading the truck, i.e. performing the action $\mathit{unload}(R,T)$, both, the car and the truck, will be in the city C:

$$\mathit{action}(\mathit{on}(R,T) \circ \mathit{tin}(T,C),\ \mathit{unload}(R,T),\ \mathit{rin}(R,C) \circ \mathit{tin}(T,C)).$$[3]

Causality is represented using a predicate symbol $\mathit{causes}/3$, whose arguments denote a state, a list of actions, and a successor state, respectively. Intuitively, an atom such as $\mathit{causes}(Z,P,Z')$ is to be understood as: The execution of a plan P transforms a state Z into a state Z'. The predicate $\mathit{causes}/3$ is defined recursively on the structure of plans, which are lists of actions:

$$\begin{array}{l}
\mathit{causes}(Z,[\,],Z') \leftarrow Z =_{AC1} Z'. \\
\mathit{causes}(Z,[A|P],Z') \leftarrow \mathit{action}(\mathit{Pre},A,E) \wedge \\
\qquad\qquad\qquad\qquad\qquad Z' =_{AC1} Z'' \circ E \wedge \\
\qquad\qquad\qquad\qquad\qquad \mathit{causes}(Z,P,Z'' \circ \mathit{Pre}).
\end{array}$$

The below-given macro, $\mathit{holds}/2$-macro, is particularly handy, when one needs to express that a particular fluent holds at a state. E.g., the fact that the racing car *ferrari* is on the truck *volvo* in a state Z can be expressed by

$$\mathit{holds}(\mathit{on}(\mathit{ferrari},\mathit{volvo}),Z)\ .$$

The *holds*-macro is defined by:

$$\mathit{holds}(F,Z) \stackrel{def}{=} \exists X.\ Z =_{AC1} F \circ X\ ,$$

where Z is a state, F and X are fluent terms.

Moreover, the *holds*-macro is convenient to express that a certain fluent does not hold in a state which is partially known. E.g., if we only know that the racing car *ferrari* is on the truck *volvo* and that the *volvo* is not in the city *monteCarlo*, and other fluents may or may not hold, then we can write:

$$\mathit{holds}(\mathit{on}(\mathit{ferrari},\mathit{volvo}),Z)\ \wedge\ \neg \mathit{holds}(\mathit{tin}(\mathit{volvo},\mathit{monteCarlo}),Z)\ .$$

On the other hand, this macro becomes quite cumbersome to express that some fluents hold in a state and all other fluents do not. In particular, to

[3]One may argue that $\mathit{tin}(T,C)$ is a precondition of this action which remains unchanged when the action is executed, and is not an effect. For technical reasons, however, such preconditions must also be specified as effects in the original fluent calculus. This was one of the reasons for revising the fluent calculus. On the other hand, a user will ideally not see a fluent calculus specification but rather will specify actions in a much higher language like, for example, a restricted subset of English. Sentences in such a language will be compiled into fluent calculus expressions. We believe that in real-world applications there are only few preconditions which remain unchanged when an action is executed and, consequently, that the technical necessity to repeat these preconditions as effects is not a considerable drawback.

formalize that the fluents $on(ferrari, volvo)$ and $tin(volvo, monteCarlo)$ hold in a state Z and all other fluents do not, would take us to write an expression of the following kind:

$$\begin{array}{l} holds(on(ferrari, volvo), Z) \wedge \\ holds(tin(volvo, monteCarlo), Z) \wedge \\ \forall F.\ (F \neq on(ferrari, volvo)\ \wedge F \neq tin(volvo, monteCarlo)) \rightarrow \\ \neg holds(F, Z), \end{array}$$

where $\neg$ is classical negation.

The Propositions 1 and 2 below state the important properties of the *holds*-macro.

Proposition 1 *Let F be a fluent term. Then $\forall F.\ holds(F, F)$.*

Proof According to the definition of the *holds*-macro we have:

$$holds(F, F) \stackrel{def}{=} \exists X.\ F =_{AC1} F \circ X.$$

Taking X=1, we obtain an expression $F =_{AC1} F \circ 1$ that is valid under the equational theory $AC1$.

□

Proposition 2 *Let Z be a state, F and G are fluent terms. Then*

$$holds(F \circ G, Z) \equiv holds(F, Z) \wedge holds(G, Z)\ . \tag{2.3}$$

To put in words, Proposition 2 states that if two fluent terms F and G hold simultaneously in a state Z, i.e., $holds(F \circ G, Z)$, then they both hold separately in Z, i.e., $holds(F, Z)\ \wedge\ holds(G, Z)$.

Proof Applying the definition of the *holds*-macro to the left-hand side of the Equation (2.3) we obtain:

$$holds(F \circ G, Z) \stackrel{def}{=} \exists X.\ Z =_{AC1} F \circ G \circ X. \tag{2.4}$$

Next, we transform the right hand side of (2.4) and obtain the formula:

$$\exists X_1.\ Z =_{AC1} F \circ X_1\ \wedge\ \exists X_2.\ Z =_{AC1} G \circ X_2 \tag{2.5}$$

where $X_1 = G \circ X$ and $X_2 = F \circ X$.

Applying the definition of the *holds*-macro to Equation (2.5) from right to the left, we obtain the Equation (2.6)

$$holds(F, Z)\ \wedge\ holds(G, Z)\ . \tag{2.6}$$

Finally, we get a desired result

$$holds(F \circ G, Z) \equiv holds(F, Z) \wedge holds(G, Z)\ .$$

$\square$

One should observe that for the case with negation,

$$\neg holds(F \circ G, Z)$$

and

$$\neg holds(F, Z) \wedge \neg holds(G, Z)$$

are **not** semantically equivalent.

To illustrate this observation please recall our example with racing cars, trucks and cities. Consider the two *holds*-macro definitions

$$\neg holds(on(ferrari, honda) \circ tin(honda, paris), Z) \tag{2.7}$$

$$\neg holds(on(ferrari, honda), Z) \wedge \neg holds(tin(honda, paris), Z). \tag{2.8}$$

Equations (2.7) and (2.8) are not semantically equivalent because, e.g., the state $Z = on(ferrari, honda)$ satisfies Equation (2.7) but does not satisfy Equation (2.8).

2.3 Symbolic Dynamic Programming within the Fluent Calculus

Symbolic DP approach within FC recently presented in [13] is aimed at combining the symbolic reasoning and the reasoning under uncertainty paradigms. SDP&FC addresses the scalability problem of classical DP by partitioning the state space into partitions, called abstract states, that are described logically by using notions of FC.

The classical VIA [4], an instance of a classical DP method, is taken for further consideration. It is an algorithm for computing an optimal policy in the environment that is formalized as a Markov decision process.

2.3.1 Markov Decision Processes and Classical Value Iteration

A Markov decision process is a tuple $(\mathcal{Z}, \mathcal{A}, \mathcal{P}, \mathcal{R})$, where $\mathcal{Z}$ is a finite set of states, $\mathcal{A}$ is a finite set of actions, $\mathcal{P} : \mathcal{Z} \times \mathcal{A} \times \mathcal{Z} \to [0, 1]$ is a state transition function, and $\mathcal{R}$ is a reward function. $\mathcal{P}(z, a, z')$ denotes the probability with which a state z' is reached by executing an action a in a state z. A reward function $\mathcal{R} : \mathcal{Z} \to \Re$ is a real-valued function associating with each state z its immediate utility $\mathcal{R}(z)$. Throughout the paper, we consider only fully observable MDPs, i.e., an agent fully observes the state it is currently in, as well as discrete state and action spaces.

$n := 0$.
Specify $\varepsilon > 0$.
Initialize $V_0(z)$ arbitrarily, e.g., $V_0(z) := \mathcal{R}(z)$, for all $z \in \mathcal{Z}$.
`loop`
  $n := n + 1$.
  `loop` for all $z \in \mathcal{Z}$.
    `loop` for all $a \in \mathcal{A}$.
      $Q_n(a, z) := \mathcal{R}(z) + \gamma \sum_{z' \in \mathcal{Z}} \mathcal{P}(z, a, z') V_{n-1}(z')$.
    `end loop`.
    $V_n(z) := \max_{a \in \mathcal{A}} Q_n(a, z)$.
  `end loop`.
`until` $|V_n(z) - V_{n-1}(z)| < \varepsilon$ for all $z \in \mathcal{Z}$.

Figure 2.1: The classical VIA.

A sequential decision problem consists of an MDP and is the problem of finding a policy $\pi : \mathcal{Z} \to \mathcal{A}$ that maximizes the total discounted and accumulated reward received when executing the policy. Such a policy is called optimal. The value of a state z with respect to a policy π is defined recursively as:

$$V_\pi(z) = \mathcal{R}(z) + \gamma \sum_{z' \in \mathcal{Z}} \mathcal{P}(z, \pi(z), z') V_\pi(z'),$$

where $0 \leq \gamma < 1$ is a discount factor. Finding an optimal policy corresponds to finding an optimal value function V^*, i.e. a value function which asserts that the value of a state z is the expected instantaneous reward plus the expected discounted value of the next state executing the best available action:

$$V^*(z) = \max_{a \in \mathcal{A}} \left\{ \mathcal{R}(z) + \gamma \sum_{z' \in \mathcal{Z}} \mathcal{P}(z, a, z') V^*(z') \right\}.$$

The classical VIA is an iterative approximation algorithm for constructing optimal policies. The algorithm is depicted in Figure 2.1. The function $Q_n(a, z)$ used in this algorithm is called n-stage-to-go Q-function and denotes the expected value of executing action a in state z with n steps to go and acting optimally thereafter. If the optimal value function is computed, then an optimal policy π^* executed in state z can be obtained as:

$$\pi^*(z) = \arg\max_{a \in \mathcal{A}} \left\{ \mathcal{R}(z) + \gamma \sum_{z' \in \mathcal{Z}} \mathcal{P}(z, a, z') V^*(z') \right\}.$$

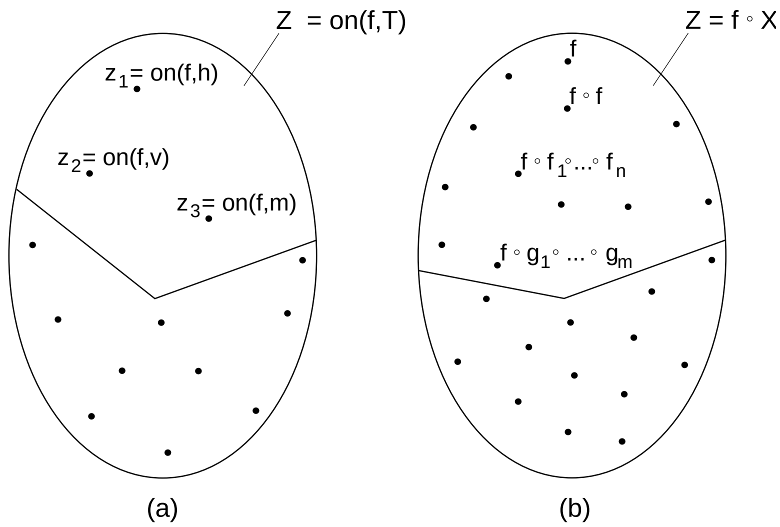

Figure 2.2: Partitioning of the state space into abstract states. (a): an abstract state $Z = on(f, T)$ represents a set of states $\{on(f, h), on(f, v), on(f, m)\}$. (b): an abstract state $Z = f \circ X$ represents a set of states that contains such states as f, $f \circ f$, $f \circ f_1 \circ \ldots \circ f_n$, $f \circ g_1 \circ \ldots \circ g_m$ and other states, where the fluent f is present and holds and other fluents, e.g., $f_1, \ldots, f_n, g_1, \ldots, g_m$, may hold as well.

2.3.2 Abstract States

In the symbolic DP, the fluent calculus formalization of the dynamic programming paradigm is proposed. We develop a symbolic description of the classical VIA, called first-order VIA, by representing stochastic actions, value functions, and the optimality criterion for choosing an optimal value function within the fluent calculus. Our approach performs a minimal partition of the state space and associates to each obtained partition the utility value.

Such partitions of the state space are called *abstract states*. The notion of an abstract state is a fundamental notion in the first-order VIA that is presented in the next section.

Intuitively, an abstract state represents a set of states that have similar symbolic description. For instance, consider a state space that is depicted in Figure 2.2. Among other states the state space contains the states z_1, z_2, z_3 described as fluents $z_1 = on(f, h)$, $z_2 = on(f, v)$ and $z_3 = on(f, m)$, where the fluents $on(f, h)$, $on(f, v)$ and $on(f, m)$ have the meaning that the racing car ferrari (f) is on the truck honda (h), volvo (v) or mercedes (m), respectively. Having assumed that h, v and m are the only trucks available in the domain, we may combine the states z_1, z_2 and z_3 into one abstract state $Z = on(f, T)$, where T refers to some truck the racing car ferrari is on. Thus, the abstract state Z represents the set of states z_1, z_2 and z_3.

One should be aware that from now on, the variables $Z, Z_1, Z_2, \ldots, Z_n$ will denote abstract states.

The second example shown in Figure 2.2 illustrates another possibility of partitioning the state space into abstract states. The states, in which the fluent f holds and other fluents may hold too, can be combined in the abstract state described as $f \circ X$. Due to the presence of a variable X an abstract state $f \circ X$ represents a set of states that contains such states as f, $f \circ f$, $f \circ f_1 \circ \ldots \circ f_n$, $f \circ g_1 \circ \ldots \circ g_m$ and other states where the fluent f is present and holds and other fluents, e.g., $f_1, \ldots, f_n, g_1, \ldots, g_m$, may hold as well. Moreover, since in FC the states are multisets of fluents, that is, a fluent may appear more than once in a state, the set of states represented by an abstract state $f \circ X$ is infinite.

By proposing the partitioning of the state space into abstract states, we address the scalability problem of the classical value iteration algorithm. Namely, instead of computing the optimal value function for all states out of a state space we operate on partitions of a state space, i.e., abstract states. Indeed, in the example above, the optimal value function will be calculated only for the abstract state Z, not for all three states z_1, z_2 and z_3.

2.3.3 First-Order Value Iteration Algorithm

The first-order VIA performs a minimal partition of a state space into abstract states and associates to each abstract state its utility value. The algorithm presented in Figure 2.3 is a detailed version of the algorithm designed in [13].

```
Input: (G, A, prob, reward), where
          G goal state space, consists of a finite number of abstract states
          A finite set of actions
          prob : A' × A × G → [0,1] is a probability function,
          A' := {a' | choice(a, a') for all a ∈ A} and A' is finite
          reward : G → ℜ is a real-valued reward function
Output: Z := {⟨Z, v(Z)⟩} finite set of abstract states Z
          together with the value of the optimal value function v(Z)
n := 0
Specify ε > 0
Initialize v_0(Z') arbitrarily, e.g., v_0(Z') := reward(Z'), for all Z' ∈ G
Z := {⟨Z', v_0(Z')⟩ | Z' ∈ G}
loop
    n := n + 1
    loop for all a ∈ A
      loop for all Z' such that ⟨Z', v_{n-1}(Z')⟩ ∈ Z
        loop for all a_j ∈ {a' | choice(a, a')}
          S_a^j(Z') := {⟨Z, reward(Z)⟩ | FC ⊨ causes(Z, [a_j], Z')}   (dotted braces)
        end loop
      end loop
      S_a := ⊍_{Z'∈Z} ⊍_j S_a^j(Z')
      S_a^N := normalize(S_a)
      loop for all Z such that ⟨Z, reward(Z)⟩ ∈ S_a^N
        q_n(a, Z) := reward(Z) + γ Σ_j prob(a_j, a, Z) v_{n-1}(Z'_j), where
        FC ⊨ causes(Z, [a_j], Z'_j)
      end loop
      S_a := {⟨Z, q_n(a, Z)⟩ | ⟨Z, reward(Z)⟩ ∈ S_a^N}   (dotted braces)
    end loop
    S := ⊍_{a∈A} S_a
    S^N := normalize(S)
    loop for all Z such that ⟨Z, q_n(a, Z)⟩ ∈ S^N
      v_n(Z) := max_a q_n(a, Z)
    end loop
    Z := {⟨Z, v_n(Z)⟩ | ⟨Z, q_n(a, Z)⟩ ∈ S^N}
until |v_n(Z) − v_{n-1}(Z)| < ε for all Z such that ⟨Z, v_n(Z)⟩ ∈ Z
```

Figure 2.3: The first-order value iteration algorithm.

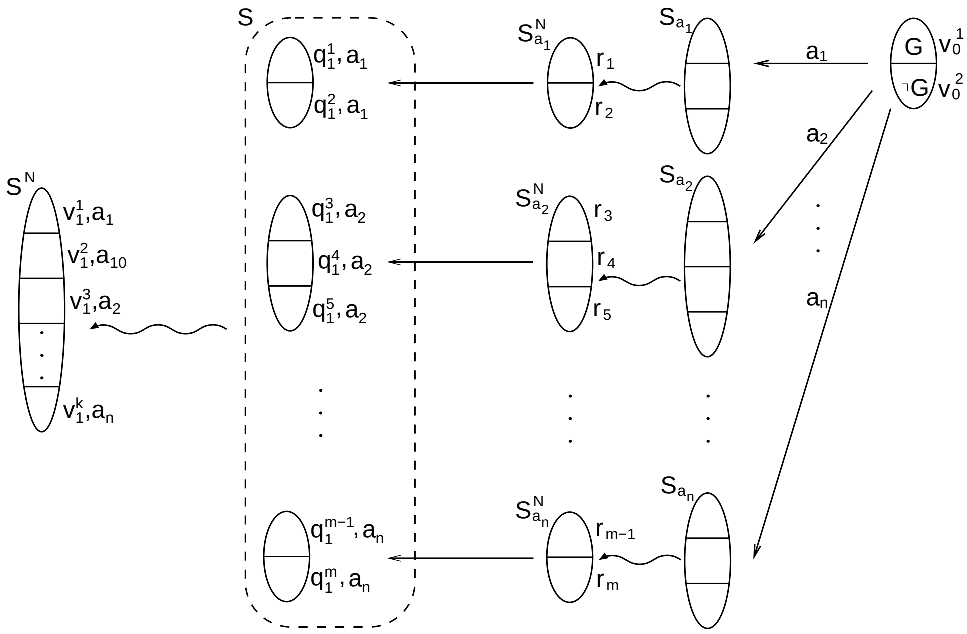

Figure 2.4: The one iteration step of the first-order value iteration algorithm.

Proposition 3 *If $\mathcal{Z}$ is finite before entering the main loop of the algorithm in Figure 2.3 then $\mathcal{Z}$ remains finite after one cycle of the loop.*

Proof Let $|\mathcal{X}|$ denote the cardinality of a set $\mathcal{X}$. Let $\mathcal{Z}_0$ and $\mathcal{Z}_1$ denote the set of abstract states $\mathcal{Z}$ before/after entering/exiting the main loop of the algorithm in Figure 2.3, respectively.

Assume, for the worst case, that the normalization procedure *normalize* had no effect with respect to the cardinalities of systems, namely $|\mathcal{S}_a^N| = |\mathcal{S}_a|$ and $|\mathcal{S}^N| = |\mathcal{S}|$. Then the upper bound of $|\mathcal{Z}_1| = |\mathcal{S}|$, where $|\mathcal{S}|$ is defined as follows

$$|\mathcal{S}| = \sum_{a \in \mathcal{A}} |\mathcal{S}_a| \leq \sum_{a \in \mathcal{A}} \sum_{Z' \in \mathcal{Z}_0} \sum_{a_j \in \mathcal{A}'} |\mathcal{S}_a^j(Z')| \ .$$

One should observe that for all $a_j \in \mathcal{A}'$ and $Z' \in \mathcal{Z}_0$ $|\mathcal{S}_a^j(Z')|$ is finite since the $AC1$-unification procedure terminates delivering a finite set of most general unifiers, for the proof see [32]. Hence, $|\mathcal{S}|$ is a finite summation of finite numeric values, which is finite. Consequently, $|\mathcal{Z}_1|$ is finite.

□

Theorem 1 *The first-order value iteration algorithm in Figure 2.3 provides indeed the optimal value function and, consequently, an optimal policy.*[4]

[4]Since the proof of this theorem is beyond the scope of our work, it will not be provided here, but rather considered as an interesting direction of the future work.

In order to give a flavor of how the first-order value iteration algorithm works, we consider its one iteration step in Figure 2.4.

The abstract states G and $\neg G$ together with the values of the value function v_0^1, v_0^2, respectively, describe the goal state space. The systems $\mathcal{S}_{a_1}, \ldots, \mathcal{S}_{a_n}$ are systems of the predecessor abstract states obtained by decision-theoretic regression of the goal state space through actions $a_1, \ldots, a_n$, respectively. The systems $\mathcal{S}_{a_1}, \mathcal{S}_{a_2}, \ldots, \mathcal{S}_{a_n}$ may contain the redundancies that have to be removed. Therefore, a normalization algorithm is executed here and delivers the normal forms of the systems $\mathcal{S}_{a_1}, \ldots, \mathcal{S}_{a_n}$, denoted as $\mathcal{S}_{a_1}^N, \ldots, \mathcal{S}_{a_n}^N$. The rewards of the abstract states in the systems $\mathcal{S}_{a_1}^N, \ldots, \mathcal{S}_{a_n}^N$ are defined as $r_1, \ldots, r_m$, respectively. Next, for every abstract state out of $\mathcal{S}_{a_1}^N, \ldots, \mathcal{S}_{a_n}^N$ its q-value is computed. Since on the next stage an optimal action is to be chosen, at this stage we associate to each abstract state an action the regression has been performed upon. The system $\mathcal{S}$ contains then the abstract states out of the systems $\mathcal{S}_{a_1}^N, \ldots, \mathcal{S}_{a_n}^N$ together with the respective q-values. Though the system $\mathcal{S}$ consists of the normalized systems, it may still contain the redundancies. Thus, a normalization algorithm is executed one more time for the system $\mathcal{S}$ as an input and delivers a system $\mathcal{S}^N$ as a result. Next, the values of the value function for all abstract states out of the system $\mathcal{S}^N$, denoted as $v_1^1, v_1^2, \ldots, v_1^k$, are computed.

We recall the example scenario given in Section 2.2 in order to illustrate at which stage one has to perform the normalization within the first-order VIA, we propose to execute the regression of the goal state space through an action $\mathit{unload}(R,T)$.

Since $\mathit{unload}(R,T)$ is a stochastic action, we need to introduce a technique to cope with stochastic actions within FC. The method used here is to decompose a stochastic action into deterministic primitives under nature's control, referred to as nature's choices. The action $\mathit{unload}(R,T)$ has two deterministic primitives. Therefore, using a pre-defined relation symbol $\mathit{choice}/2$, we obtain

$$\mathit{choice}(\mathit{unload}(R,T),A) \leftrightarrow \ (A = \mathit{unloadS}(R,T) \vee A = \mathit{unloadF}(R,T)),$$

where $\mathit{unloadS}(R,T)$ and $\mathit{unloadF}(R,T)$ define two nature's choices for action $\mathit{unload}(R,T)$, viz., that it is successfully executed or fails. Likewise, nature's choices can be defined for other actions in the example scenario.

Since a stochastic action has different outcomes that can be described using probabilities, we would like to formalize probabilities within FC as well. For each of the nature's choices a_j associated with an action a we define the probability $\mathit{prob}(a_j, a, Z)$. It denotes the probability with which one of nature's choices a_j is chosen in a state Z. The probability of the nature choice $\mathit{unloadS}(R,T)$, namely

$$\mathit{prob}(\mathit{unloadS}(R,T), \mathit{unload}(R,T), Z) = .7 \leftrightarrow \ \mathit{holds}(\mathit{rain}, Z)\ ,$$

is .7 if it is raining. Altogether, the probabilities of the unload-action are specified in Figure 2.5.

Besides the nature's choices of the unload-action and their probabilities, the conditions and effects of every nature's choice should be specified by means of

$$
\begin{array}{l}
prob\,(unloadS\,(R,T), unload\,(R,T), Z) = .7 \leftrightarrow \;\; holds\,(rain\,, Z) \\
prob\,(unloadS\,(R,T), unload\,(R,T), Z) = .9 \leftrightarrow \;\; \neg holds\,(rain\,, Z) \\
prob\,(unloadF\,(R,T), unload\,(R,T), Z) = .3 \leftrightarrow \;\; holds\,(rain\,, Z) \\
prob\,(unloadF\,(R,T), unload\,(R,T), Z) = .1 \leftrightarrow \;\; \neg holds\,(rain\,, Z)
\end{array}
$$

Figure 2.5: The probabilities of the *unload*-action.

$$
\begin{array}{l}
action\,(on\,(R,T) \circ tin\,(T,C),\;\; unloadS\,(R,T),\;\; rin\,(R,C) \circ tin\,(T,C)) \\
action\,(on\,(R,T) \circ tin\,(T,C),\;\; unloadF\,(R,T),\;\; on\,(R,T) \circ tin\,(T,C))
\end{array}
$$

Figure 2.6: The action specifications of the *unload*-action.

the predicate symbol *action*/3. For instance, the conditions and effects of the nature choice $unloadS\,(R,T)$ are specified as

$$
action\,(on\,(R,T) \circ tin\,(T,C), unloadS\,(R,T),\;\; rin\,(R,C) \circ tin\,(T,C))\ ,
$$

with the understanding that after successful unloading of a racing car R from a truck T the car R is in a city C and a truck T is in a city C too, having assumed that before execution of an action $unloadS(R,T)$ the car R has been on the truck T and this truck has been in a city C.

Altogether, the specification of the nature's choices of the *unload*-action is shown in Figure 2.6.

In the next step, we give a reward of 10 to all abstract states in which the racing car ferrari (f) is in the city Monte-Carlo (m) and 0, otherwise:

$$
\begin{array}{l}
reward\,(Z) = 10 \leftrightarrow holds\,(rin\,(f,m), Z) \\
reward\,(Z) = 0 \leftrightarrow \neg holds\,(rin\,(f,m), Z).
\end{array}
$$

The classical n-stage-to-go Q-function defined in Equation (2.9) has a FC analogue presented in Equation (2.10).

$$
Q_n(a,z) = \mathcal{R}(z) + \gamma \sum_{z' \in \mathcal{Z}} \mathcal{P}(z,a,z') V_{n-1}(z')\ . \tag{2.9}
$$

$$
q_n(a,Z) = reward\,(Z) + \gamma \sum_j \; prob\,(a_j, a, Z)\, v_{n-1}(Z'_j) \tag{2.10}
$$

with

$$
\mathcal{FC} \models causes\,(Z, [a_j], Z'_j)\ . \tag{2.11}
$$

Finally, Equation (2.12) defines the FC analogue for the optimal value function

$$
v_n(Z) = \max_{a \in \mathcal{A}} q_n(a,Z)\ . \tag{2.12}
$$

Recalling our running example, we will illustrate the regression of a goal state space through the action $\mathit{unload}(R, T)$. Let the discount factor γ be set to .9 and initialize $v_0(Z) = \mathit{reward}(Z)$ for all abstract states Z.

Following the first-order value iteration algorithm in Figure 2.3, we have to obtain the systems $\mathcal{S}_a^j = \dot{\{}\langle Z, reward(Z)\rangle | \mathcal{FC} \models causes(Z, [a_j], Z')\dot{\}}$ for both nature's choices of the action *unload*, i.e.,

$$\mathcal{S}^1_{\mathit{unload}} = \dot{\{}\langle Z, \mathit{reward}(Z)\rangle | \mathcal{FC} \models \mathit{causes}(Z, [\mathit{unloadS}(R, T)], Z')\dot{\}} \tag{2.13}$$

$$\mathcal{S}^2_{\mathit{unload}} = \dot{\{}\langle Z, \mathit{reward}(Z)\rangle | \mathcal{FC} \models \mathit{causes}(Z, [\mathit{unloadF}(R, T)], Z')\dot{\}} \ . \tag{2.14}$$

In order to obtain the logical descriptions Z of the predecessor abstract states, the regression of a goal state space has to be performed. The goal state space consists of two abstract states described with the help of *holds*-macro, namely $holds(rin(f, m), Z')$ with the informal meaning that the racing car ferrari (f) is in the city Monte-Carlo (m) in a goal abstract state Z' and $\neg holds(rin(f, m), Z')$ with the understanding that the ferrari is not in Monte-Carlo in Z'.

Consider first Equation (2.13) and the goal abstract state where the ferrari is in Monte Carlo, i.e., $\mathit{holds}(rin(f, m), Z')$. In this case, the *holds*-macro can be merged into the *causes*-statement. Thus we obtain

$$\mathcal{FC} \models \mathit{causes}(Z, [\mathit{unloadS}(R, T)], rin(f, m) \circ Z'') \ . \tag{2.15}$$

To solve (2.15), we effectively have to find a refutation for:

$$?- \mathit{causes}(Z, [\mathit{unloadS}(R, T)], rin(f, m) \circ Z'').$$

Within two steps, this goal can be reduced to:

$$\begin{aligned} ?\!-\ & rin(R, C) \circ \mathit{tin}(T, C) \circ X =_{AC1} rin(f, m) \circ Z'', \\ & \mathit{causes}(Z, [\,], \mathit{on}(R, T) \circ \mathit{tin}(T, C) \circ X). \end{aligned} \tag{2.16}$$

In the next step, the AC1-unification algorithm is called and terminates successfully with a set of most general unifiers consisting of

$$\sigma_1 = \{R \mapsto f,\ C \mapsto m,\ Z'' \mapsto \mathit{tin}(T, m) \circ X\}$$

and

$$\sigma_2 = \{X \mapsto rin(f, m) \circ Y,\ Z'' \mapsto rin(R, C) \circ \mathit{tin}(T, C) \circ Y\}.$$

Thus, we obtain the two goals

$$?- \mathit{causes}(Z, [\,], \mathit{on}(f, T) \circ \mathit{tin}(T, m) \circ X)$$

and

$$?- \mathit{causes}(Z, [\,], \mathit{on}(R, T) \circ \mathit{tin}(T, C) \circ rin(f, m) \circ Y).$$

Both goals can be refuted in one step leading to the computed answer substitutions

$$\theta_1 = \{Z \mapsto on(f,T) \circ tin(T,m) \circ X,\ R \mapsto f,\ Z'' \mapsto tin(T,m) \circ X, C \mapsto m\}$$

and

$$\theta_2 = \{Z \mapsto on(R,T) \circ tin(T,C) \circ rin(f,m) \circ Y,\ Z'' \mapsto rin(R,C) \circ tin(T,C) \circ Y\},$$

respectively. Furthermore, it can be computed that $reward(Z\theta_1) = 0$ and $reward(Z\theta_2) = 10$.

Thus, we obtain two descriptions A and B for abstract predecessor states Z

$$A \stackrel{def}{=} holds(on(f,T) \circ tin(T,m), Z) \ \wedge \neg holds(rin(f,m), Z)$$

and

$$B \stackrel{def}{=} holds(on(R,T) \circ tin(T,C) \circ rin(f,m), Z).$$

Now consider the goal abstract state where the ferrari is not in Monte Carlo, i.e., $\neg holds(rin(f,m), Z')$. In this case, we obtain

$$\mathcal{FC} \models causes(Z, [unloadS(R,T)], Z') \tag{2.17}$$

with extra condition that $\neg holds(rin(f,m), Z')$ is true.

To solve (2.17), we have to find a refutation for

$$?-\ causes(Z, [unloadS(R,T)], Z').$$

Within two steps, this goal can be reduced to:

$$\begin{aligned} ?\!-\ & rin(R,C) \circ tin(T,C) \circ X =_{AC1} Z', \\ & causes(Z, [\,], on(R,T) \circ tin(T,C) \circ X). \end{aligned} \tag{2.18}$$

In the next step, the AC1-unification algorithm is called and terminates successfully with a most general unifier

$$\sigma = \{Z' \mapsto rin(R,C) \circ tin(T,C) \circ X\}$$

Then we conclude that $\neg holds(rin(f,m), X)$ is true and $(R \neq f \ \vee \ C \neq m)$. So, the computed answer substitution is

$$\theta = \{Z \mapsto on(R,T) \circ tin(T,C) \circ X\}$$

with the condition that $\neg holds(rin(f,m), X)$ is true and $(R \neq f \ \vee \ C \neq m)$.

Since $\neg holds(rin(f,m), X)$ is true, $reward(Z\theta) = 0$. Thus, we obtain a description C of an abstract predecessor state Z

$$C \stackrel{def}{=} holds(on(R,T) \circ tin(T,C), Z) \wedge \neg holds(rin(f,m), Z) \wedge (R \neq f \vee C \neq m).$$

Next, we combine the obtained logical descriptions with the respective reward values to the system $\mathcal{S}^1_{unload}$

$$\mathcal{S}^1_{unload} = \begin{cases} \langle A, 0\rangle \\ \langle B, 10\rangle \\ \langle C, 0\rangle \ . \end{cases}$$

Likewise, consider Equation (2.14) and the goal abstract state where the ferrari is in Monte Carlo, i.e., $holds(rin(f, m), Z')$. In this case, the *holds*-macro can be merged into the *causes*-statement. We obtain

$$\mathcal{FC} \models causes(Z, [unloadF(R, T)], rin(f, m) \circ Z'') \tag{2.19}$$

To solve (2.19), we have to find a refutation for:

$$?-\ causes(Z, [unloadF(R, T)], rin(f, m) \circ Z'')\ .$$

Within two steps, this goal can be reduced to:

$$\begin{aligned} ?\!-\ & on(R, T) \circ tin(T, C) \circ X =_{AC1} rin(f, m) \circ Z'', \\ & causes(Z, [\,], on(R, T) \circ tin(T, C) \circ X)\ . \end{aligned} \tag{2.20}$$

In the next step, the AC1-unification algorithm is called and terminates successfully with a most general unifier

$$\sigma = \{X \mapsto rin(f, m) \circ Y,\ Z'' \mapsto on(R, T) \circ tin(T, C) \circ Y\}\ .$$

Thus, we obtain the goal

$$?-\ causes(Z, [\,], on(R, T) \circ tin(T, C) \circ rin(f, m) \circ Y)\ ,$$

that can be refuted in one step leading to the computed answer substitution

$$\theta = \{Z \mapsto on(R, T) \circ tin(T, C) \circ rin(f, m) \circ Y\}.$$

Since $holds(rin(f, m), Z)$ is true, $reward(Z\theta) = 10$.

Thus, we obtain the description D of an abstract predecessor state Z

$$D \stackrel{def}{=} holds(on(R, T) \circ tin(T, C) \circ rin(f, m), Z)\ .$$

Now consider the second abstract state where the ferrari is not in Monte Carlo, i.e., $\neg holds(rin(f, m), Z')$ is true. In this case, we obtain

$$\mathcal{FC} \models causes(Z, [unloadF(R, T)], Z') \tag{2.21}$$

with extra condition that $\neg holds(rin(f, m), Z')$ is true.

To solve (2.21), we have to find a refutation for

$$?-\ causes(Z, [unloadF(R, T)], Z')\ .$$

Within two steps, this goal can be reduced to

$$\begin{aligned} ?\!- \ & \mathit{on}(R,T) \circ \mathit{tin}(T,C) \circ X =_{AC1} Z', \\ & \mathit{causes}(Z, [\,], \mathit{on}(R,T) \circ \mathit{tin}(T,C) \circ X) \ . \end{aligned} \tag{2.22}$$

In the next step, the AC1-unification algorithm is called and terminates successfully with a most general unifier

$$\sigma = \{Z' \mapsto \mathit{on}(R,T) \circ \mathit{tin}(T,C) \circ X\}$$

Then we conclude that $\neg \mathit{holds}(rin(f,m), X)$ is true and obtain the computed answer substitution

$$\theta = \{Z \mapsto \mathit{on}(R,T) \circ \mathit{tin}(T,C) \circ X\}$$

with the condition that $\neg \mathit{holds}(rin(f,m), X)$ is true.

Since $\neg \mathit{holds}(rin(f,m), X)$ is true $\mathit{reward}(Z\theta) = 0$.

Thus we obtain the description E of an abstract predecessor state Z

$$E \stackrel{def}{=} \mathit{holds}(\mathit{on}(R,T) \circ \mathit{tin}(T,C), Z) \ \wedge \neg \mathit{holds}(rin(f,m), Z) \ .$$

Next, we combine the obtained logical descriptions with the respective reward values to the system $\mathcal{S}^2_{\mathit{unload}}$

$$\mathcal{S}^2_{\mathit{unload}} = \begin{cases} \langle D, 10 \rangle \\ \langle E, 0 \rangle \ . \end{cases}$$

Altogether, we have obtained five logical descriptions of the predecessor abstract states Z, that are combined in one system $\mathcal{S}_{\mathit{unload}}$ with the respective rewards

$$\mathcal{S}_{\mathit{unload}} = \begin{cases} \langle A, 0 \rangle \\ \langle B, 10 \rangle \\ \langle C, 0 \rangle \\ \langle D, 10 \rangle \\ \langle E, 0 \rangle \ , \end{cases}$$

where

$$\begin{aligned} A &\stackrel{def}{=} \mathit{holds}(\mathit{on}(f,T) \circ \mathit{tin}(T,m), Z) \ \wedge \neg \mathit{holds}(rin(f,m), Z) \\ B &\stackrel{def}{=} \mathit{holds}(\mathit{on}(R,T) \circ \mathit{tin}(T,C) \circ rin(f,m), Z) \\ C &\stackrel{def}{=} \mathit{holds}(\mathit{on}(R,T) \circ \mathit{tin}(T,C), Z) \ \wedge \neg \mathit{holds}(rin(f,m), Z) \ \wedge \\ & \quad (R \neq f \ \vee \ C \neq m) \\ D &\stackrel{def}{=} \mathit{holds}(\mathit{on}(R,T) \circ \mathit{tin}(T,C) \circ rin(f,m), Z) \\ E &\stackrel{def}{=} \mathit{holds}(\mathit{on}(R,T) \circ \mathit{tin}(T,C), Z) \ \wedge \neg \mathit{holds}(rin(f,m), Z) \ . \end{aligned}$$

One should observe that the system $\mathcal{S}_{\mathit{unload}}$ contains some redundancies. E.g., the descriptions B and D are the same and have the same reward values. In addition, the description E is more general than the description A since it states that some racing car is on a truck T and this truck is in some city, whereas the description A specifies that the certain racing car, namely ferrari (f), is on a truck T and this truck is in the certain city, namely Monte-Carlo (m).

It is vital to remove such kind of redundancy from the system $\mathcal{S}_{unload}$ and obtain the so called normal form of the system $\mathcal{S}_{unload}$. Because otherwise, if the redundancies are kept in $\mathcal{S}_{unload}$, they will be propagated to the further iteration steps, thereby negatively affecting the computational requirements of the first-order VIA.

Thus, some normalization process is indeed required. Unfortunately, such normalization of systems has been done by hand so far. Therefore the objective of our work is to provide the solid approach of mechanizing the normalization process by developing the normalization algorithms for various classes of systems of abstract states and analyzing the properties of these algorithms.

Next, we summarize the major procedures of the first-order VIA with an emphasis on possible alternatives of its future implementation.

2.3.4 An Implementation of First-order VIA: Major Procedures

Working towards an implementation of the first-order VIA, we are choosing the normalization process as an objective of our work. Besides the normalization, there are at least two more procedures, that one has to take care of, while designing an implementation of the first-order VIA. Below, we summarize all three procedures together with the possible alterations and additions.

Regression In order to compute the predecessor abstract state space within the first-order VIA the decision-theoretic regression is called. In our approach, the decision-theoretic regression is to be realized by means of the $causes(Z, P, Z')$-predicate, whose arguments denote a state, a sequence of actions (a plan, or policy), and a successor state, respectively. Intuitively, an atom such as $causes(Z, P, Z')$ is to be understood as: The execution of a plan P transforms a state Z into a state Z'. The predicate $causes/3$ is defined recursively on the structure of plans:

$$
\begin{aligned}
&causes(Z, [\,], Z') \leftarrow Z =_{AC1} Z'.\\
&causes(Z, [A|P], Z') \leftarrow action(Pre, A, E) \wedge\\
&\qquad\qquad\qquad\qquad\qquad Z' =_{AC1} Z'' \circ E \wedge\\
&\qquad\qquad\qquad\qquad\qquad causes(Z, P, Z'' \circ Pre).
\end{aligned}
$$

The first clause corresponding to the base case says that if the goal state contains the initial one then there is nothing to do. The second clause can be read as follows: an execution of the plan $[A|P]$ transforms a state Z into a state Z' if there is an action A with preconditions Pre and effects E and there is a state Z'' satisfying an equation $Z' =_{AC1} Z'' \circ E$ and after performing an action A the plan P transforms Z into $Z'' \circ Pre$. An equation $Z' =_{AC1} Z'' \circ E$ represents the unification procedure for the fluent terms Z' and $Z'' \circ E$ under the $AC1$ equational system.

One should observe that already for *CN-states* the regression through *causes*-predicate becomes problematic. The problems arise because the *causes*-predicate was initially designed for states that are defined as fluent terms. Adding the negation and disjunction into the language of abstract states appeals to the formal refinement of the *causes*-predicate and, consequently, the $AC1$-unification procedure, with respect to the negation and disjunction.

Therefore, in order to realize decision-theoretic regression for *CN-states* as well as for *CND-states* (they are *CN-states* that allow for disjunction within the abstract state description) we would need to make an additional effort in refining the *causes*-predicate. Since the negative part of a *CN-state* is just a set of fluent terms, the *causes*-predicate is applied to every fluent term out of a negative part of a *CN-state* in a usual way. The disjunction within a *CN-state* is to be treated as described in Section 3.2.5. Therefore, the *causes*-predicate called for a *CND-state* will be applied to every disjunct.

Alternatively, the decision-theoretic regression can be realized (as it is claimed by authors of FC [33]) with the help of the Fluent Calculus Executor (FLUX) that is an implementation of the last version of the fluent calculus [33, 34] using constraint logic programming.

Constraint logic programming could help us to treat the constraints that may appear in the abstract state descriptions. The constraints like in the following description

$$\begin{aligned} C \stackrel{def}{=} & \ holds\,(on\,(R,T) \circ tin\,(T,C), Z) \ \wedge \neg holds\,(rin(f,m), Z) \ \wedge \\ & (R \neq f \ \vee \ C \neq m) \end{aligned}$$

would be processed by the specially designed constraint solver.

The constraint solver would enable to obviate the direct transformation (that is actually applied in our approach) of the constraints like $(R \neq f \ \vee \ C \neq m)$ into the negated *holds*-statements.

In order to illustrate such transformation, we consider again the description C, where the constraint $(R \neq f \ \vee \ C \neq m)$ is replaced by the respective *holds*-statement.

$$\begin{aligned} C \stackrel{def}{=} & \ holds\,(on\,(R,T) \circ tin\,(T,C), Z) \ \wedge \neg holds\,(rin(f,m), Z) \ \wedge \\ & \neg holds\,(on(f,T) \circ tin(T,m), Z)\ . \end{aligned}$$

Implementation of the decision-theoretic regression within FLUX using constraint handling rules seems to be a prominent direction of the future work, but the (minor or major) changes should be performed in order to turn the underlying fluent calculus into the backward reasoning approach. Since our goal was mainly to enhance the first-order VIA presented in Figure 2.3 by introducing the suitable normalization techniques, we have opted for the original version of the FC that initially allows for backward reasoning.

Normalization Within the first-order value iteration algorithm the systems of abstract states are computed. These systems may contain redundant abstract states. For instance, in the example scenario in Section 2.3.3, we had five abstract states where three of them were redundant. If the redundancies are kept in a system, they may cause the exponentially increasing number of redundant computations within the future iteration steps of the algorithm. Therefore, in order to improve the computational requirements of the first-order value iteration algorithm it has to rely on some kind of normalization technique. A normalization procedure should be designed in such a way that, given a system of abstract states, it delivers a normal form of this system that is free of redundancies. Thus, the kernel of the normalization procedure should contain a method that enables to determine the redundant abstract states.

Verification Given an abstract state description Z and the agent's state Δ of the real world, the process of detecting whether $\Delta \models Z$ is called a verification process. In order to operate with such notions as entailment relation we need to define the semantics of the languages of abstract states first. Presumably, a kind of a calculus that allows to answer the question of whether Z is derivable from Δ has to be accomplished. In case of a sound and complete calculus, this question coincides with the entailment problem $\Delta \models Z$.

Chapter 3

Formal Description

This chapter is divided into two sections, corresponding to two classes of abstract states: *C-states* and *CN-states*. Section 3.1 devises a normalization algorithm for *C-states* by defining the language of *C-states*, introducing the subsumption relation on them, and specifying the simplification rules. The chapter is closed by elaborating on the complexity results and proving such important properties of the normalization algorithm as correctness and termination. In Section 3.2, the language of *CN-states* is developed and the similar issues as in Section 3.1 are elaborated upon.

Consider an example scenario with cars, trucks and cities from Section 2.2 and assume that one step of the decision-theoretic regression has delivered the system $\mathcal{S}$, consisting of the pairs $\langle \mathit{abstract\ state}, \mathit{reward} \rangle$, as a result.

$$\mathcal{S} = \begin{cases} \langle Z_1 = on(f,T) \circ tin(T,m) \circ X_1 \ \wedge \ \neg holds(rin(f,m), Z_1), \quad 0 \rangle \\ \langle Z_2 = on(R,T) \circ tin(T,C) \circ rin(f,m) \circ X_2, \quad 10 \rangle \\ \langle Z_3 = on(R,T) \circ tin(T,C) \circ X_3 \ \wedge \ \neg holds(rin(f,m), Z_3) \ \wedge \\ \qquad \neg holds(on(f,T) \circ tin(T,m), Z_3), \quad 0 \rangle \\ \langle Z_4 = on(R,T) \circ tin(T,C) \circ rin(f,m) \circ X_4, \quad 10 \rangle \\ \langle Z_5 = on(R,T) \circ tin(T,C) \circ X_5 \wedge \ \neg holds(rin(f,m), Z_5), \quad 0 \rangle \ . \end{cases}$$

One can observe that the logical descriptions in the system $\mathcal{S}$ consist of a positive part, that is specified with the help of the symbol $\circ/2$ (it works, in fact, as conjunction), and a negative part, that is specified by means of the *holds*-macro taken with negation.

Thus, such logical descriptions originate in two various classes of abstract states. The first class is the class of *C-states*, where C stands for the 'conjunction', since the conjunction symbol is the only available connective used within an abstract state description. The second class of abstract states, called *CN-states*, where N stands for negation, is an extension of the class of *C-states* by negation.

We proceed by defining the language of *C-states*, elaborating on their properties and designing a normalization algorithm for them.

3.1 *C-states*

3.1.1 The Language

We start with defining a language $\mathcal{L}_1$ of *C-states*.

Definition 11 *A C-state Z is either one of two:*

- *Z is a fluent term.*
- *Z has the form $Z = F \circ X$, where F is a fluent term and X is a variable that does not occur in F.*

For instance, consider a *C-state* Z_1, that is a ground fluent term with an informal semantics that the racing car ferrari is on the truck honda and this truck is in Paris,

$$Z_1 = on(ferrari, honda) \circ tin(honda, paris) \ .$$

A *C-state* could be also represented as a non-ground fluent term. As for example, a *C-state* Z_2 with the informal meaning 'the racing car ferrari is on some truck T and this truck is in Paris':

$$Z_2 = on(ferrari, T) \circ tin(T, paris) \ .$$

Suppose that the domain has only three trucks. Namely, *honda*, *toyota* and *volvo*. Then the *C-state* Z_2 represents a set of ground fluent terms

$$\begin{array}{l} on(ferrari, honda) \circ tin(honda, paris) \\ on(ferrari, toyota) \circ tin(toyota, paris) \\ on(ferrari, volvo) \circ tin(volvo, paris) \ , \end{array}$$

whereas the *C-state* Z_1 represents a one element set, consisting of the ground fluent term $on(ferrari, honda) \circ tin(honda, paris)$.

And finally, take the *C-state* Z_3 as an illustration of the second case in Definition 11,

$$Z_3 = on(ferrari, honda) \circ X,$$

with the understanding that the fluent $on(ferrari, honda)$ holds in Z_3 and, in addition, some other fluents may hold too. This kind of uncertainty is expressed by means of a variable X. In fact, the presence of a variable X in the *C-state* Z_3 allows us to represent the set of ground fluent terms that always contain the fluent term $on(ferrari, honda)$. E.g., if we have only three fluents in the domain, i.e., $on(R, T)$ (a racing car R is on a truck T), $rin(R, C)$ (a racing car R is in a city C) and $rain$ (it is raining) and four domain individuals, i.e., *ferrari* as a racing car, *honda* as a truck, and *paris* and *monteCarlo* as cities then the fluent terms of the kind $on(ferrari, honda)$, $on(ferrari, honda) \circ rin(ferrari, monteCarlo)$, $on(ferrari, honda) \circ rin(ferrari, paris)$, $on(ferrari, honda) \circ rain$ and others are represented by the *C-state* Z_3.

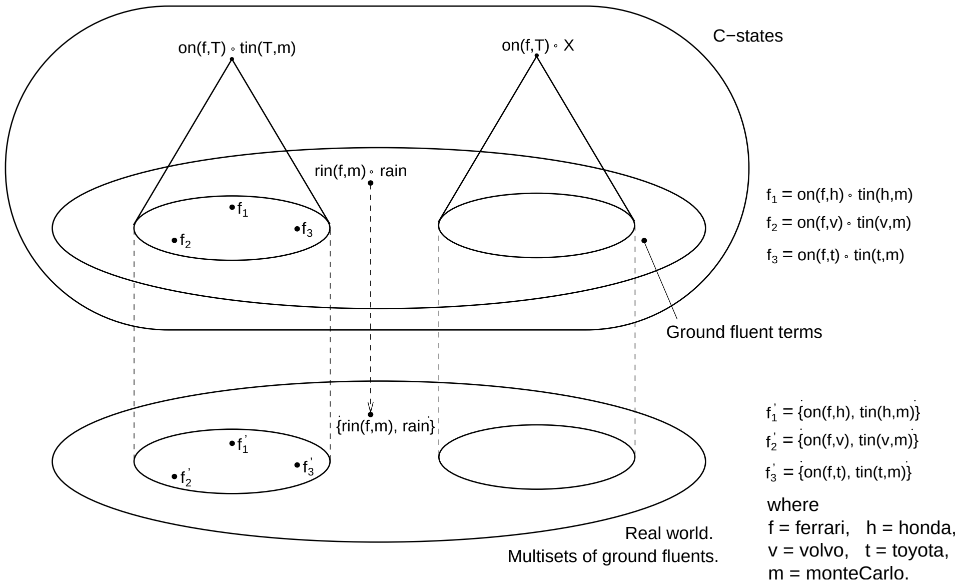

Figure 3.1: *C-states*: Intuition.

The illustration of the above-presented intuition for the *C-states* is presented in Figure 3.1. As it is shown in Figure 3.1 the *C-states* can be ground fluent terms, e.g., $on(f,h) \circ tin(h,m)$ or $rin(f,m) \circ rain$, where f stands for the racing car ferrari, h for the truck honda, and m for the city Monte Carlo. All fluent terms in Figure 3.1 have the intended meaning. Additionally, a *C-state* can be represented by a non-ground fluent term, as for example, $on(f,T) \circ tin(T,m)$, or can be of the form $F \circ X$, where F is a fluent term and X is a variable, e.g., $on(f,T) \circ X$. The latter two kinds of the *C-states* correspond to the sets of ground fluent terms. These sets are then projected to the respective sets of multisets of ground fluents. For instance, the set of ground fluent terms

$$\{on(f,h) \circ tin(h,m), on(f,v) \circ tin(v,m), on(f,t) \circ tin(t,m)\}$$

corresponds to the set of multisets of the ground fluents

$$\{\dot{\{}on(f,h), tin(h,m)\dot{\}}, \dot{\{}on(f,v), tin(v,m)\dot{\}}, \dot{\{}on(f,t), tin(t,m)\dot{\}}\} \ .$$

For the case where the *C-state* is a ground fluent term we have a one-to-one correspondence between the *C-states* and the multisets of ground fluents. For instance, the ground fluent term

$$rin(f,m) \circ rain$$

becomes a multiset

$$\dot{\{}rin(f,m), rain\dot{\}} \ .$$

3.1.2 The Simplification of Systems

The first-order VIA presented in Section 2.3 operates on multisets (sets, resp.) of pairs of the kind $\langle Z, \alpha \rangle$, where Z is an abstract state and α is its numeric value. We call such multisets (sets, resp.) of pairs systems.

Definition 12 *A system $\mathcal{S}$ is a multiset of pairs $\langle Z, \alpha \rangle$, where Z is a C-state and α is a numeric value for Z.*

At different steps of the first-order VIA a numeric value α can represent a reward $reward(Z)$, a q-value, or a value of the optimal value function $v(Z)$ for a *C-state* Z.

As an example of a system consider the system $\mathcal{S}$ that consists of five *C-states* together with their numeric values.

$$\begin{array}{lr}
\langle Z_1 = on(f,T) \circ tin(T,m) \circ X_1, & 0\rangle \\
\langle Z_2 = on(R,h) \circ tin(h,C) \circ rin(f,m) \circ X_2, & 10\rangle \\
\langle Z_3 = on(R,T) \circ tin(T,C) \circ X_3, & 0\rangle \\
\langle Z_4 = on(R,T) \circ tin(T,C) \circ rin(f,m) \circ X_4, & 10\rangle \\
\langle Z_5 = on(R,T) \circ tin(T,C) \circ X_5, & 0\rangle \ .
\end{array}$$

One should observe that the system $\mathcal{S}$ contains redundancies. Namely, the *C-state* Z_3 is more general than the *C-state* Z_1 because the *C-state* Z_3 states that *some* racing car R is on some truck T and this truck is in *some* city C. Whereas the *C-state* Z_1 provides us with more specific information; the concrete racing car, i.e., *ferrari* (f), is on some truck T and this truck in the concrete city, i.e., Monte Carlo (m). It appears that the *C-state* Z_1 is redundant since there exists the *C-state* Z_3 that expresses more general information.

Using the same reasoning we conclude that the *C-state* Z_2 is redundant. Moreover, the *C-states* Z_3 and Z_5 have the same logical descriptions. Therefore, one of them is redundant, but which one will be clear in the sequel.

Intuitively, all pairs $\langle Z, \alpha \rangle \in \mathcal{S}$, where Z is a redundant *C-state* must be deleted from the system $\mathcal{S}$. Then as a result we obtain the system $\mathcal{S}'$ that consists of only two pairs.

$$\begin{array}{lr}
\langle Z_4 = on(R,T) \circ tin(T,C) \circ rin(f,m) \circ X_4, & 10\rangle \\
\langle Z_5 = on(R,T) \circ tin(T,C) \circ X_5, & 0\rangle \ .
\end{array}$$

One should note here that only the pairs $\langle Z_1, \alpha_1 \rangle$ and $\langle Z_2, \alpha_2 \rangle$ that have the same numeric values, i.e., $\alpha_1 = \alpha_2$, can be compared. Otherwise, the pairs are said to be incomparable.

As it was shown in the above example the *C-state* Z_3 is more general than the *C-state* Z_1. It implies that these *C-states* are in some kind of a generalization relation in which a subsumer (Z_3) expresses the generalization over a subsumee (Z_1).

Our task is then to develop such kind of relation to compare the *C-states*. This relation should give an ability to determine which *C-state* is 'more general' than another one.

Such generalization relation is called *subsumption relation* and defined as follows.

Definition 13 *Let Z_1, Z_2 be C-states. Then Z_1 is said **to subsume** Z_2 written $Z_2 \sqsubseteq Z_1$ iff one of the following cases holds:*

1. $Z_1 = T_1 \circ \ldots \circ T_n$ *and* $Z_2 = S_1 \circ \ldots \circ S_m$ *and* $\exists \sigma.\ (T_1 \circ \ldots \circ T_n)\sigma =_{AC1} S_1 \circ \ldots \circ S_m$

2. $Z_1 = T_1 \circ \ldots \circ T_n \circ X$ *and (*$Z_2 = S_1 \circ \ldots \circ S_m \circ Y$ *or* $Z_2 = S_1 \circ \ldots \circ S_m$*) and* $\exists \sigma.\ (T_1 \circ \ldots \circ T_n \circ X)\sigma =_{AC1} S_1 \circ \ldots \circ S_m$,

where $T_1, \ldots, T_n$, $S_1, \ldots, S_m$ *are fluents,* σ *is a substitution,* X *is a variable that does not occur in* T_j, $1 \leq j \leq n$, *and* Y *is a variable that does not occur in* S_k, $1 \leq k \leq m$.

Proposition 4 *Let* Z_1 *and* Z_2 *be C-states. Then* Z_1 *and* Z_2 *are* ***variants*** *modulo* $AC1$ *if*

$$Z_2 \sqsubseteq Z_1 \wedge Z_1 \sqsubseteq Z_2\ .$$

Proof From $Z_2 \sqsubseteq Z_1 \wedge Z_1 \sqsubseteq Z_2$ we obtain

$$\begin{array}{l} \exists \sigma.\ Z_1\sigma =_{AC1} Z_2 \\ \exists \theta.\ Z_2\theta =_{AC1} Z_1\ . \end{array}$$

Since it is known that two terms S and T are variants of each other if the following condition holds

$$\begin{array}{l} \exists \sigma.\ T\sigma = S \\ \exists \theta.\ S\theta = T\ , \end{array}$$

then we conclude that Z_1 and Z_2 are variants modulo $AC1$.

□

In order to illustrate the subsumption relation over *C-states* we propose to consider the following example.

Example 2 Take the *C-states* Z_1 and Z_2 such that

$$Z_1 = on(ferrari, honda) \circ tin(honda, monteCarlo)$$

informally denoting the fact that the racing car ferrari is on the truck honda and this truck is in the city Monte-Carlo; and

$$Z_2 = on(ferrari, T) \circ tin(T, monteCarlo)$$

with the understanding that the racing car ferrari is on some truck T and this truck is in Monte-Carlo.

Then according to Definition 13 $Z_1 \sqsubseteq Z_2$, since there exists a substitution $\sigma = \{T \mapsto honda\}$ such that

$$\begin{array}{l} (on(ferrari, T) \circ tin(T, monteCarlo))\sigma =_{AC1} \\ \qquad\qquad on(ferrari, honda) \circ tin(honda, monteCarlo). \end{array}$$

For the illustration of the second case of Definition 13 consider two *C-states* Z_3 and Z_4 such that

$$Z_3 = on(\mathit{ferrari}, T) \circ X \ ,$$

with the understanding that the racing car ferrari is on some truck T and some other fluents may hold in the *C-state* Z_3, and

$$Z_4 = on(\mathit{ferrari}, \mathit{honda}) \circ tin(\mathit{honda}, C) \circ Y$$

with the informal semantics that the racing car ferrari is on the truck honda that is in a city C and some other fluents may hold in Z_4 as well. Then $Z_4 \sqsubseteq Z_3$ since there exists a substitution $\sigma = \{T \mapsto \mathit{honda}, X \mapsto tin(\mathit{honda}, C)\}$ such that

$$(on(\mathit{ferrari}, T) \circ X)\sigma =_{AC1} on(\mathit{ferrari}, \mathit{honda}) \circ tin(\mathit{honda}, C).$$

Finally, take the *C-state* Z_5 as

$$Z_5 = tin(TR, \mathit{monteCarlo}) \circ on(\mathit{ferrari}, TR) \ .$$

Then according to Definition 13, $Z_2 \sqsubseteq Z_5$ with the substitution $\sigma = \{TR \mapsto T\}$ and $Z_5 \sqsubseteq Z_2$ with the substitution $\theta = \{T \mapsto TR\}$. By Proposition 4, Z_2 and Z_5 are variants modulo $AC1$.

Hence, one chooses don't-care non-deterministically for which of the two alternatives to opt. For instance, in our implementation in Appendix A, in such cases we always assume that the *C-state*, that was given second to the program, subsumes the one given first. Under this assumption, in the example above, we will opt for the first alternative, i.e., $Z_2 \sqsubseteq Z_5$.

■

The subsumption relation is reflexive and transitive.

Property 1 *The subsumption relation on C-states is reflexive.*

Proof The statement will be shown by case analysis on the definition of the *C-states*. Altogether, there are two cases

1. $Z = T_1 \circ \ldots \circ T_n$, where $T_1, \ldots, T_n$ are fluents. Then by Definition 13, $Z \sqsubseteq Z$ since there exists a substitution $\sigma = \epsilon$ such that

$$(T_1 \circ \ldots \circ T_n)\sigma =_{AC1} T_1 \circ \ldots \circ T_n.$$

2. $Z = T_1 \circ \ldots \circ T_n \circ X$, where $T_1, \ldots, T_n$ are fluents and X is a variable that does not occur in T_j, $1 \leq j \leq n$. Then by Definition 13, $Z \sqsubseteq Z$, since there exists a substitution $\sigma = \{X \mapsto 1\}$ such that

$$(T_1 \circ \ldots \circ T_n \circ X)\sigma =_{AC1} T_1 \circ \ldots \circ T_n.$$

□

Property 2 *The subsumption relation on C-states is transitive.*

Proof We shall prove the statement by case analysis. Altogether, there are eight cases to consider. Let $T_1, \ldots, T_n$, $S_1, \ldots, S_m$, and $M_1, \ldots, M_k$ denote fluent terms.

1. $Z_1 = T_1 \circ \ldots \circ T_n$, $Z_2 = S_1 \circ \ldots \circ S_m$, $Z_3 = M_1 \circ \ldots \circ M_k$. We need to show that $Z_3 \sqsubseteq Z_1$ if $Z_2 \sqsubseteq Z_1$ and $Z_3 \sqsubseteq Z_2$. By Definition 13 we obtain the following
$$\exists \sigma_1.\ (T_1 \circ \ldots \circ T_n)\sigma_1 =_{AC1} S_1 \circ \ldots \circ S_m$$
$$\exists \sigma_2.\ (S_1 \circ \ldots \circ S_m)\sigma_2 =_{AC1} M_1 \circ \ldots \circ M_k$$
A substitution $\sigma_3 = \sigma_1\ \sigma_2$ satisfies the equation
$$(T_1 \circ \ldots \circ T_n)\sigma_3 =_{AC1} M_1 \circ \ldots \circ M_k\ .$$
Hence, $Z_3 \sqsubseteq Z_1$.

2. $Z_1 = T_1 \circ \ldots \circ T_n$, $Z_2 = S_1 \circ \ldots \circ S_m$, $Z_3 = M_1 \circ \ldots \circ M_k \circ Z$. We need to show that $Z_3 \sqsubseteq Z_1$ if $Z_2 \sqsubseteq Z_1$ and $Z_3 \sqsubseteq Z_2$. But Z_2 does not subsume Z_3 because neither first nor second case from Definition 13 is applied.

3. $Z_1 = T_1 \circ \ldots \circ T_n$, $Z_2 = S_1 \circ \ldots \circ S_m \circ Y$, $Z_3 = M_1 \circ \ldots \circ M_k$. By the same reason as in case 2, Z_1 does not subsume Z_2.

4. $Z_1 = T_1 \circ \ldots \circ T_n$, $Z_2 = S_1 \circ \ldots \circ S_m \circ Y$, $Z_3 = M_1 \circ \ldots \circ M_k \circ Z$. A state Z_1 does not subsume a state Z_2 as in case 3.

5. $Z_1 = T_1 \circ \ldots \circ T_n \circ X$, $Z_2 = S_1 \circ \ldots \circ S_m$, $Z_3 = M_1 \circ \ldots \circ M_k$.
Case 5 is similar to case 1. Thus, $Z_3 \sqsubseteq Z_1$ with the resulting substitution $\sigma_3 = \sigma_1\ \sigma_2$.

6. $Z_1 = T_1 \circ \ldots \circ T_n \circ X$, $Z_2 = S_1 \circ \ldots \circ S_m$, $Z_3 = M_1 \circ \ldots \circ M_k \circ Z$. A state Z_2 does not subsume a state Z_3 as in case 2.

7. $Z_1 = T_1 \circ \ldots \circ T_n \circ X$, $Z_2 = S_1 \circ \ldots \circ S_m \circ Y$, $Z_3 = M_1 \circ \ldots \circ M_k$.
By Definition 13 we have
$$\exists \sigma_1.\ (T_1 \circ \ldots \circ T_n \circ X)\sigma_1 =_{AC1} S_1 \circ \ldots \circ S_m$$
$$\exists \sigma_2.\ (S_1 \circ \ldots \circ S_m \circ Y)\sigma_2 =_{AC1} M_1 \circ \ldots \circ M_k\ .$$
Hence, $Z_2 \sqsubseteq Z_1$ and $Z_3 \sqsubseteq Z_2$. A resulting substitution
$$\sigma_3 = \sigma_1|_{\text{VAR}(T_1,\ldots,T_n)}\ \sigma_2|_{\text{VAR}(S_1,\ldots,S_m)} \cup \{X \mapsto U_1 \circ \ldots \circ U_l \circ V_1 \circ \ldots \circ V_p\}\ ,$$
where
$$\dot{\{}U_1, \ldots, U_l\dot{\}} \doteq \dot{\{}S_1, \ldots, S_m\dot{\}} \setminus \dot{\{}T_1\sigma_1, \ldots, T_n\sigma_1\dot{\}}$$
and
$$\dot{\{}V_1, \ldots, V_p\dot{\}} \doteq \dot{\{}M_1, \ldots, M_k\dot{\}} \setminus \dot{\{}S_1\sigma_2, \ldots, S_m\sigma_2\dot{\}}$$
satisfies the equation
$$(T_1 \circ \ldots \circ T_n \circ X)\sigma_3 =_{AC1} M_1 \circ \ldots \circ M_k\ .$$

Let $\mathcal{S}$ be a system such that $\langle Z_1, \alpha\rangle, \langle Z_2, \alpha\rangle \in \mathcal{S}$, where Z_1, Z_2 are *C-states* and α is a numeric value.

$$\frac{\langle Z_1, \alpha\rangle \quad \langle Z_2, \alpha\rangle \quad Z_2 \sqsubseteq Z_1}{\langle Z_1, \alpha\rangle}\ .$$

Figure 3.2: Simplification rule for the *C-states*.

8. $Z_1 = T_1 \circ \ldots \circ T_n \circ X$, $Z_2 = S_1 \circ \ldots \circ S_m \circ Y$, $Z_3 = M_1 \circ \ldots \circ M_k \circ Z$. Since the cases 7 and 8 are similar, σ_3 for the case 8 is constructed in the same way as for the case 7.

$\square$

As we have stated in the beginning of the current section, a *C-state*, that is more specific than another one, or, in terms of the subsumption relation, is subsumed by another one, is redundant and must be deleted from a system, under the assumption that *C-states* have the same numeric values.

This statement expresses the intuition of the simplification rule on *C-states* that is formally specified in Figure 3.2. According to this rule, having assumed that two *C-states* Z_1, Z_2 have the same numeric value α, if Z_1 subsumes Z_2 then the pair $\langle Z_2, \alpha\rangle$ is to be removed from a system.

It is crucial to require that the simplification step, based on the simplification rule applied to a system $\mathcal{S}$, delivers a system that is equivalent to $\mathcal{S}$.

The equivalence between the systems is defined in a natural way. Namely, the systems $\mathcal{S}_1$ and $\mathcal{S}_2$ are called equivalent if and only if the following conditions are met. First, if for all pairs $\langle Z_i, \alpha_i\rangle \in \mathcal{S}_1$ there exists a pair $\langle Z_j, \alpha_i\rangle \in \mathcal{S}_2$ such that Z_j subsumes Z_i. The second condition described the opposite case. That is to say, if for all pairs $\langle Z_j, \alpha_j\rangle \in \mathcal{S}_2$ there exists a pair $\langle Z_i, \alpha_j\rangle \in \mathcal{S}_1$ such that Z_i subsumes Z_j.

Definition 14 *Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be systems. The systems $\mathcal{S}_1$ and $\mathcal{S}_2$ are called* ***equivalent*** *iff both conditions are satisfied:*

1. $\forall Z_i, \alpha_i$ *such that* $\langle Z_i, \alpha_i\rangle \in \mathcal{S}_1$. $\exists\langle Z_j, \alpha_i\rangle \in \mathcal{S}_2$. $Z_i \sqsubseteq Z_j$;
2. $\forall Z_j, \alpha_j$ *such that* $\langle Z_j, \alpha_j\rangle \in \mathcal{S}_2$. $\exists\langle Z_i, \alpha_j\rangle \in \mathcal{S}_1$. $Z_j \sqsubseteq Z_i$,

where Z_i, Z_j are C-states.

For instance, consider two systems $\mathcal{S}_1$ and $\mathcal{S}_2$.

$$\mathcal{S}_1 = \begin{cases} \langle on(R,T) \circ tin(T,C) \circ X, 10\rangle \\ \langle on(f,T) \circ tin(T,m) \circ rain \circ Y, 10\rangle \\ \langle rin(f,m), 9\rangle \end{cases}$$

$$\mathcal{S}_2 = \left\{ \begin{array}{l} \langle on(R,T) \circ tin(T,C) \circ X, 10\rangle \\ \langle rin(f,m), 9\rangle \end{array} \right.$$

These systems are equivalent to each other because the conditions from Definition 14 are met. Namely, for the pair $\langle on(R,T) \circ tin(T,C) \circ X, 10\rangle \in \mathcal{S}_1$ there exists a pair $\langle on(R,T) \circ tin(T,C) \circ X, 10\rangle \in \mathcal{S}_2$ such that

$$on(R,T) \circ tin(T,C) \circ X \sqsubseteq on(R,T) \circ tin(T,C) \circ X \ .$$

For the pair $\langle on(f,T) \circ tin(T,m) \circ rain \circ Y, 10\rangle \in \mathcal{S}_1$ there exists a pair $\langle on(R,T) \circ tin(T,C) \circ X, 10\rangle \in \mathcal{S}_2$ such that

$$on(f,T) \circ tin(T,m) \circ rain \circ Y \sqsubseteq on(R,T) \circ tin(T,C) \circ X \ .$$

And finally, for the pair $\langle rin(f,m), 9\rangle \in \mathcal{S}_1$ we find the pair $\langle rin(f,m), 9\rangle \in \mathcal{S}_2$ such that

$$rin(f,m) \sqsubseteq rin(f,m) \ .$$

As for the second condition in Definition 14, for each pair in the system $\mathcal{S}_2$ we need to find a pair in the system $\mathcal{S}_1$ such that a *C-state* in $\mathcal{S}_1$ subsumes a respective *C-state* in $\mathcal{S}_2$. Namely, for the pair $\langle on(R,T) \circ tin(T,C) \circ X, 10\rangle \in \mathcal{S}_2$ there exists a pair $\langle on(R,T) \circ tin(T,C) \circ X, 10\rangle \in \mathcal{S}_1$ such that

$$on(R,T) \circ tin(T,C) \circ X \sqsubseteq on(R,T) \circ tin(T,C) \circ X \ .$$

For the pair $\langle rin(f,m), 9\rangle \in \mathcal{S}_2$ we find the pair $\langle rin(f,m), 9\rangle \in \mathcal{S}_1$ such that

$$rin(f,m) \sqsubseteq rin(f,m) \ .$$

Since both conditions in Definition 14 are satisfied, the systems $\mathcal{S}_1$ and $\mathcal{S}_2$ are equivalent.

Below are given the systems that are not equivalent to the system $\mathcal{S}_1$.

$$\mathcal{S}_3 = \left\{ \begin{array}{l} \langle on(R,T) \circ tin(T,C) \circ X, 20\rangle \\ \langle rin(f,m), 9\rangle \end{array} \right.$$

In the first case, the systems $\mathcal{S}_1$ and $\mathcal{S}_3$ are not equivalent because the values of the respective *C-states* are different. Namely, the *C-states*

$$on(R,T) \circ tin(T,C) \circ X \ such\ that\ \langle on(R,T) \circ tin(T,C) \circ X, 10\rangle \in \mathcal{S}_1$$

and

$$on(R,T) \circ tin(T,C) \circ X \ such\ that\ \langle on(R,T) \circ tin(T,C) \circ X, 20\rangle \in \mathcal{S}_2$$

have different numeric values.

$$\mathcal{S}_4 = \left\{ \begin{array}{l} \langle on(f,h) \circ tin(h,m) \circ X, 10\rangle \\ \langle rin(f,m), 9\rangle \end{array} \right.$$

The systems $\mathcal{S}_1$ and $\mathcal{S}_4$ are not equivalent because for the pair $\langle on(R,T) \circ tin(T,C) \circ X, 10\rangle \in \mathcal{S}_1$ there exists no pair $\langle Z, 10\rangle \in \mathcal{S}_4$ such that

$$on(R,T) \circ tin(T,C) \circ X \sqsubseteq Z \ .$$

Property 3 *The equivalence relation on systems over C-states is transitive.*

Proof We need to prove the following two statements

1. If
$$\forall Z_i, \alpha_i \text{ such that } \langle Z_i, \alpha_i \rangle \in \mathcal{S}_1.\ \exists \langle Z_j, \alpha_i \rangle \in \mathcal{S}_2.\ Z_i \sqsubseteq Z_j\ ;$$
and
$$\forall Z_j, \alpha_j \text{ such that } \langle Z_j, \alpha_j \rangle \in \mathcal{S}_2.\ \exists \langle Z_k, \alpha_j \rangle \in \mathcal{S}_3.\ Z_j \sqsubseteq Z_k\ ;$$
then
$$\forall Z_i, \alpha_i \text{ such that } \langle Z_i, \alpha_i \rangle \in \mathcal{S}_1.\ \exists \langle Z_k, \alpha_i \rangle \in \mathcal{S}_3.\ Z_i \sqsubseteq Z_k\ .$$
The statement follows directly from the Property 2.

2. If
$$\forall Z_j, \alpha_j \text{ such that } \langle Z_j, \alpha_j \rangle \in \mathcal{S}_2.\ \exists \langle Z_i, \alpha_j \rangle \in \mathcal{S}_1.\ Z_j \sqsubseteq Z_i$$
and
$$\forall Z_i, \alpha_i \text{ such that } \langle Z_i, \alpha_i \rangle \in \mathcal{S}_1.\ \exists \langle Z_k, \alpha_i \rangle \in \mathcal{S}_3.\ Z_i \sqsubseteq Z_k$$
then
$$\forall Z_j, \alpha_j \text{ such that } \langle Z_j, \alpha_j \rangle \in \mathcal{S}_2.\ \exists \langle Z_k, \alpha_j \rangle \in \mathcal{S}_3.\ Z_j \sqsubseteq Z_k\ .$$
The statement holds by Property 2.

□

Once the reader has become accustomed to the simplification rule based on the subsumption relation for the *C-states*, next we will concentrate on the major result of our work. Namely, on the normalization algorithm and its properties.

3.1.3 The Normalization Algorithm

The normalization algorithm is intended to deliver normal form of a system being normalized if certain conditions are met. But before we go into depths of the normalization algorithm and its properties, we need to formally define *normal form* of a system. The definition of a normal form should certainly capture the nature of the simplification rule given in Figure 3.2.

Definition 15 *Let $\mathcal{S}$ be a system such that $\langle Z_k, \alpha_k \rangle \in \mathcal{S}$ for all $k = 1 \ldots n$. A system $\mathcal{S}$ is in **normal form** iff the following condition is satisfied:*

- *There is no $i \neq j$ such that $Z_j \sqsubseteq Z_i$ and $\alpha_i = \alpha_j$.*

`Input:` A system $\mathcal{S}$ such that $\langle Z_i, \alpha_i \rangle \in \mathcal{S}$ where Z_i is a *C-state*, α_i is a numeric value for Z_i, and $i = 1 \dots n$.
`Output:` A system $\mathcal{S}'$ in normal form such that $\mathcal{S}'$ is equivalent to $\mathcal{S}$.

`while` $\mathcal{S}$ is not in normal form `do`

1. select don't-know non-deterministically two pairs $\langle Z_1, \alpha_1 \rangle$, $\langle Z_2, \alpha_2 \rangle \in \mathcal{S}$ such that the rule from Figure 3.2 is applicable;

2. if such two pairs exist then apply the rule on Figure 3.2 to them;

`end while`
exit with $\mathcal{S}' = \mathcal{S}$.

Figure 3.3: Normalization algorithm for *C-states*.

Sometimes, by the term 'normal form of a system $\mathcal{S}$' we mean a system that is equivalent to $\mathcal{S}$ and is in normal form.

As an example, consider the systems $\mathcal{S}_1$ and $\mathcal{S}_2$ defined above. The system $\mathcal{S}_1$ has the redundancies because it contains the pairs

$$\langle Z_1 = on(R, T) \circ tin(T, C) \circ X, 10 \rangle$$

and

$$\langle Z_2 = on(f, T) \circ tin(T, m) \circ rain \circ Y, 10 \rangle$$

such that $Z_2 \sqsubseteq Z_1$ with a substitution $\sigma = \{R \mapsto f, C \mapsto m, X \mapsto rain\}$.

Hence, the system $\mathcal{S}_1$ is not in normal form. Whereas, the system $\mathcal{S}_2$ is in normal form.

The normalization algorithm based on the simplification rule from Figure 3.2 is presented in Figure 3.3. The normalization algorithm takes as an input a system $\mathcal{S}$ and returns a system $\mathcal{S}'$ that is a normal form of $\mathcal{S}$.

Although this result is intuitive and straightforward, it lacks the formal proof. First, we introduce Lemma 1 that states that a system $\mathcal{S}_n$ is equivalent to a system $\mathcal{S}_0$, where $\mathcal{S}_n$ is obtained as a result of applying the rule in Figure 3.2 to $\mathcal{S}_0$ n times.

Lemma 1 *Let $\mathcal{S}_0$ be a system. A system $\mathcal{S}_n$ is obtained as a result of applying the rule on Figure 3.2 to $\mathcal{S}_0$ n times, where n is an arbitrary natural number. Then the system $\mathcal{S}_n$ is equivalent to the system $\mathcal{S}_0$.*

Proof The proof will be conducted by induction over n. Base case is n=0. By Definition 14, the system $\mathcal{S}_0$ is equivalent to itself. Assume that for n-times-application of the rule in Figure 3.2 the system $\mathcal{S}_n$ is equivalent to $\mathcal{S}_0$. Let $\mathcal{S}_{n+1}$

denote the system obtained as a result of applying the simplification rule one time to the system $\mathcal{S}_n$.

Let $\mathcal{S}_n = \{\langle Z_1, \alpha_1\rangle, \dots \langle Z_l, \alpha_l\rangle, \dots \langle Z_k, \alpha_k\rangle\}$ be a system. Let

$$\mathcal{S}_{n+1} = \{\langle Z_1, \alpha_1\rangle, \dots \langle Z_{l-1}, \alpha_{l-1}\rangle, \langle Z_{l+1}, \alpha_{l+1}\rangle, \dots \langle Z_k, \alpha_k\rangle\}$$

be a result of a single application of the rule in Figure 3.2. This means that there exists a pair $\langle Z_m, \alpha_l\rangle \in \mathcal{S}_n$ such that $Z_l \sqsubseteq Z_m$.

In order to show that the system $\mathcal{S}_{n+1}$ is equivalent to the system $\mathcal{S}_n$, we need to check that the systems $\mathcal{S}_n$ and $\mathcal{S}_{n+1}$ satisfy both conditions in Definition 14.

1. In order to check that the first condition in Definition 14 is satisfied, we distinguish two cases.

   - Firstly, we need to prove that

     $$\forall Z_i, \alpha_i.\ \langle Z_i, \alpha_i\rangle \in \mathcal{S}_n,\ i \neq l.\ \exists \langle Z_j, \alpha_i\rangle \in \mathcal{S}_{n+1}.\ Z_i \sqsubseteq Z_j \tag{3.1}$$

     Since the system $\mathcal{S}_{n+1}$ is obtained as a result of removal of the pair $\langle Z_l, \alpha_l\rangle$ from $\mathcal{S}_n$, all pairs $\langle Z_i, \alpha_i\rangle \in \mathcal{S}_n$, $i \neq l$, are contained in $\mathcal{S}_{n+1}$, i.e., $\langle Z_i, \alpha_i\rangle \in \mathcal{S}_{n+1}$. Having obtained by Property 1 that $Z_i \sqsubseteq Z_i$, we conclude that Equation (3.1) holds.

   - Secondly, we show that

     $$\text{for } Z_l, \alpha_l.\ \langle Z_l, \alpha_l\rangle \in \mathcal{S}_n\ \exists \langle Z_j, \alpha_l\rangle \in \mathcal{S}_{n+1}.\ Z_l \sqsubseteq Z_j \tag{3.2}$$

     Since the system $\mathcal{S}_{n+1}$ is a result of a single application of the rule on Figure 3.2 to the system $\mathcal{S}_n$, there exists a pair $\langle Z_m, \alpha_l\rangle \in \mathcal{S}_n$ such that $Z_l \sqsubseteq Z_m$. Moreover, we find that $\langle Z_m, \alpha_l\rangle \in \mathcal{S}_{n+1}$, because the rule has removed from the system $\mathcal{S}_n$ only the pair $\langle Z_l, \alpha_l\rangle$. Since we know that $\langle Z_m, \alpha_l\rangle \in \mathcal{S}_{n+1}$ and $Z_l \sqsubseteq Z_m$, Equation (3.2) holds.

2. The second condition is specified as Equation (3.3).

   $$\forall Z_j, \alpha_j \text{ such that } \langle Z_j, \alpha_j\rangle \in \mathcal{S}_{n+1}.\ \exists \langle Z_i, \alpha_j\rangle \in \mathcal{S}_n.\ Z_j \sqsubseteq Z_i \tag{3.3}$$

   Since the system $\mathcal{S}_{n+1}$ is obtained as a result of applying the rule from Figure 3.2 to the system $\mathcal{S}_n$ once, more precisely, as a result of removal of the pair $\langle Z_l, \alpha_l\rangle$ from $\mathcal{S}_n$, all pairs $\langle Z_j, \alpha_j\rangle \in \mathcal{S}_{n+1}$ are contained in $\mathcal{S}_n$, i.e. $\langle Z_j, \alpha_j\rangle \in \mathcal{S}_n$. Having obtained by Property 1 that $Z_j \sqsubseteq Z_j$, we conclude that Equation (3.3) holds.

We conclude that both conditions in Definition 14 are satisfied. And hence the system $\mathcal{S}_{n+1}$ is equivalent to the system $\mathcal{S}_n$.

By induction hypothesis and Property 3, we conclude that the system $\mathcal{S}_{n+1}$ is equivalent to the system $\mathcal{S}_0$. Therefore, the system $\mathcal{S}_n$ is equivalent to the system $\mathcal{S}_0$ for an arbitrary natural number n.

□

Theorem 2 *Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be systems such that the system $\mathcal{S}_2$ is obtained as a result of applying the normalization algorithm in Figure 3.3 to the system $\mathcal{S}_1$. Then*

1. *$\mathcal{S}_2$ is equivalent to $\mathcal{S}_1$.*
2. *$\mathcal{S}_2$ is in normal form.*

Proof

1. Since the system $\mathcal{S}_2$ is obtained as a result of applying the normalization algorithm in Figure 3.3 to the system $\mathcal{S}_1$, the system $\mathcal{S}_2$ is a result of the n-times application of the simplification rule in Figure 3.2 to $\mathcal{S}_1$, where n is an arbitrary natural number. Therefore, by Lemma 1 the system $\mathcal{S}_2$ is equivalent to the system $\mathcal{S}_1$.
2. It follows from the structure of the normalization algorithm. The algorithm is designed in such a way that after termination it delivers a normal form of the initial system being normalized. If the system $\mathcal{S}_2$ is not in normal form then the algorithm has not terminated with the system $\mathcal{S}_2$ as a result. Latter contradicts with the precondition of the theorem where it is stated that the system $\mathcal{S}_2$ is a result of applying the normalization algorithm in Figure 3.3 to the system $\mathcal{S}_1$.

□

3.1.4 The Properties of the Normalization Algorithm

Next we will concentrate on the important properties of the normalization algorithm depicted in Figure 3.3. Let $\mathcal{S}$, $\mathcal{S}_1$ and $\mathcal{S}_2$ be systems of *C-states*.

- *Termination:* For every system $\mathcal{S}$, the algorithm terminates in a finite number of steps.
- *Correctness:* The system $\mathcal{S}_2$ which is obtained as a result of applying the algorithm to the system $\mathcal{S}_1$ is equivalent to $\mathcal{S}_1$.
- *Completeness:* The algorithm delivers all normal forms of a system $\mathcal{S}$.
- *Complexity:* The complexity of the algorithm is investigated.

Theorem 3 (Termination) *For every finite system $\mathcal{S} = \{\langle Z_i, \alpha_i \rangle\}$, $1 \leq i \leq n$, where Z_i is a C-state and α_i its numeric value, the normalization algorithm, applied to the system $\mathcal{S}$, terminates in at most n number of steps.*

Proof
In the following we will reason by contradiction. Suppose that there exists a system $\mathcal{S}' = \{\langle Z_j, \alpha_j\rangle, 1 \leq j \leq n'\}$, for which the algorithm does not terminate in n' number of steps. We omit 'at most' here, because if the algorithm does not terminate in n' steps then it does not terminate in any number of steps that is less than n' as well.

Since the algorithm does not terminate in n' number of steps, the simplification rule in Figure 3.2 can be applied once again.

Therefore, the simplification rule has been applied to the initial system $\mathcal{S}'$ $n' + 1$ times and has removed $n' + 1$ pairs from $\mathcal{S}'$ (since every application of the simplification rule removes exactly one pair from a system).

This contradicts with the fact that the system $\mathcal{S}'$ contains n' pairs.

□

Theorem 4 (Correctness) *The normalization algorithm in Figure 3.3 is correct.*

Proof Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be systems such that $\mathcal{S}_2$ is obtained as a result of applying the normalization algorithm to $\mathcal{S}_1$. Since the correctness of the normalization algorithm is interpreted as the equivalence relation on the systems $\mathcal{S}_1$ and $\mathcal{S}_2$, the statement of the theorem follows directly from the first statement of the Theorem 2.

□

Theorem 5 (Uniqueness) *For every system $\mathcal{S}$ of C-states there exists the unique (up to variable renaming) system $\mathcal{S}^N$ that is equivalent to $\mathcal{S}$ and is in normal form.*

Proof The existence of such a system $\mathcal{S}^N$ follows from Theorem 2. What is left to prove is the uniqueness of the system $\mathcal{S}^N$.

In the following, we reason by contradiction. Assume that there exist two systems of *C-states* $\mathcal{S}_1^N$ and $\mathcal{S}_2^N$ that are both in normal forms and equivalent to $\mathcal{S}$ but *not* equal (up to variable renaming) to each other.

Since $\mathcal{S}_1^N$ and $\mathcal{S}_2^N$ are not equal (up to variable renaming), without loss of generality, we assume that there exists a pair $\langle Z_1, \alpha\rangle \in \mathcal{S}_1^N$ such that there exists no such $\langle Z_2, \alpha\rangle \in \mathcal{S}_2^N$ and Z_2 is a variant modulo $AC1$ of Z_1.

By the transitivity of the equivalence relation on systems of *C-states*, the system $\mathcal{S}_1^N$ is equivalent to the system $\mathcal{S}_2^N$.

Hence, there exists a pair $\langle Z_2, \alpha\rangle \in \mathcal{S}_2^N$ such that $Z_1 \sqsubseteq Z_2$, and there exists a pair $\langle Z_1', \alpha\rangle \in \mathcal{S}_1^N$ such that $Z_2 \sqsubseteq Z_1'$.

Then by transitivity property of the subsumption relation we conclude $Z_1 \sqsubseteq Z_1'$.

Since the system $\mathcal{S}_1^N$ is in normal form, $Z_1' = Z_1$. Indeed, if $Z_1' \neq Z_1$ then the simplification rule can be applied to the pairs $\langle Z_1', \alpha \rangle$ and $\langle Z_1, \alpha \rangle$, yielding the contradiction with the fact that the system $\mathcal{S}_1^N$ is in normal form.

Hence, we have

$$\begin{aligned} Z_1 &\sqsubseteq Z_2 \\ Z_2 &\sqsubseteq Z_1 \ , \end{aligned}$$

consequently, by Proposition 4, Z_1 and Z_2 are variants modulo $AC1$. The latter contradicts with the assumption.

□

Theorem 6 (Completeness) *The normalization algorithm in Figure 3.3 is complete.*

Proof Let $\mathcal{S}$ be a system of *C-states*. By Theorem 5, for the system $\mathcal{S}$ there exists the *unique* normal form. Since, by Theorems 2, 3, the algorithm delivers a normal form of a system $\mathcal{S}$, we conclude that the algorithm delivers all possible normal forms of $\mathcal{S}$.

□

Definition 16 *Let F be a fluent term. Then the* ***length of the fluent term*** *F, denoted as $l(F)$, is defined as follows*

- *$F = 1$. Then $l(F) = 0$.*
- *$F = T$, where T is a fluent. Then $l(F) = 1$.*
- *$F = F_1 \circ F_2$, where F_1 and F_2 are fluent terms. Then $l(F) = l(F_1) + l(F_2)$.*

Definition 17 *Let Z be a C-state. Then the* ***length of a*** *C-state Z, denoted as $l(Z)$, is defined as follows*

- *$Z = F$, where F is a fluent term. Then $l(Z) = l(F)$.*
- *$Z = F \circ X$, where F is a fluent term and X is a variable. Then $l(Z) = l(F)$.*

In other words, the length of a *C-state* is a number of fluents in it.

Theorem 7 (Complexity) *For the complexity of the normalization algorithm, denoted as a function $\mu(k, n)$, where $\mathcal{S} = \{\langle Z^i, \alpha^i \rangle, 1 \leq i \leq k\}$ is a system of the C-states Z^i and $n = max(n_1, \ldots, n_k)$, where $n_1, \ldots, n_k$ are the lengths of the C-states Z^i, $1 \leq i \leq k$, holds*

$$\mu(k, n) = \mathcal{O}((k^2 - k + 2)n!) \ .$$

Proof Consider a system $\mathcal{S}$ of *C-states*. Without loss of generality, we assume that all *C-states* are of the form $F \circ X$, where F is a fluent term and X is a variable.

$$\mathcal{S} = \begin{cases} \langle S^1_1 \circ \ldots \circ S^1_{n_1} \circ X^1, \alpha^1 \rangle \\ \langle S^2_1 \circ \ldots \circ S^2_{n_2} \circ X^2, \alpha^2 \rangle \\ \vdots \\ \langle S^k_1 \circ \ldots \circ S^k_{n_k} \circ X^k, \alpha^k \rangle \ , \end{cases}$$

To normalize the system $\mathcal{S}$ we need to apply the rule 1 from Figure 3.2 to every two pairs $\langle Z^1, \alpha^1 \rangle, \langle Z^2, \alpha^2 \rangle \in \mathcal{S}$.

The number of pairs to consider is equal to

$$2\sum_{i=1}^{k-1}(k-i) = 2 + (k-1)k = k^2 - k + 2 \ .$$

Within the subsumption relation we need to solve an $AC1$- matching problem for two fluent terms $S = S_1 \circ \ldots \circ S_n$ and $T = T_1 \circ \ldots \circ T_m$, $n \geq m$. To do so, we would need to perform at most $\frac{n!}{(n-m)!}$ of the elementary comparisons of the fluent terms. An elementary comparison of two fluent terms checks whether one term is an instance of another without taking into account the $AC1$ equational theory.

In a worst case, when $n = max(n_1, \ldots, n_k)$, the overall complexity of the normalization algorithm will be a function $\mu(k, n) = \mathcal{O}((k^2 - k + 2)n!)$.

□

The complexity of the algorithm can be represented in a more general way as

$$\mu(k, n) = P_1(k)\mu_{AC1}(n) \ ,$$

where $P_1(k)$ is polynomial on the parameter k; $\mu_{AC1}(x)$ stands for the complexity of an $AC1$-matching problem for two fluent terms of length x. Since it is known that an $AC1$-matching problem is NP-hard [19, 32], the normalization algorithm is NP-hard as well.

This is very important result because it assures that the complexity of the overall normalization algorithm for *C-states* does *not* go beyond the complexity of its core, an $AC1$-matching.

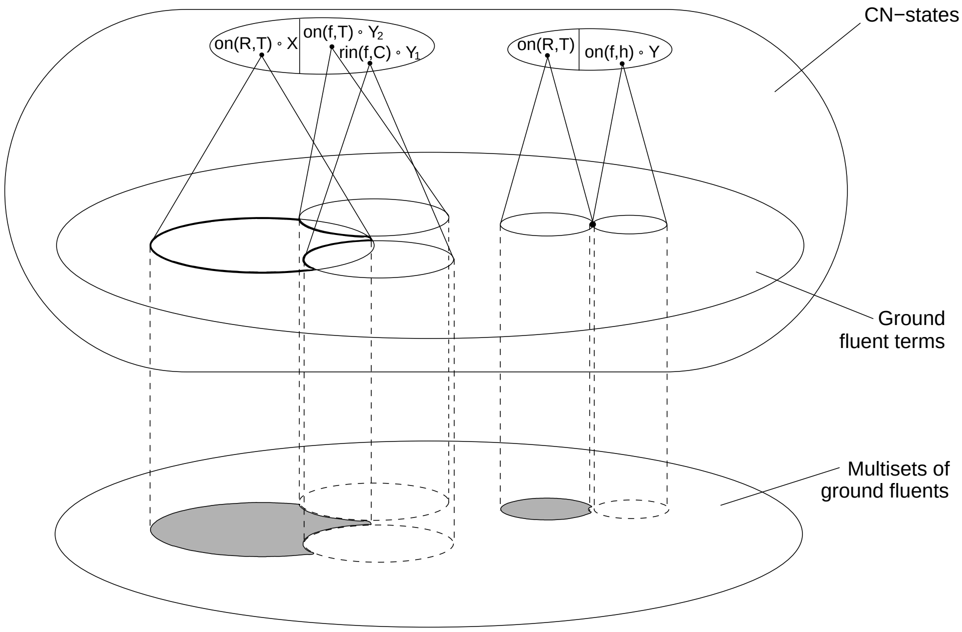

Figure 3.4: *CN-states*: Intuition.

3.2 *CN-states*

3.2.1 The Language

The language of *C-states* $\mathcal{L}_1$ presented in Section 3.1.1 describes the *C-states* that contain conjunction as the only available connective. In other words, the state descriptions, like

$$Z_1 = on(\mathit{ferrari}, T) \circ tin(T, paris) \ ,$$

or

$$Z_2 = on(\mathit{ferrari}, T) \circ X$$

are allowed in $\mathcal{L}_1$. Whereas, the descriptions, specified with the help of a *holds*-macro by stating what fluent terms do not hold in a state, as for instance,

$$\neg holds(on(\mathit{ferrari}, paris), Z_1) \ ,$$

are not allowed in $\mathcal{L}_1$.

Since the usage of negation would add the expressive power to the SDP&FC approach, we would like to extend the language $\mathcal{L}_1$ by negation. As a result, we obtain the language $\mathcal{L}_2$ of extended abstract state descriptions, called *CN-states*, where the letters C and N stand for conjunction and negation symbols, respectively.

Definition 18 *A CN-state Z is described by a pair (P, N), written $Z \triangleq (P, N)$, where P is a C-state of the form $P = P_1 \circ \ldots \circ P_m$ or $P = P_1 \circ \ldots \circ P_m \circ X$ and N is a set of fluent terms $F_1, F_2, \ldots, F_n$ such that*

$$holds(P_1 \circ \ldots \circ P_m, Z) \wedge \neg holds(F_1, Z) \wedge \neg holds(F_2, Z) \wedge \ldots \wedge \neg holds(F_n, Z) \ ,$$

where $P_1, \ldots, P_m$ are fluents and X is a variable.

Please note that if any of $F_1, F_2, \ldots, F_n$ are non-ground fluent terms then all variables occurring in $F_1, F_2, \ldots, F_n$ are universally quantified. For instance, take $F_1 = on(ferrari, T)$. Then the expression $\neg holds(F_1, Z)$ will be interpreted as 'in a *CN-state* Z there is no truck the racing car *ferrari* is on'.

The intuition of *CN-states* is shown in Figure 3.4. Consider a *CN-state* Z_1 described as a pair (P_1, N_1), where

$$P_1 = on(R, T) \circ X$$

and

$$N_1 = \{rin(f, C), on(f, T)\} .$$

One should note that we have omitted the variables Y_1 and Y_2 in the elements of the negative part N_1. The variables Y_1 (Y_2) are used only to reflect the intuitive fact that together with the fluent term $rin(f, C)$ ($on(f, T)$) the fluent terms of the form $rin(f, C) \circ F_1 \circ \ldots \circ F_n$ ($on(f, T) \circ G_1 \circ \ldots \circ G_m$) are negated as well, where $F_1, \ldots, F_n, G_1, \ldots, G_m$ are some fluents.

The positive part of the *CN-state* Z_1 is projected to the set of ground fluent terms. Analogously, the fluent terms $rin(f, C)$ and $on(f, T)$ represent the sets of ground fluent terms.

Since the fluent terms $rin(f, C)$ and $on(f, T)$ belong to the negative part of the *CN-state* Z_1, the respective sets of ground fluent terms should be subtracted from the set of ground fluent terms that corresponds to the positive part P. The resulting set of ground fluent terms is then projected to the set of multisets of ground fluents (the grey area on the picture).

As for the second *CN-state* Z_2 described by a pair (P_2, N_2), where

$$\begin{aligned} P_2 &= on(R, T) \\ N_2 &= \{on(f, h)\} , \end{aligned}$$

the set consisting of only one ground fluent term $on(f, h)$ is subtracted from the set of ground fluent terms that corresponds to the fluent term $on(R, T)$.

3.2.2 The Simplification of Systems

One should note that a system of *CN-states* is defined in similar way as in Section 3.1.2. As in the case of *C-states* we start with defining a subsumption relation on *CN-states*. Since a *CN-state* is described by a pair the second component of which is a set of fluent terms, we need to specify the subsumption relation on sets of fluent terms first. We call this relation *'as set' subsumption relation.*

Definition 19 *Let N_1, N_2 be sets of fluent terms. N_2 is said* ***to subsume as set*** *N_1, written $N_1 \sqsubseteq_s N_2$, iff*

$$\forall F_1 \in N_1. \ \exists F_2 \in N_2. \ \exists \sigma. \ (F_2 \circ X)\sigma =_{AC1} F_1 ,$$

where X is a variable, and σ is a substitution.

As an example, consider two sets of fluent terms N_1 and N_2

$$N_1 = \{tin(honda, paris), tin(honda, monteCarlo)\}$$

$$N_2 = \{tin(honda, C)\} \; .$$

In this case, $N_1 \sqsubseteq_s N_2$ since there exists a variable X_1 and a substitution $\sigma_1 = \{X_1 \mapsto 1, C \mapsto paris\}$ such that

$$(tin(honda, C) \circ X_1) \; \sigma_1 =_{AC1} tin(honda, paris)$$

and there exists a variable X_2 and a substitution

$$\sigma_2 = \{X_2 \mapsto 1, C \mapsto monteCarlo\}$$

such that

$$(tin(honda, C) \circ X_2) \; \sigma_2 =_{AC1} tin(honda, monteCarlo) \; .$$

Hence, for all fluent terms out of N_1 there exists a fluent term out of N_2 that satisfies the condition in Definition 19.

Having defined the subsumption relation on negative parts of *CN-states*, we proceed by defining the subsumption relation for the *CN-states* themselves.

Definition 20 *Let Z_1, Z_2 be CN-states such that $Z_1 \triangleq (P_1, N_1)$ and $Z_2 \triangleq (P_2, N_2)$. Then Z_1 is said* ***to subsume*** *Z_2, written $Z_2 \sqsubseteq Z_1$, iff both conditions are satisfied*

- $P_2 \sqsubseteq P_1$
- $N_1 \sqsubseteq_s N_2$.

The positive parts of the *CN-states* are *C-states*. Therefore, the first condition in Definition 20 will be checked according to the definition of the subsumption relation on *C-states*.

The second condition in Definition 20 needs, however, an additional explanation. Assume that N_2 subsumes N_1. Then, informally speaking, N_2 is more general than N_1. Since N_1 and N_2 correspond to the negative parts of the respective *CN-states* Z_1 and Z_2, the subsumption relation is taken in the reverse order, namely Z_1 subsumes Z_2.

Let us consider an example to illustrate the definition of the subsumption relation on *CN-states*. Take two *CN-states* $Z_1 \triangleq (P_1, N_1)$ with

$$\begin{array}{l} P_1 = on(f, T) \circ tin(T, m) \\ N_1 = \{r \circ rin(f, m)\} \end{array}$$

and $Z_2 \triangleq (P_2, N_2)$ with

$$\begin{array}{l} P_2 = on(f, h) \circ tin(h, m) \\ N_2 = \{r\}, \end{array}$$

where f stands for the racing car ferrari, m for the city Monte-Carlo, r for rain, and h for the truck honda.

According to Definition 13, $P_2 \sqsubseteq P_1$ since there exists a substitution $\sigma = \{T \mapsto h\}$ such that

$$(on(f,T) \circ tin(T,m))\sigma =_{AC1} on(f,h) \circ tin(h,m) \ .$$

Moreover, according to Definition 19 $N_1 \sqsubseteq_s N_2$ since

$$\forall F_1 \in N_1.\ \exists F_2 \in N_2.\ \exists \sigma, X.\ (F_2 \circ X)\sigma =_{AC1} F_1 \ .$$

Or coming back to the example, there exists a variable X and a substitution $\sigma = \{X \mapsto rin(f,m)\}$ such that

$$(r \circ X)\sigma =_{AC1} r \circ rin(f,m).$$

In fact, the obtained result, i.e., $N_1 \sqsubseteq_s N_2$, is desired. N_1 states that the fluent term $r \circ rin(f,m)$ does not occur in the *CN-state* Z_1, whereas, N_2 assures that the fluent r does not hold in Z_2. This informal interpretation for the sets N_1 and N_2 shows that N_2 is more general than N_1, or more formally, that N_2 subsumes N_1.

In some cases, it is easier to use an equivalent definition for the subsumption relation on the negative parts of *CN-states* based on the submultiset relation.

Definition 21 *Let N_1, N_2 be sets of fluent terms.*

$$N_1 \sqsubseteq_s N_2 \ \textit{iff}\ \forall F_1 \in N_1.\ \exists F_2 \in N_2.\ \exists \sigma.\ (F_2\sigma)^I \dot{\subseteq} F_1^I \ ,$$

where $\dot{\subseteq}$ denotes a submultiset relation and σ is a substitution.

Coming back to our example, we would have that $N_1 \sqsubseteq_s N_2$ since

$$\dot{\{} r\sigma \dot{\}} \ \dot{\subseteq} \ \dot{\{} r, rin(f,m) \dot{\}}$$

with σ as an empty substitution.

As another example consider *CN-states* $Z_3 \triangleq (P_3, N_3)$ and $Z_4 \triangleq (P_4, N_4)$, where $P_4 \sqsubseteq P_3$ and

$$N_3 = \{tin(h,p), tin(h,m)\} \ ,$$
$$N_4 = \{tin(h,C)\} \ .$$

In order to show that $N_3 \sqsubseteq_s N_4$, we have to find substitutions σ_1 and σ_2 such that

$$\dot{\{} tin(h,C)\sigma_1 \dot{\}} \ \dot{\subseteq} \ \dot{\{} tin(h,p) \dot{\}}$$

and

$$\dot{\{} tin(h,C)\sigma_2 \dot{\}} \ \dot{\subseteq} \ \dot{\{} tin(h,m) \dot{\}}.$$

The substitutions $\sigma_1 = \{C \mapsto p\}$ and $\sigma_2 = \{C \mapsto m\}$ will solve these submultiset problems.

The examples show that Definition 21 can be used instead of Definition 19 when it is needed. For instance, in our implementation of the normalization algorithm for *CN-states* in Section 4 we make use of Definition 21 because it can be specified in Prolog in a rather straightforward way.

One should observe that the only difference between a *C-state* and a *CN-state* is the presence of the negative part in the latter. Therefore, a *C-state* could be considered as a *CN-state* with an empty negative part, namely a set N contains no elements. Exactly by this reason, in order to compare a *C-state* and a *CN-state* under subsumption, we should use the definition of the subsumption relation on *CN-states*.

Proposition 5 *Let Z_1 and Z_2 be CN-states. Then Z_1 and Z_2 are* ***variants*** *modulo $AC1$ if*

$$Z_2 \sqsubseteq Z_1 \wedge Z_1 \sqsubseteq Z_2 \ .$$

Proof Let $Z_1 \triangleq (P_1, N_1)$ and $Z_2 \triangleq (P_2, N_2)$ be *CN-states*. From

$$Z_2 \sqsubseteq Z_1 \wedge Z_1 \sqsubseteq Z_2$$

it follows, by Proposition 4, that P_1 and P_2 are variants modulo $AC1$. Please recall that P_1 and P_2 are *C-states*. The N-parts N_1 and N_2 are variants modulo $AC1$ by the same proposition since they are sets of fluent terms that are the special case of the *C-states*.

$\square$

Next, we prove that the subsumption relation on *CN-states* is reflexive and transitive.

Property 4 *The subsumption relation on CN-states is reflexive.*

Proof Let $Z \triangleq (P, N)$, where P is a *C-state* and N is a set of fluent terms. By Definition 20, $Z \sqsubseteq Z$ iff

$$P \sqsubseteq P \textit{ and } N \sqsubseteq_s N \ .$$

Since P is a *C-state* by Property 1 $P \sqsubseteq P$. By Definition 19 $N \sqsubseteq_s N$ iff

$$\forall F \in N.\ \exists F \in N.\ \exists \sigma.\ (F \circ X)\sigma =_{AC1} F \ . \tag{3.4}$$

Equation (3.4) holds because there exists a substitution $\sigma = \{X \mapsto 1\}$ such that $(F \circ X)\sigma =_{AC1} F$. So, we obtain that $P \sqsubseteq P$ and $N \sqsubseteq_s N$. Hence, $Z \sqsubseteq Z$.

$\square$

Property 5 *The subsumption relation on CN-states is transitive.*

Proof Let $Z_1 = (P_1, N_1)$, $Z_2 = (P_2, N_2)$, and $Z_3 = (P_3, N_3)$ be the *CN-states*. In order to prove the transitivity property, we will prove that

$$Z_3 \sqsubseteq Z_1 \textit{ iff } Z_2 \sqsubseteq Z_1 \ \wedge \ Z_3 \sqsubseteq Z_2 \ . \tag{3.5}$$

By Definition 20 the Equation (3.5) is transformed into two equations

$$P_3 \sqsubseteq P_1 \textit{ iff } P_2 \sqsubseteq P_1 \ \wedge \ P_3 \sqsubseteq P_2 \tag{3.6}$$

$$N_1 \sqsubseteq_s N_3 \textit{ iff } N_1 \sqsubseteq_s N_2 \ \wedge \ N_2 \sqsubseteq_s N_3 \ . \tag{3.7}$$

Since P_1, P_2, and P_3 are the *C-states*, the Equation (3.6) holds by Property 2.

What is left to prove is the Equation (3.7). By Definition 19 we obtain that $N_1 \sqsubseteq_s N_2$, iff

$$\forall F_1 \in N_1.\ \exists F_2 \in N_2.\ \exists \sigma_1.\ (F_2 \circ X_1)\sigma_1 =_{AC1} F_1 \ ,$$

where X_1 is a variable, and σ_1 is a substitution. And $N_2 \sqsubseteq_s N_3$, iff

$$\forall F_2 \in N_2.\ \exists F_3 \in N_3.\ \exists \sigma_2.\ (F_3 \circ X_2)\sigma_2 =_{AC1} F_2 \ ,$$

where X_2 is a variable, and σ_2 is a substitution. $N_1 \sqsubseteq_s N_3$, iff

$$\forall F_1 \in N_1.\ \exists F_3 \in N_3.\ \exists \sigma_3.\ (F_3 \circ X_3)\sigma_3 =_{AC1} F_1 \ , \tag{3.8}$$

where X_3 is a variable, and σ_3 is a substitution. To satisfy the Equation (3.8) the substitution σ_3 should be taken as

$$\sigma_3 = \sigma_2|_{\textsc{Var}(F_3)}\ \sigma_1|_{\textsc{Var}(F_2)} \cup \{X_3 \mapsto X_2\sigma_2 \circ X_1\sigma_1\} \ ,$$

where $\sigma_1|_{\textsc{Var}(F_2)}$, $\sigma_2|_{\textsc{Var}(F_3)}$ are restrictions of the substitutions σ_1, σ_2 to $\textsc{Var}(F_2)$, $\textsc{Var}(F_3)$, respectively.

$\square$

Due to the fact that the subsumption relation is defined in a very natural and concise way, the simplification rule depicted on Figure 3.2 is valid for *CN-states* as well. Therefore, there is no need to re-define it. In addition, we need to define a simplification rule which simplifies negative parts N (N-components) of *CN-states*. The rules are shown on Figure 3.5.

The rule 1 removes the redundancy of the following kind. If the *CN-states* Z_1, Z_2 have the same numeric value α and Z_1 subsumes Z_2 then the rule 1 removes the pair $\langle Z_2, \alpha\rangle$ because the *CN-state* Z_2 is said to be 'more specific' than Z_1.

The rule 2 operates within one *CN-state*, namely it is applied to the negative part of a *CN-state*. Let Z be a *CN-state* such that $Z \triangleq (P, N)$ and $F_1, F_2 \in N$ are fluent terms. Then if there exists a substitution σ such that $(F_1 \circ X)\sigma =_{AC1} F_2$ then the fluent term F_2 is redundant and ought to be deleted from the negative part N.

Let $\mathcal{S}$ be a system such that $\langle Z_1, \alpha_1\rangle, \langle Z_2, \alpha_2\rangle \in \mathcal{S}$, where Z_1, Z_2 are *CN-states* such that $Z_1 \triangleq (P_1, N_1)$, $Z_2 \triangleq (P_2, N_2)$ and α_1, α_2 are numeric values for Z_1, Z_2, respectively.

1. $$\frac{\langle Z_1, \alpha\rangle \quad \langle Z_2, \alpha\rangle \quad Z_2 \sqsubseteq Z_1}{\langle Z_1, \alpha\rangle}$$

2. Let Z be a *CN-state* such that $Z \triangleq (P, N)$. Then
$$\frac{F_1 \quad F_2 \quad \exists\sigma.(F_1 \circ X)\sigma =_{AC1} F_2}{F_1} \ ,$$
where $F_1, F_2 \in N$ are fluent terms, X is a variable that does not occur in F_1, and σ is a substitution.

Figure 3.5: Simplification rules for *CN-states*.

For instance, consider a *CN-state* $Z \triangleq (P, N)$, where

$$P = on(ferrari, honda)$$

with the intended meaning that the racing car ferrari is on the truck honda, and

$$N = \{rain, rain \circ thunder\}$$

with the understanding that it is not raining and it is not raining together with thundering in the *CN-state* Z.

The fluent term $rain \circ thunder$ is redundant since there exists a substitution $\sigma = \{X \mapsto thunder\}$ such that

$$(rain \circ X)\sigma =_{AC1} rain \circ thunder \ .$$

One should observe that the rule 2 preserves the intended meaning. Namely, if we have already specified the fact that it is not raining in a *CN-state* $Z = (P, N)$, i.e., $rain \in N$, the statement that there is no rain together with thunder in Z, i.e., $rain \circ thunder \in N$, is redundant.

The equivalence relation between the systems over *CN-states* has the same intuition behind and is defined in the same way as for the systems over *C-states*.

Definition 22 *Let S_1 and S_2 be systems. The systems S_1 and S_2 are called* ***equivalent*** *iff both conditions are satisfied:*

1. $\forall Z_i, \alpha_i$ *such that* $\langle Z_i, \alpha_i\rangle \in S_1$. $\exists\langle Z_j, \alpha_i\rangle \in S_2$. $Z_i \sqsubseteq Z_j$;

2. $\forall Z_j, \alpha_j$ *such that* $\langle Z_j, \alpha_j\rangle \in S_2$. $\exists\langle Z_i, \alpha_j\rangle \in S_1$. $Z_j \sqsubseteq Z_i$,

where Z_i, Z_j are CN-states.

Property 6 *The equivalence relation on systems over CN-states is transitive.*

Proof To prove the statement we can use all argumentations from the proof of Property 3 in Section 3.1.2. Only instead of Property 2, we rely on Property 5.

□

3.2.3 The Normalization Algorithm

Since for *CN-states* there were defined two simplification rules, both of them should be reflected in definition of a normal form of a system over *CN-states*.

Definition 23 *Let $\mathcal{S}$ be a system over CN-states. Let $\langle Z_k, \alpha_k \rangle \in \mathcal{S}$ where Z_k is a CN-state such that $Z_k = (P_k, N_k)$, $1 \leq k \leq m$. A system $\mathcal{S}$ is in **normal form** iff the following conditions are satisfied:*

1. *There is no $i \neq j$ such that $Z_j \sqsubseteq Z_i$ and $\alpha_i = \alpha_j$, $1 \leq i \leq m$, $1 \leq j \leq m$.*
2. *For all k there is no $i \neq j$ such that $\exists \sigma.(F_i \circ X)\sigma =_{AC1} F_j$, where $F_i, F_j \in N_k$, $1 \leq i \leq l$, $1 \leq j \leq l$, l is a number of fluent terms in N_k, X is a variable that does not occur in F_i and σ is a substitution.*

Informally speaking, a system $\mathcal{S} = \{\langle Z_k, \alpha_k \rangle\}$, $1 \leq k \leq m$, is in normal form if no rules from Figure 3.5 are applied to $\mathcal{S}$. Therefore, the conditions in Definition 23 specify that neither rule 1 nor rule 2 is applied.

The first condition states that the rule 1 is not applied, i.e., there are no *CN-states* $Z_i, Z_j \in \mathcal{S}$ with the same numeric values such that $Z_j \sqsubseteq Z_i$, where $i \neq j$, $1 \leq i \leq m$, $1 \leq j \leq m$.

Similarly, the second condition formalizes the fact that the rule 2 is not applied, i.e., for all negative parts N_k of the *CN-states* $Z_k = (P_k, N_k)$, $1 \leq k \leq m$, the following condition holds. There is no index i that is not equal to index j, such that $\exists \sigma.(F_i \circ X)\sigma =_{AC1} F_j$.

The normalization algorithm based on the simplification rules given on Figure 3.5 is presented on Figure 3.6. The normalization algorithm takes as an input a system $\mathcal{S}$ and returns a system $\mathcal{S}'$ that is a normal form of the system $\mathcal{S}$. Since the second simplification rule has been introduced for *CN-states*, the proof of this result has to be done explicitly by using the argumentations from the similar proof for *C-states*.

Theorem 8 *Let S_1 and S_2 be systems of CN-states such that S_2 is obtained as a result of applying the normalization algorithm on the Figure 3.6 to S_1. Then*

1. *S_2 is equivalent to S_1.*

`Input:` A system $\mathcal{S}$ such that $\langle Z_i, \alpha_i \rangle \in \mathcal{S}$, where Z_i is a *CN-state*, α_i is a numeric value for Z_i, and $i = 1 \ldots n$.
`Output:` A system $\mathcal{S}'$ in normal form such that $\mathcal{S}'$ is equivalent to the system $\mathcal{S}$.

`while` $\mathcal{S}$ is not in normal form `do`

1. select don't-know non-deterministically a *CN-state* such that the rule 2 from Figure 3.5 is applicable;
2. if such a *CN-state* exists then apply the rule 2 on Figure 3.5 to it;
3. select don't-know non-deterministically two pairs $\langle Z_1, \alpha_1 \rangle$, $\langle Z_2, \alpha_2 \rangle \in \mathcal{S}$ such that the rule 1 from Figure 3.5 is applicable;
4. if such two pairs exist then apply the rule 1 on Figure 3.5 to them;

`end while`
exit with $\mathcal{S}' = \mathcal{S}$.

Figure 3.6: Normalization algorithm for the *CN-states*.

2. S_2 is in normal form.

Proof

1. Since by Property 6 the equivalence relation on the systems over the *CN-states* is transitive, it is enough to prove the statement of the theorem for the one step of the normalization algorithm only.

   First, we need to show that the rule 2 from Figure 3.5 preserves the equivalence relation on the systems over the *CN-states*.

   Suppose that $\langle (P_1, N_1), \alpha_1 \rangle \in \mathcal{S}_1$ and $\langle (P_1, N_1'), \alpha_1 \rangle \in \mathcal{S}_1'$, where N_1' is a result of applying the rule 2 on Figure 3.5 to N_1. Hence, the sets N_1 and N_1' are of the following form

$$N_1 = \{F_1, \ldots, F_l, \ldots, F_k, \ldots, F_n\}$$

$$N_1' = \{F_1, \ldots, F_{l-1}, F_{l+1}, \ldots, F_k, \ldots, F_n\}$$

   and there exists a substitution σ such that $(F_k \circ X)\sigma =_{AC1} F_l$, where X is a variable.

In fact, in order to ensure that the rule 2 from Figure 3.5 preserves the equivalence relation on the systems $\mathcal{S}_1$ and $\mathcal{S}_1'$, we need to show that the below-given conditions are satisfied

(a) If $N' \sqsubseteq_s N_1$ then $N' \sqsubseteq_s N_1'$ for some N'.

(b) If $N'' \sqsubseteq_s N_1'$ then $N'' \sqsubseteq_s N_1$ for some N'' .

We start with the first condition.

(a) By Definition 19 from $N' \sqsubseteq_s N_1$ it follows that

$$\forall F' \in N'.\ \exists F_1 \in N_1.\ \exists \sigma_1.\ (F_1 \circ X_1)\sigma_1 =_{AC1} F' \ , \tag{3.9}$$

where X_1 is a variable. $N' \sqsubseteq_s N_1'$ implies that

$$\forall F' \in N'.\ \exists F_1' \in N_1'.\ \exists \sigma_1'.\ (F_1' \circ X_1')\sigma_1' =_{AC1} F' \ , \tag{3.10}$$

where X_1' is a variable.
Here, we distinguish the two cases

- $F_1 = F_l$. We have that $\exists F_k \in N_1'$ such that

$$\exists \sigma_2.\ (F_k \circ X_2)\sigma_2 =_{AC1} F_l \ , \tag{3.11}$$

where X_2 is a variable. From Equation (3.9) we have that

$$\exists \sigma_1.\ (F_l \circ X_1)\sigma_1 =_{AC1} F'. \tag{3.12}$$

Replacing in Equation (3.10) F_1' with F_k, we obtain

$$\forall F' \in N'.\ \exists F_k \in N_1'.\ \exists \sigma_1'.\ (F_k \circ X_1')\sigma_1' =_{AC1} F' \ . \tag{3.13}$$

In order to satisfy the Equation (3.13), we take

$$\sigma_1' = \sigma_1|_{\textsc{Var}(F_l)}\ \sigma_2|_{\textsc{Var}(F_k)} \cup \{X_1' \mapsto X_1\sigma_1 \circ X_2\sigma_2\} \ ,$$

where $\sigma_1|_{\textsc{Var}(F_l)}$, $\sigma_2|_{\textsc{Var}(F_k)}$ are restrictions of σ_1, σ_2 to $\textsc{Var}(F_l)$, $\textsc{Var}(F_k)$, respectively.

- $F_1 \neq F_l$. Then $F_1 \in N_1'$. Hence, replacing in Equation (3.10) F_1' with F_1, we obtain

$$\forall F' \in N'.\ \exists F_1 \in N_1'.\ \exists \sigma_1'.\ (F_1 \circ X_1')\sigma_1' =_{AC1} F' \ . \tag{3.14}$$

Equation (3.14) holds by Equation (3.9).

(b) By Definition 19 from $N'' \sqsubseteq_s N_1'$ it follows that

$$\forall F'' \in N''.\ \exists F_1' \in N_1'.\ \exists \sigma_1'.\ (F_1' \circ X_1')\sigma_1' =_{AC1} F'' \ , \tag{3.15}$$

where X_1' is a variable. $N'' \sqsubseteq_s N_1$ implies that

$$\forall F'' \in N''.\ \exists F_1 \in N_1.\ \exists \sigma_1.\ (F_1 \circ X_1)\sigma_1 =_{AC1} F'' \ , \tag{3.16}$$

where X_1 is a variable. Since an every fluent term from N_1' is contained in N_1, we obtain that $F_1' \in N_1$. So, we have that Equation (3.16) holds.

Consequently, the rule 2 preserves the equivalence relation on the systems $\mathcal{S}_1$ and $\mathcal{S}'_1$.

Next, we would need to show that the system $\mathcal{S}''_1$ that is obtained via application of the rule 1 from Figure 3.5 to the system $\mathcal{S}'_1$ is equivalent to $\mathcal{S}'_1$. It can be shown in the same way as in Theorem 2.

Thus, by Property 6 the system $\mathcal{S}_2$ is equivalent to the system $\mathcal{S}_1$.

2. It follows from the structure of the normalization algorithm. The normalization algorithm is designed in such a way that after termination it delivers a normal form of the initial system being normalized. If the system $\mathcal{S}_2$ is not in normal form then the algorithm has not terminated with the system $\mathcal{S}_2$ as a result. Latter contradicts with the precondition of the theorem where it is stated that the system $\mathcal{S}_2$ is a result of applying the normalization algorithm on Figure 3.6 to the system $\mathcal{S}_1$.

$\square$

3.2.4 The Properties of the Normalization Algorithm

Next we will concentrate on the important properties of the normalization algorithm depicted in Figure 3.6. In the theorems below, we will call this algorithm 'the normalization algorithm', or 'the algorithm'.

Let $\mathcal{S}$, $\mathcal{S}_1$ and $\mathcal{S}_2$ be systems over *CN-states*.

- *Termination:* For every system $\mathcal{S}$, the normalization algorithm terminates in a finite number of steps.
- *Correctness:* The system $\mathcal{S}_2$ that is obtained as a result of applying the normalization algorithm to the system $\mathcal{S}_1$ is equivalent to $\mathcal{S}_1$.
- *Completeness:* The algorithm delivers all normal forms of a system $\mathcal{S}$.
- *Complexity:* The complexity of the algorithm is investigated.

Theorem 9 (Termination) *Let $\mathcal{S} = \{\langle Z_i, \alpha_i \rangle\}$, $1 \leq i \leq n$, where $Z_i \triangleq (P_i, N_i)$ is a CN-state with a finite number of fluent terms in N_i and α_i is its numeric value, be an arbitrary finite system. Then the normalization algorithm, applied to the system $\mathcal{S}$, terminates in finite number of steps.*

Proof Due to the facts:

1. $\mathcal{S}$ is a finite system and every application of the simplification rule 1 removes exactly one pair from $\mathcal{S}$;
2. all N-parts of the *CN-states* out of the system $\mathcal{S}$ contain a finite number of fluent terms and every application of the simplification rule 2 removes exactly one fluent term from one of the N-parts

we conclude, by the same reasoning as in Theorem 3, that none of the simplification rules in Figure 3.5 can be applied infinitely many times.

Hence, the normalization algorithm terminates in finite number of steps.

□

Theorem 10 (Correctness) *The normalization algorithm is correct.*

Proof Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be systems over *CN-states* such that $\mathcal{S}_2$ is obtained as a result of applying the normalization algorithm in Figure 3.6 to $\mathcal{S}_1$. Since the correctness of the normalization algorithm is interpreted as the equivalence relation on the systems $\mathcal{S}_1$ and $\mathcal{S}_2$, the statement of the theorem follows directly from the first statement of the Theorem 8.

□

Theorem 11 (Uniqueness) *For every system $\mathcal{S}$ of CN-states there exists the unique (up to variable renaming) system $\mathcal{S}^N$ that is equivalent to $\mathcal{S}$ and is in normal form.*

Proof Since the subsumption relation over *CN-states* is transitive and the equivalence on the systems over *CN-states* is transitive too, the same reasoning as in the proof of the Theorem 5 is applied here. The only difference is that instead of Proposition 4, Proposition 5 is employed.

□

Theorem 12 (Completeness) *The normalization algorithm is complete.*

Proof Let $\mathcal{S}$ be a system of *CN-states*. By Theorem 11, for the system $\mathcal{S}$ there exists the *unique* normal form. Since, by Theorems 8, 9, the algorithm delivers a normal form of the system $\mathcal{S}$, we conclude that the algorithm delivers all possible normal forms of $\mathcal{S}$.

□

Theorem 13 (Complexity) *For the complexity of the normalization algorithm, denoted as a function $\mu(k, n, m, l)$, the following property holds*

$$\mu(k, n, m, l) = \mathcal{O}((k^2 - k + 2)n! + m^2(k^2 - k + 2)l! + k(m^2 - m + 2)l!) \ ,$$

where $\mathcal{S} = \{\langle Z^i, \alpha^i \rangle\}$, $1 \leq i \leq k$ is a system of the CN-states $Z^i = (P^i, N^i)$, n is a maximum length of the fluent terms in P^i-parts, m is a maximum number of the fluent terms in the N^i-parts, and l is a maximum length of the fluent terms in the N^i-parts.

Proof Consider a system $\mathcal{S}$ of *CN-states*. Without loss of generality, we assume that all positive parts of the *CN-states* are of the form $F \circ X$, where F is a fluent term and X is a variable.

$$\mathcal{S} = \begin{cases} \langle (S_1^1 \circ \ldots \circ S_{n_1}^1 \circ X^1, N^1), \alpha^1 \rangle \\ \langle (S_1^2 \circ \ldots \circ S_{n_2}^2 \circ X^2, N^2), \alpha^2 \rangle \\ \vdots \\ \langle (S_1^k \circ \ldots \circ S_{n_k}^k \circ X^k, N^k), \alpha^k \rangle \ , \end{cases}$$

where N^i, $1 \leq i \leq k$ are represented as the systems of the fluent terms of the following form

$$N^i = \begin{cases} N_1^i \\ N_2^i \\ \vdots \\ N_{m_i}^i \end{cases}$$

The number of the fluent terms in a system N^i is m_i.

To normalize the system $\mathcal{S}$ we need to apply the rule 2 from Figure 3.5 to the negative parts of all *CN-states* first. According to the rule 2 from Figure 3.5 we need to solve an $AC1$-matching problem for all fluent terms out of N^i. In order to do so, we would need to consider $m_i^2 - m_i + 2$ number of pairs, where m_i is a number of the fluent terms in N^i.

To solve an $AC1$-matching problem for two fluent terms F_1 and F_2 of the lengths l_1 and l_2, respectively, we would need to perform at most $\frac{l_1!}{(l_1-l_2)!}$, $l_1 \geq l_2$ elementary comparisons of the fluent terms. An elementary comparison of two fluent terms checks whether one term is an instance of another without taking into account the $AC1$ equational theory.

Assume that $l_1^1, \ldots, l_{m_1}^1$ are the lengths of the fluent terms $N_1^1, \ldots, N_{m_1}^1$, resp. Similarly, $l_1^k, \ldots, l_{m_k}^k$ are the lengths of the fluent terms $N_1^k, \ldots, N_{m_k}^k$.

In a worst case, where $l = max(l_1^1, \ldots, l_{m_1}^1, \ldots, l_1^k, \ldots, l_{m_k}^k)$, in order to solve an $AC1$-matching problem in the rule 2, we would need to perform

$$\sum_{i=1}^{k} (m_i^2 - m_i + 2) l!$$

elementary comparisons. And when $m = max(m_1, \ldots, m_k)$, we obtain

$$\sum_{i=1}^{k} (m^2 - m + 2) l! = k(m^2 - m + 2) l! \ .$$

Next, we need to perform the further normalization based on the rule 1 from Figure 3.5. So, we need to solve an $AC1$-matching problem for the positive parts as well as for the negative parts of all *CN-states* out of $\mathcal{S}$. Since the positive parts of the *CN-states* are *C-states*, by Theorem 7 we need at most $(k^2 - k + 2)n!$ steps to solve an $AC1$-matching problem for the positive parts, where $n = max(n_1, \ldots, n_k)$ and k is a number of the *CN-states* in $\mathcal{S}$.

To solve an $AC1$-matching problem on the negative parts $N^1, \ldots, N^k$ of the *CN-states*, we would need to perform $m^2(k^2 - k + 2)l!$ steps, where $m = max(m_1, \ldots, m_k)$, k is a number of the *CN-states* in the system $\mathcal{S}$, and $l = max(l_1^1, \ldots, l_{m_1}^1, \ldots, l_1^k, \ldots, l_{m_k}^k)$.

So, altogether we have

$$\mu(k, n, m, l) = \mathcal{O}((k^2 - k + 2)n! + m^2(k^2 - k + 2)l! + k(m^2 - m + 2)l!) \ ,$$

where k is a number of the *CN-states* in the system $\mathcal{S}$, $n = max(n_1, \ldots, n_k)$, $m = max(m_1, \ldots, m_k)$, and $l = max(l_1^1, \ldots, l_{m_1}^1, \ldots, l_1^k, \ldots, l_{m_k}^k)$.

□

The complexity of the algorithm stated in Theorem 13 can be represented in a more general way as

$$\mu(k, n, m, l) = P_1(k)\mu_{AC1}(n) + P_2(k, m)\mu_{AC1}(l) \ ,$$

where $P_1(k)$ and $P_2(k, m)$ are polynomials on the parameters k, m of a system; $\mu_{AC1}(x)$ stands for the complexity of an $AC1$-matching problem for two fluent terms of length x.

Since it is known that an $AC1$-matching problem is NP-hard [19, 32], our normalization algorithm is NP-hard as well.

This is very important result because it assures that the complexity of the overall normalization algorithm for *CN-states* does *not* go beyond the complexity of its core, an $AC1$-matching.

3.2.5 Some Notes on Disjunction within *CN-states*

The language of *CN-states*[1] $\mathcal{L}_2$ can be extended by adding disjunction to the positive as well as negative parts of *CN-states* yielding a new kind of abstract states, called *CND-states* .[2]

Obviously, the normalization algorithm on Figure 3.6 is designed to operate with *CN-states* only. There arises a natural question of whether it is possible to extend or adapt the existing normalization algorithm in such a way that it operates over *CN-states* with disjunction as well.

As a possible solution we could try to convert a system of *CN-states* with disjunction to a system of *CN-states*. To illustrate the conversion technique we will concentrate on a single conversion step only.

The next two cases specify the way how the disjunction symbol can be eliminated.

[1] We omit the consideration of the *C-states*, since the *C-states* are the *CN-states* with the empty set of the fluent terms N.

[2] Please note that we omit the proofs here. The main objective is to investigate the possibility of using the disjunction within the *CN-states*.

1. *The disjunction within the positive part of a CN-state*. A *CN-state* that contains **n** disjunction symbols within its positive part can be rewritten to the system of **n+1** *CN-states* positive parts of which do not contain any disjunction symbols.

$$\langle (F_1 \vee \ldots \vee F_{n+1}, N), \alpha \rangle \rightarrow \begin{cases} \langle (F_1, N), \alpha \rangle \\ \vdots \\ \langle (F_{n+1}, N), \alpha \rangle \end{cases}$$

2. *The disjunction within the negative part of a CN-state*. If a *CN-state* contains **n** disjunction symbols within an element N_1 of its negative part, i.e., $N_1 = F_1 \vee \ldots \vee F_{n+1}$, then N_1 can be rewritten to the set of fluent terms $\{F_1, \ldots, F_{n+1}\}$.

$$\langle (P, \{F_1 \vee \ldots \vee F_{n+1}, G\}), \alpha \rangle \rightarrow \langle (P, \{F_1, \ldots, F_{n+1}, G\}), \alpha \rangle$$

In such a way we propose to eliminate the disjunction within *CN-states*. We believe that the systems over *CN-states* that contain disjunction can be normalized with the help of the normalization algorithm on Figure 3.6 extended by an additional conversion procedure.

Chapter 4

The Aspects of the Implementation

In this section, we intend to provide the description of the methods that have been used to implement the normalization algorithms in Figures 3.3 and 3.6. Since *C-states* can be considered as *CN-states* with the empty negative part, the implementation only for *CN-states* has been designed. The Prolog program is presented in Appendix A.

As a general remark, one should note that the fluent terms are implemented as lists of terms. For instance, the fluent term $on(f,h) \circ tin(h,m)$ is represented as `[on(f,h),tin(h,m)]`.

A system $\mathcal{S}$ is considered as a list of lists of the following kind `[P,value,N]`, where `P` is a positive part of a *CN-state* $Z \triangleq (P,N)$, `value` is a numeric value of Z, and `N` is a negative part of Z. Since by Definition 18 `P` is a *C-state* and, by Definition 11 we know that P is either a fluent term, or is of the form $P = F \circ X$, where F is a fluent term and X is a variable, `P` is represented by a list of terms, and a variable may appear as an element of the list. The second argument of a list `[P,value,N]` is a numeric value associated to a *CN-state* $Z = (P,N)$. The third argument of a list `[P,value,N]` is a negative part N of Z. Since by Definition 18 `N` is a set of fluent terms and every fluent term is a list of terms, `N` is realized as a list of lists. Every sublist represents a fluent term that does not hold in Z.

For instance, consider the system $\mathcal{S} = \dot{\{}\langle Z_1, \alpha_1\rangle, \langle Z_2, \alpha_2\rangle\dot{\}}$, where $\alpha_1 = \alpha_2 = 0$ and

$$\begin{aligned}
Z_1 &= on(f,h) \circ tin(h,m) \circ X \;\wedge\; \neg holds(rin(f,m), Z_1)\;, \\
Z_2 &= on(R,T) \circ tin(T,C) \circ Y \;\wedge \\
&\qquad \neg holds(rin(f,m), Z_1) \;\wedge\; \neg holds(on(f,T) \circ tin(T,m), Z_2)\;.
\end{aligned}$$

In the implementation, the system $\mathcal{S}$ would be realized as the following list

```
S=[ [[on(f,h),tin(h,m),X],0,[[rin(f,m)]]],
    [[on(R,T),tin(T,C),Y],0,[[rin(f,m)],[on(f,T),tin(T,m)]]] ].
```

The predicates `write_list(L)`, `elem(E,T)`, `mult_minus(L,E,L2)`, `mult_minus_set(L1,L2,L3)`, and `has_var(S1,S2)` are auxiliary. The predicate `write_list(L)` performs the element-by-element output of a list `L`. The predicate `elem(E,T)` succeeds when `E` is an element of a list `T`. The removal of exactly one occurrence of an element `E` from a multiset `L` is implemented by the predicate `mult_minus(L,E,L2)`. The predicate `mult_minus_set(L1,L2,L3)` produces a multiset $L_3 = L_1 \setminus L_2$. The predicate `has_var(S1,S2)` checks whether a *C-state* S_1 is of the form $F \circ X$, where F is a fluent term and X is a variable. If $S_1 = F \circ X$, the predicate drops the variable X and creates the *C-state* $S_2 = F$.

The predicate `normalize_all2(S,NS)` delivers in `NS` a normal form of the initial system `S`.

First of all, the predicate `remove1_all(S,S1)` removes all occurrences of an empty fluent term 1 from every *CN-state* out of the initial system `S`. The auxiliary predicates `remove1(P,P2)` and `remove1_list(N,N2)` delete all occurrences of an empty fluent term 1 in the positive and negative parts of a *CN*-state, respectively.

The predicate `normalize_neg_all(S1,S2)` removes from the system `S1` the redundancy with respect to the rule 2 from Figure 3.5. As an output it returns a system `S2` consisting of the *CN-states* the negative parts of which have no two fluent terms F_1, F_2 such that there exists a substitution σ such that $(F_1 \circ X)\sigma =_{AC1} F_2$, or vice versa. Since a non-ground $AC1$-matching problem can be transformed into a submultiset matching problem, then instead of checking whether there exists a substitution σ such that $(F_1 \circ X)\sigma =_{AC1} F_2$, we should solve a question of whether there exists a substitution $\sigma' = \sigma|_{\mathrm{VAR}(F_1)}$ such that $(F_1\sigma')^I \dot{\subseteq} F_2^I$, where $\mathrm{VAR}(F_1)$ is a set of variables in a fluent term F_1, $(F_1\sigma')^I$ and F_2^I are multisets that correspond to the fluent terms $(F_1\sigma')$ and F_2, respectively.

For instance, the query

```
[eclipse2]:
  normalize_neg_all([ [[on(f,t),tin(t,m),r,X],9,[[r],[r,sn],[s]]],
                      [[on(R,T),Z],9,[]] ],S2),
```

where `f` stands for the racing car ferrari, `t` for the truck toyota, `m` for the city Monte-Carlo; `r`, `s`, and `sn` for the fact that it is raining, sunny, and snowing, respectively, would give the answer

```
S2=[ [[on(f,t),tin(t,m),r,X],9,[[r],[s]]],
     [[on(R,T),Z],9,[]] ].
```

The predicate `normalize_all(S2,D)` takes a system `S2` that is a result of the predicate `normalize_neg_all(S1,S2)` and applies the normalization with respect to the rule 1 from Figure 3.5 to `S2`. This predicate returns a list `D` of the *CN-states* that are redundant with respect to the simplification rule 2.

For instance, the query

```
[eclipse 10]:
    normalize_all([ [[on(f,t),tin(t,m),r,X],9,[[r],[s]]],
                    [[on(R,T),Z],9,[]] ],D).
```

would give the answer that the first *CN-state* is redundant, namely

```
D=[ [[on(f,t),tin(t,m),r,X],9,[[r],[s]]] ].
```

The system `NS`, that is a normal form of the initial system `S`, is then obtained by subtraction the list `D` from the system `S2`. As for the previous example, the system `NS` will be specified as

```
NS=[ [[on(R,T),Z],9,[]] ].
```

To implement the rule 1 on Figure 3.5, and consequently, the subsumption relation on *CN-states* $Z_1 \triangleq (P_1, N_1)$, $Z_2 \triangleq (P_2, N_2)$, by Definition 20 we need to satisfy both conditions, namely

- $P_2 \sqsubseteq P_1$,
- $N_1 \sqsubseteq_s N_2$.

Since P_1 and P_2 are *C-states* the predicate `ac1_match(A,B)` implements the subsumption relation over *C-states*. The predicate `sN_all(L1,L2)` realizes the second condition of the Definition 20.

As it has been already mentioned in Section 2.1 the subsumption relation over *C-states* can be translated to a non-ground $AC1$-matching problem. The predicate `ac1_match(A,B)` is designed to solve a non-ground $AC1$-matching problem for the fluent terms `A` and `B`. The predicate `ac1_match(A,B)` succeeds if and only if there exists a substitution σ such that $(A \circ X)\sigma =_{AC1} B$, where A and B are fluent terms.

Since a non-ground $AC1$-matching problem can be transformed into a submultiset matching problem, it was decided to reformulate the task of an $AC1$-matching to a submultiset matching problem specified in Definition10. Moreover, Eclipse Prolog 4.2.3, that is used here, has a very useful built-in predicate `instance(S,T)` that succeeds iff a term `S` is an instance of a term `T`.

The submultiset matching problem is solved with the help of the predicate `sub_multi_set_match(A,T,Ti,Ai)`. This predicate succeeds iff there is a solution for the problem: 'knowing that a term A_i is an instance of a term A, whether there exist elements $B_{i_1}, \ldots, B_{i_n}$ in T_i such that $A_i + [B_{i_1}, \ldots, B_{i_n}]$ is an instance of $A + T$, where $+$ stands for the lists concatenation, or not'.

Having defined the predicate `sub_multi_set_match(A,T,Ti,Ai)` in this way, we have ensured that the fluent terms, as e.g., $F_1 = on(ferrari, T) \circ tin(T, monteCarlo)$ and $F_2 = on(ferrari, honda) \circ tin(toyota, monteCarlo)$ are not related under subsumption.

For these two terms the predicate `sub_multi_set_match/4` will be called as `sub_multi_set_match([],F1,F2,[])`. The predicate starts with searching for a fluent in F_2 that could be an instance for the first fluent in F_1, i.e., $\mathit{on}(\mathit{ferrari}, T)$. The fluent $\mathit{on}(\mathit{ferrari}, \mathit{honda})$ is an instance of the fluent $\mathit{on}(\mathit{ferrari}, T)$, since there exists a substitution $\sigma = \{T \mapsto \mathit{honda}\}$ such that

$$\dot{\{}\mathit{on}(\mathit{ferrari}, T)\sigma\dot{\}} \dot{\subseteq} \dot{\{}\mathit{on}(\mathit{ferrari}, \mathit{honda})\dot{\}} \ .$$

Then the predicate takes the next fluent in F_1, i.e., $\mathit{tin}(T, \mathit{monteCarlo})$, and checks whether there exists a fluent f in F_2 such that the fluent term

$$\mathit{on}(\mathit{ferrari}, \mathit{honda}) \circ f$$

is an instance of the fluent term

$$\mathit{on}(\mathit{ferrari}, T) \circ \mathit{tin}(T, \mathit{monteCarlo}) \ .$$

There exists no such fluent f, hence, there exists no substitution σ such that $(F_1\sigma)^I \dot{\subseteq} F_2^I$.

The subsumption relation for the *C-states* $Z_1,\ Z_2$ of the form $F \circ X$, where F is a fluent term and X is a variable, can be expressed with the help of the submultiset relation and hence, implemented by the predicate `ac1_match/2` directly. But in case when the *C-states* $Z_1,\ Z_2$ are represented as fluent terms $F_1,\ F_2$, the subsumption relation can not be implemented by the predicate `ac1_match/2` only. Here, we rely on the following property

$$(F_1\sigma)^I \doteq F_2^I \ \mathit{iff}\ (F_1\sigma)^I \dot{\subseteq} F_2^I \ \mathit{and}\ \mid F_1^I \mid = \mid F_2^I \mid \ ,$$

where F_1 and F_2 are fluent terms and σ is a substitution. This property allows to apply the predicate `ac1_match/2` in its original form with additional condition that the cardinalities of the multisets $F_1^I,\ F_2^I$ should be equal. Therefore, the implementation of the subsumption relation on the *C-states* as fluent terms is presented as an extra predicate `instance_ac1(A,B)`.

If the condition, that the cardinalities of the multisets $F_1^I,\ F_2^I$ are equal, would not be specified then it would be possible to relate under the subsumption such fluent terms as

$$F_1 = \mathit{on}(f, h) \circ \mathit{tin}(h, m) \circ \mathit{rain}$$

and

$$F_2 = \mathit{on}(f, T) \circ \mathit{tin}(T, m) \ .$$

In other words, the predicate `ac1_match(F_2,F_1)` would succeed with the substitution $\sigma = \{T \mapsto h, X \mapsto \mathit{rain}\}$ such that $(F_2 \circ X)\sigma =_{AC1} F_1$. And consequently, $F_1 \sqsubseteq F_2$.

But intuitively, the fluent terms F_1 and F_2 are not related under the subsumption relation. Taking $F_1' = \mathit{on}(f, h) \circ \mathit{tin}(h, m)$ instead of F_1 would satisfy the condition that $\mid F_1^I \mid = \mid F_2^I \mid$.

The above reasoning implies that in order to decide whether $F_1 \sqsubseteq F_2$, where F_1 and F_2 are fluent terms, the

condition that $\exists\sigma.\ (F_2 \circ X)\sigma =_{AC1} F_1$ is necessary but not sufficient.

Please recall that we need to give an implementation of the second condition in Definition 20, namely $N_1 \sqsubseteq_s N_2$. It will be done by the predicate `sN_all(N1,N2)` that implements the Definition 21 in a straightforward way. The predicate `sN_all(N1,N2)` succeeds iff for all elements F_1 of the list `N1` there exists an element F_2 in a list $N2$ such that there exists a substitution σ such that $(F_2\sigma)^I \dot{\subseteq} F_1^I$.

Finally, to illustrate the normalization algorithm we present some systems of the *CN-states* and the results that the algorithm delivers. As a first example, consider the list `S1`

```
S1=[ [[on(f,T),tin(T,m),X1],0,[[rin(F,M)]]]
     [[on(R,T),tin(T,C),rin(f,m),X2],10,[]]
     [[on(R,T),tin(T,C),X3],0,[[rin(F,M)],[on(f,T),tin(T, m)]]]
     [[on(R,T),tin(T,C),rin(f,m),X4],10,[]]
     [[on(R,T),tin(T,C),X5],0,[[rin(f,m)],
                                [rin(F,M)],[rin(f,m),rain]]] ]
```

that represents the system $\mathcal{S}_1$

$$\begin{cases} \langle Z_1 = on(f,T) \circ tin(T,m) \circ X_1 \ \wedge \ \forall F, M.\ \neg holds(rin(F,M), Z_1), 0\rangle \\ \langle Z_2 = on(R,T) \circ tin(T,C) \circ rin(f,m) \circ X_2, 10\rangle \\ \langle Z_3 = on(R,T) \circ tin(T,C) \circ X_3 \ \wedge \ \forall F, M.\ \neg holds(rin(F,M), Z_3) \ \wedge \\ \qquad \neg holds(on(f,T) \circ tin(T,m), Z_3), 0\rangle \\ \langle Z_4 = on(R,T) \circ tin(T,C) \circ rin(f,m) \circ X_4, 10\rangle \\ \langle Z_5 = on(R,T) \circ tin(T,C) \circ X_5 \wedge \ \neg holds(rin(f,m), Z_5) \ \wedge \\ \qquad \forall F, M.\ \neg holds(rin(F,M), Z_5) \ \wedge \ \neg holds(rin(f,m) \circ rain, Z_5), 0\rangle \ . \end{cases}$$

The system $\mathcal{S}_1$ has both forms of redundancy, therefore it should be normalized with the help of the normalization algorithm on Figure 3.6. As a result the algorithm returns the following normalized system `S2`

```
S2=[ [[on(R,T),tin(T,C),rin(f,m),X4],10,[]]
     [[on(R,T),tin(T,C),X5],0,[[rin(F,M)]]] ]
```

that represents the system $\mathcal{S}_2$

$$\begin{cases} \langle Z_4 = on(R,T) \circ tin(T,C) \circ rin(f,m) \circ X_4, 10\rangle \\ \langle Z_5 = on(R,T) \circ tin(T,C) \circ X_5 \wedge \ \forall F, M.\ \neg holds(rin(F,M), Z_5), 0\rangle \ . \end{cases}$$

Please note that the normalization algorithm is able to treat the variables in the negative parts of the *CN-states* in the intended way. In the above-given example the fluent term $rin(F,M)$ within the negative part of the *CN-state* Z_5 has been defined as the most general one.

We have executed the implementation of our normalization algorithm on the systems of *C-states* as well as *CN-states* for the logistics example that has been first introduced in Section 2.2. In Appendix B some tests on the systems over *C-states* as well as *CN-states* are presented.

In order to answer the question of whether the complexity of the implementation exceeds the worst-case complexity or not, we have tried to build the abstract state descriptions in such a way that they force the Prolog-interpreter to perform maximum number of computational steps. In particular, the Prolog-interpreter was forced to call the *instance*-predicate (as the elementary computational step) maximum number of times.

We have compared the computational behavior of the program with the worst-case complexity for the systems with $k = n = l = m = 2, 3, 4$, where k is a number of *C-states* (*CN-states*, resp.) in a system; n is a maximal length of the *C-states* (maximal length of the P-parts of the *CN-states*); l is a maximal number of the fluent terms in the N-parts of the *CN-states*; m is a maximal length of the fluent terms in the N-parts of the *CN-states*.

So, we intended to invent (by hand) possibly the most complex permutations inside and between the fluent terms within the *C-states* (*CN-states*, resp.) in order to increase the computational requirements of the $AC1$-matching routine on the fluent terms.

As a result of our investigation, we have obtained that the complexity of the implementation does not exceed the worst-case complexity, as expected.

One should note that the analysis of the computational behavior of our program is limited only to the cases, where $k = n = l = m = 2, 3, 4$, because it is rather difficult to construct *by hand* the 'complex enough' permutations for cases, where $k = n = l = m > 4$.

Our analysis should be considered as a preliminary investigation towards the computational behavior, the tests on the systems of larger size need to be accomplished. Moreover, it would be interesting to design the larger examples for the domains like the office domain, and test our approach on them.

Chapter 5

Related Work

We start with comparing the ideas underlying the simplification process in SDP&SC with the ideas of the normalization procedure in SDP&FC. Since the subsumption relation in Description Logics(DLs) has the same intuition as the one in our normalization approach, in the second paragraph, we briefly sketch some algorithms for solving subsumption problem in DLs and determine their applicability for our normalization tasks.

Symbolic Dynamic Programming within Situation Calculus Symbolic Dynamic Programming within Situation Calculus, SDP&SC for short, see [8], relies on the logical simplification of abstract states descriptions. But in comparison to Symbolic Dynamic Programming within the Fluent Calculus, SDP&FC for short, the computation of an optimal plan depends on the logical simplification of *first-order formulae*. Boutilier et al. in their work in [8] operate with so called case statements that define the value function. These case statements are just an abbreviation for the first-order formula of the following kind $\bigvee_{i \leq n}\{\phi_i \wedge t = t_i\}$, where ϕ_i is a logical description of an abstract state and t_i is its numeric value. So, in fact, the simplification, that is involved in SDP&SC, is the simplification of the first-order formulae.

The main goal of the simplification process, that is called 'normalization', in SDP&FC is to deliver the system in normal form. Such a system is supposed to be free of redundancies, namely, all redundant abstract states have been removed from a system via a normalization. We call an abstract state redundant if it is more specific than another one.

SDP&SC does not provide the means to detect which abstract state is more specific (hence, redundant) between two abstract states. Therefore, this allows us to conclude that the way of simplifying the abstract state descriptions in [8] is not applicable for solving the simplification tasks in SDP&FC. So, even if there exists (as the authors claim) an implementation of the logical simplification in SDP&SC, it can not be employed to reduce a system of abstract states into its normal form in SDP&FC, because SDP&SC and SDP&FC rely on the simplification of a different nature.

A Glance at Description Logics Since the subsumption relation in DLs has the same intuition as in our normalization approach, the notions of DLs can be potentially used to determine which abstract state is more general than another one within SDP&FC.

In order to ensure a reasonable behavior of a DL system, a subsumption problem for the DL, underlying the system, should be at least decidable, and preferably of low complexity. Consequently, the expressive power of the DL must be restricted in an appropriate way. This trade-off between the expressivity of the DLs and the complexity of their inference problems is the crucial issue in DL research [2]. For an excellent introduction to the basic DLs see [1].

The original DL system KL-ONE [10] as well as its successors, e.g., BACK [26], K-REP [23], LOOM [22], employ so-called structural subsumption algorithms to solve subsumption between concept descriptions. The main idea of these algorithms is as follows. First, concept descriptions are turned into certain normal forms. Second, subsumption between concept descriptions is tested by recursively comparing the corresponding normal forms (see e.g., [24] for a description of such an algorithm).

The structural subsumption algorithms are quite efficient (polynomial) but with the disadvantage that they are incomplete for expressive DLs, i.e., not all subsumption relations can be detected. Later, it has been shown that already rather inexpressive DLs cannot have polynomial subsumption algorithms [11, 25, 30], and that the language underlying the first DL-system, KL-ONE, has an undecidable subsumption problem [30]. In particular, these results showed the incompleteness of the (polynomial) structural subsumption algorithms.

The one way how the researchers have tried to overcome this dilemma was to restrict the DL underlying a DL system. It was done, e.g., in CLASSICAL system that is based on the $\mathcal{ALNS}$-language. It was shown in [20] that for the $\mathcal{ALNS}$-concept descriptions deciding subsumption is a polynomial time problem.

When one wants to extend the DL by disjunction and/or full negation, the structural subsumption algorithms become incomplete. Recently, tableau-based algorithms have turned out to be useful: they are sound and complete and often of optimal (worst-case) complexity [2].

The first such algorithm has been proposed in [31] for a DL that they called $\mathcal{ALC}$. Since $\mathcal{ALC}$ is a syntactic variant of propositional multi-modal logic K_n, a result obtained by Ladner [21], i.e., that the satisfiability in K_n is PSPACE-complete, holds also in $\mathcal{ALC}$.

The language of *C-states* as well as the language of *CN-states* can be viewed as a subset of the $\mathcal{ALNS}$-language. Thus, one of the structural subsumption algorithms for the $\mathcal{ALNS}$-language can be employed to solve subsumption within SDP&FC.

We have devised our own algorithm for solving subsumption, because whenever we extend our language of abstract states by disjunction and full negation, structural subsumption algorithms become incomplete.

As for our normalization algorithm, it is not yet known whether it remains complete after extending the language by additional connectives, i.e., disjunction, full negation, or not. If our algorithm looses its completeness property via extensions, then one of the existing tableau-based algorithms [31, 18] can be employed to solve subsumption problem within SDP&FC.

Chapter 6

Conclusions and Future Work

This project introduces a solid and concise approach for mechanizing the normalization process of the systems obtained as a result of a decision-theoretic regression step within the first-order VIA in Section 2.3.3. These systems contain the logical descriptions of abstract states along with their numeric values. Indeed, mechanizing the normalization process of such systems allows to shrink the abstract state space by removing redundancies, thereby improving the computational requirements of the first-order VIA.

We have provided the normalization algorithms for two various classes of systems: the systems of *C-states*, that allow only for conjunction within a logical description; and the systems of *CN-states*, a logical description of which is enhanced by negation, in addition to conjunction.

The normalization algorithm designed for systems over *C-states* is similar to the one for *CN-states*, in particular, the latter represents the enhanced version of the first one. Therefore, for this chapter, we use the notion of 'the normalization algorithm' referring to the normalization algorithm for *CN-states*. In addition, all results and properties for the normalization algorithm for *CN-states* are valid for the normalization algorithm for *C-states* too.

The normalization algorithm delivers a normal form of a system of abstract states by removing redundancies. We have investigated that for systems over *C-states* there exists only one kind of redundancy. Namely, for two pairs $\langle Z_1, \alpha_1 \rangle$ and $\langle Z_2, \alpha_2 \rangle$, where Z_1, Z_2 are *C-states* and α_1, α_2 are their numeric values, having assumed that $\alpha_1 = \alpha_2$, a *C-state* Z_2 is said to be redundant iff it is more specific than a *C-state* Z_1. In order to cope with such kind of redundancy, we have defined a kind of generalization relation, called subsumption relation, that allows to determine which abstract state is more general than another one.

As one possibility, the subsumption relation defined as a subset relation on abstract states that are represented by sets of ground fluent terms has been studied. In other words, in this approach logical descriptions of abstract states are always translated into sets of ground fluent terms first. On the one hand, the set representation of abstract states allows to reflect the intuition of subsumption relation in a straightforward way. Indeed, recalling that abstract states are represented as sets of ground fluent terms, one abstract state is said to be more specific if it is a subset of another abstract state. On the other hand, this approach relies on the obligatory domain grounding which certainly obviates the intention of the first-order value iteration algorithm, namely to avoid the explicit enumeration of the domain individuals.

As an alternative possibility, an $AC1$-matching [32] can be taken as a kernel of subsumption relation. Elaborating on this alternative, we have concluded that an $AC1$-matching indeed provides a solid representation of subsumption relation following the intuition of the latter. An $AC1$-matching grants a possibility to consider the logical descriptions of abstract states alone without their rigorous translation into sets of ground fluent terms. Thereby, in contrast to the previous alternative, the problem of explicit domain grounding is avoided. Moreover, an $AC1$-matching problem has been extensively studied, e.g., in [32]. The correct, complete and terminating algorithm for solving an $AC1$-matching problem has been developed there.

Having these pieces in mind, we have opted for the second alternative, namely we have defined the subsumption relation in terms of an $AC1$-matching.

On the basis of subsumption relation, that is a core of the whole normalization process, we have designed a simplification rule that serves as a rewrite rule. Namely, having determined with the help of the subsumption relation which abstract state is more specific than another one, a simplification rule simply removes this abstract state together with its numeric value from a system. The simplification rule had been designed in such a way that its shape is independent from the class of abstract states, i.e., the rule is specified in the same way for *C-states* as well as for *CN-states*. The only change that has been done is adaption of the definition of subsumption from *C-states* to *CN-states*.

As for the systems over *CN-states*, we have derived that the kind of redundancy discussed above can be applied to *CN-states* as well because a *CN-state* is described by a pair (P, N), where P is a *C-state*. In addition, the redundancy within the negative part N of a *CN-state* has been treated. Such redundancy takes place only if the negative part of a *CN-state* contains two fluent terms that can be compared with respect to subsumption relation. E.g., a negative part represented by a set of fluent terms $rain$ and $rain \circ thunder$ contains redundancies because the fluent term $rain \circ thunder$ provides superfluous information. A simplification rule that treats such kind of redundancy has been developed.

The normalization algorithm presented in our work is correct, viz., a system obtained as a result of applying the algorithm to an initial system is equivalent to the initial system.

Moreover, our normalization algorithm terminates in a finite number of steps.

The worst-case complexity analysis has shown that our normalization algorithm is NP-hard. This is a crucial result since our algorithm remains in the same complexity class as an $AC1$-matching underlying the subsumption relation ($AC1$-matching is NP-hard as well [19, 32]).

Along with the theoretical framework of the normalization algorithm, we have designed its Prolog-implementation. The implementation in Appendix A relies mainly on the conversion property of an $AC1$-matching problem to a submultiset matching problem.

An analysis of the computational behavior of our Prolog-program has shown that the complexity of the implementation does not exceed the worst-case complexity, as expected.

To our knowledge, the normalization algorithms presented in this work or any other normalization techniques within recently developed symbolic dynamic programming approaches [8, 9, 13] have not been encountered in the literature to date.

6.1 Future Work

This project devised the mechanization technique for the normalization process of the systems obtained as a result of a decision-theoretic regression step within the first-order VIA. Apparently, the latter lacks the formal proof of the fact that it delivers indeed the optimal value function and consequently, an optimal policy. In order to assure the integrity of the first-order VIA we would need to provide such a proof in the future.

The simplification rules underlying the normalization algorithm can be extended by introducing additional kinds of redundancy. E.g., the current simplification rules do not yet capture the inconsistent abstract state descriptions of the following kind. A *CN-state* described by a pair (P, N) is said to be inconsistent if there exists a fluent term that holds in its P- as well as in N-part. One should be aware of the fact that accompanying the approach by any new components such as, e.g., simplification rules or new classes of abstract states, all previous results and properties must be carefully checked for the extended approach as well.

In Section 3.2.5, we have elaborated on the usage of disjunction within *CN-states*. Of course, the formal specification of *CND-states*, the normal forms of the systems over *CND-states*, together with an extension of the normalization algorithm for *CN-states* with respect to *CND-states* are still missing.

A very promising investigation is indeed the design of the complete first-order VIA's implementation. The regression of abstract states based on the $AC1$-unification is still left to be accomplished. The regression step of the first-order VIA appeals to an additional refinement of the *causes*-predicate and, consequently, the $AC1$-unification procedure, with respect to negation and disjunction.

Moreover, one needs to take care of the so called verification procedure too. In order to use our first-order VIA in the real world applications a relationship between the abstract states and the real world states has to be established. After the first-order VIA has minimally partitioned the state space into the abstract states along with their optimal value functions, an agent tries to verify its own state in the real world with one of the abstract states. The entailment relation seems to be the best candidate to solve such a verification task. More precisely, a state Δ in the real world entails a logical description Z of an abstract state, in symbols $\Delta \models Z$, iff Δ is a model for Z. More careful investigation on such semantic issues as 'entailment relation' and 'model' is left to be performed.

As another particularly interesting direction of the future work, could be the further detailed analysis of the DLs algorithms for solving subsumption.

Appendix A

An Implementation

```
write_list([]):-nl.
write_list([A|T]):-write(A),nl,write_list(T).

elem(B,[B|_]).
elem(B,[_|T]):-elem(B,T).

ac1_match(A,B):-sub_multi_set_match([],A,B,[]).

sub_multi_set_match(_,[],_,_):-!.
sub_multi_set_match(_,_,[],_):-false,!.
sub_multi_set_match(A,[H|T],Ti,Ai):-
    elem(Bi,Ti),
    append(A,[H],NewA),
    append(Ai,[Bi],NewAi),
    instance(NewAi,NewA),
    mult_minus(Ti,Bi,Ti2),
    sub_multi_set_match(NewA,T,Ti2,NewAi).

instance_ac1(A,B):-
    length(A,La),
    length(B,Lb),
    La is Lb,
    ac1_match(B,A).

mult_minus([],_,[]).
mult_minus([E1|R],E,R):-
    E==E1,!.
mult_minus([E1|R1],E,[E1|R2]):-
    mult_minus(R1,E,R2).

mult_minus_set(L,[],L).
mult_minus_set(L,[A|T],L3):-
```

```
    mult_minus(L,A,L2),
    mult_minus_set(L2,T,L3).

remove1_all([],[]).
remove1_all([Z|T],[Z2|T2]):-
    remove1_state(Z,Z2),
    remove1_all(T,T2).

remove1_state([P,V,N],[P2,V,N2]):-
    remove1(P,P2),
    remove1_list(N,N2).

remove1([],[]).
remove1([Z|T],T2):-
    Z==1,
    !,
    remove1(T,T2).
remove1([Z|T],[Z|T2]):-
    remove1(T,T2).

remove1_list([],[]).
remove1_list([Z|T],[Z2|T2]):-
    remove1(Z,Z2),
    remove1_list(T,T2).

normalize_all2(S,NS):-
    write('Initial system:'),
    nl,
    write_list(S),
    remove1_all(S,S1),
    normalize_neg_all(S1,S2),
    normalize_all(S2,D),
    mult_minus_set(S2,D,NS).

normalize_neg_all([],[]).
normalize_neg_all([Z|T],[Z2|T2]):-
    normalize_neg(Z,Z2),
    normalize_neg_all(T,T2).

normalize_neg([P,V,N],[P,V,N2]):-
    redundants(N,[],N3),
    mult_minus_set(N,N3,N2).

redundants([],_,[]):-!.
redundants([E|L1],L,[E|L2]):-
```

```
    append(L1,L,LL),
    has_subset(E,LL),
    !,
    redundants(L1,L,L2).
redundants([E|L1],L,L2):-
    redundants(L1,[E|L],L2).

has_subset(_,[]):-fail.
has_subset(N,[N2|_]):- ac1_match(N2,N),!.
has_subset(N,[_|T]):-has_subset(N,T).

normalize_all([],[]).
normalize_all([A|T],P):-
    normalize(A,T,P1,T2),
    normalize_all(T2,P2),
    append(P1,P2,P).

%Case 0: V1=\=V2
normalize([A,V1,N1],[[B,V2,N2]|T],P,[[B,V2,N2]|T2]):-
    V1=\=V2,
    !,
    normalize([A,V1,N1],T,P,T2).

%Case 1: Z1=S*X, Z2=T*Y
normalize([A,V,N1],[[B,V,N2]|T],[[A,V,N1]],[[B,V,N2]|T]):-
    has_var(A,A1),
    has_var(B,B1),
    ac1_match(B1,A1),
    sN_all(N2,N1),
    !.

normalize([A,V,N1],[[B,V,N2]|T],P,T2):-
    has_var(A,A1),
    has_var(B,B1),
    ac1_match(A1,B1),
    sN_all(N1,N2),
    !,
    normalize([A,V,N1],T,P1,T2),
    append([[B,V,N2]],P1,P).


normalize([A,V,N1],[[B,V,N2]|T],P,[[B,V,N2]|T2]):-
    has_var(A,_),
    has_var(B,_),
    normalize([A,V,N1],T,P,T2).
```

```
%Case 2: Z1=S, Z2=T*X
normalize([A,V,N1],[[B,V,N2]|T],[[A,V,N1]],[[B,V,N2]|T]):-
    \+has_var(A,_),
    has_var(B,B1),
    ac1_match(B1,A),
    sN_all(N2,N1),
    !.

normalize([A,V,N1],[[B,V,N2]|T],P,T2):-
    has_var(A,A1),
    \+has_var(B,_),
    ac1_match(A1,B),
    sN_all(N1,N2),
    !,
    normalize([A,V,N1],T,P1,T2),
    append([[B,V,N2]],P1,P).

normalize([A,V,N1],[[B,V,N2]|T],P,[[B,V,N2]|T2]):-
    has_var(A,_),
    \+has_var(B,_),
    normalize([A,V,N1],T,P,T2).

normalize([A,V,N1],[[B,V,N2]|T],P,[[B,V,N2]|T2]):-
    \+has_var(A,_),
    has_var(B,_),
    normalize([A,V,N1],T,P,T2).

%Case 3:Z1=S,Z2=T
normalize([A,V,N1],[[B,V,N2]|T],[[A,V,N1]],[[B,V,N2]|T]):-
    \+has_var(A,_),
    \+has_var(B,_),
    instance_ac1(A,B),
    sN_all(N2,N1),
    !.

normalize([A,V,N1],[[B,V,N2]|T],P,T2):-
    \+has_var(A,_),
    \+has_var(B,_),
    instance_ac1(B,A),
    sN_all(N1,N2),
    !,
    normalize([A,V,N1],T,P1,T2),
    append([[B,V,N2]],P1,P).
```

```
normalize([A,V,N1],[[B,V,N2]|T],P,[[B,V,N2]|T2]):-
    \+has_var(A,_),
    \+has_var(B,_),
    !,
    normalize([A,V,N1],T,P,T2).

normalize(_,T,[],T).

has_var([A|T],T):-var(A),!.
has_var([A|T],[A|P]):-has_var(T,P).

sN_all([],_).
sN_all([T|H],List2):-sN(T,List2),sN_all(H,List2).

sN(_,[]):-false.
sN(Elem,[H|_]):-sub_multi_set_match([],H,Elem,[]),!.
sN(Elem,[_|T]):-sN(Elem,T).
```

Appendix B

Some tests

Initial system in Prolog	`[[on(f,t),tin(t,m),X],9,[]]` `[[on(f,t),tin(t,m),Y],9,[]]`
Initial system description	$\langle Z_1 = on(f,t) \circ tin(t,m) \circ X, 9 \rangle$ $\langle Z_2 = on(f,t) \circ tin(t,m) \circ Y, 9 \rangle$
Normalized system in Prolog	`[[on(f,t),tin(t,m),Y],9,[]]`
Normalized system description	$\langle Z_2 = on(f,t) \circ tin(t,m) \circ Y, 9 \rangle$

Table B.1: Some tests: test 1.

Initial system in Prolog	`[[on(f,T),tin(T,m),X],9,[]]` `[[on(f,t),tin(t,m),Y],9,[]]`
Initial system description	$\langle Z_1 = on(f,T) \circ tin(T,m) \circ X, 9 \rangle$ $\langle Z_2 = on(f,t) \circ tin(t,m) \circ Y, 9 \rangle$
Normalized system in Prolog	`[[on(f,T),tin(T,m),X],9,[]]`
Normalized system description	$\langle Z_2 = on(f,T) \circ tin(T,m) \circ X, 9 \rangle$

Table B.2: Some tests: test 2.

Initial system in Prolog	`[[on(f,T),tin(T,m),X],9,[[rain]]]` `[[on(f,t),tin(t,m),Y],9,[]]`
Initial system description	$\langle Z_1 = on(f,T) \circ tin(T,m) \circ X \wedge \neg holds(rain, Z_1), 9\rangle$ $\langle Z_2 = on(f,t) \circ tin(t,m) \circ Y, 9\rangle$
Normalized system in Prolog	`[[on(f,T),tin(T,m),X],9,[[rain]]]` `[[on(f,t),tin(t,m),Y],9,[]]`
Normalized system description	$\langle Z_1 = on(f,T) \circ tin(T,m) \circ X \wedge \neg holds(rain, Z_1), 9\rangle$ $\langle Z_2 = on(f,t) \circ tin(t,m) \circ Y, 9\rangle$

Table B.3: Some tests: test a2.

Initial system in Prolog	`[[on(f,T),tin(T,m),X],9,[]]` `[[on(f,t),tin(t,m),Y],9,[[rain]]]`
Initial system description	$\langle Z_1 = on(f,T) \circ tin(T,m) \circ X, 9\rangle$ $\langle Z_2 = on(f,t) \circ tin(t,m) \circ Y \wedge \neg holds(rain, Z_2), 9\rangle$
Normalized system in Prolog	`[[on(f,T),tin(T,m),X],9,[]]`
Normalized system description	$\langle Z_1 = on(f,T) \circ tin(T,m) \circ X, 9\rangle$

Table B.4: Some tests: test b2.

Initial system in Prolog	`[[on(f,T)],9,[]]` `[[on(R,h)],9,[]]`
Initial system description	$\langle Z_1 = on(f,T), 9\rangle$ $\langle Z_2 = on(R,h), 9\rangle$
Normalized system in Prolog	`[[on(f,T)],9,[]]` `[[on(R,h)],9,[]]`
Normalized system description	$\langle Z_1 = on(f,T), 9\rangle$ $\langle Z_2 = on(R,h), 9\rangle$

Table B.5: Some tests: test c1.

Initial system in Prolog	`[[on(f,T),tin(T,m),X],9,[]]` `[[on(f,t),tin(h,m),Y],9,[]]`
Initial system description	$\langle Z_1 = on(f,T) \circ tin(T,m) \circ X, 9\rangle$ $\langle Z_2 = on(f,t) \circ tin(h,m) \circ Y, 9\rangle$
Normalized system in Prolog	`[[on(f,T),tin(T,m),X],9,[]]` `[[on(f,t),tin(h,m),Y],9,[]]`
Normalized system description	$\langle Z_1 = on(f,T) \circ tin(T,m) \circ X, 9\rangle$ $\langle Z_2 = on(f,t) \circ tin(h,m) \circ Y, 9\rangle$

Table B.6: Some tests: test d2.

Initial system in Prolog	`[[on(f,t),tin(t,m),X],9,[[rain]]]` `[[on(f,T),tin(T,m),Y],9,[[rain,snow]]]`
Initial system description	$\langle Z_1 = on(f,t) \circ tin(t,m) \circ X \wedge$ $\neg holds(rain, Z_1), 9\rangle$ $\langle Z_2 = on(f,T) \circ tin(T,m) \circ Y \wedge$ $\neg holds(rain \circ snow, Z_2), 9\rangle$
Normalized system in Prolog	`[[on(f,T),tin(T,m),Y],9,[[rain,snow]]]`
Normalized system description	$\langle Z_2 = on(f,T) \circ tin(T,m) \circ Y \wedge$ $\neg holds(rain \circ snow, Z_2), 9\rangle$

Table B.7: Some tests: test 3.

Initial system in Prolog	`[[on(f,t),tin(t,m),X],9,[]]` `[[on(f,T),tin(T,m),rain,Y],9,[]]`
Initial system description	$\langle Z_1 = on(f,t) \circ tin(t,m) \circ X, 9\rangle$ $\langle Z_2 = on(f,T) \circ tin(T,m) \circ rain \circ Y, 9\rangle$
Normalized system in Prolog	`[[on(f,t),tin(t,m),X],9,[]]` `[[on(f,T),tin(T,m),rain,Y],9,[]]`
Normalized system description	$\langle Z_1 = on(f,t) \circ tin(t,m) \circ X, 9\rangle$ $\langle Z_2 = on(f,T) \circ tin(T,m) \circ rain \circ Y, 9\rangle$

Table B.8: Some tests: test 4.

Initial system in Prolog	`[[on(f,t),tin(t,m),rain,X],9,[]]` `[[on(f,T),tin(T,m),Y],9,[]]`
Initial system description	$\langle Z_1 = on(f,t) \circ tin(t,m) \circ rain \circ X, 9\rangle$ $\langle Z_2 = on(f,T) \circ tin(T,m) \circ Y, 9\rangle$
Normalized system in Prolog	`[[on(f,T),tin(T,m),Y],9,[]]`
Normalized system description	$\langle Z_2 = on(f,T) \circ tin(T,m) \circ Y, 9\rangle$

Table B.9: Some tests: test 5.

Initial system in Prolog	`[[on(f,t),tin(t,m),rain,X],9,[]]`
	`[[on(f,T),tin(T,m),Y],9,[]]`
	`[[on(R,T),Z],10,[]]`

Initial system description	$\langle Z_1 = on(f,t) \circ tin(t,m) \circ rain \circ X, 9\rangle$ $\langle Z_2 = on(f,T) \circ tin(T,m) \circ Y, 9\rangle$ $\langle Z_3 = on(R,T) \circ Z, 10\rangle$
Normalized system in Prolog	`[[on(f,T),tin(T,m),Y],9,[]]` `[[on(R,T),Z],10,[]]`
Normalized system description	$\langle Z_2 = on(f,T) \circ tin(T,m) \circ Y, 9\rangle$ $\langle Z_3 = on(R,T) \circ Z, 10\rangle$

Table B.10: Some tests: test 6.

Initial system in Prolog	`[[on(f,t)],0,[[rain],[rain,snow],`
	`[snow],[sun],`
	`[rain,sun],[rain,snow,sun]]]`

Initial system description	$\langle Z_1 = on(f,t) \ \wedge \ \neg holds(rain, Z_1) \wedge$ $\neg holds(rain \circ snow, Z_1) \wedge$ $\neg holds(snow, Z_1) \wedge$ $\neg holds(sun, Z_1) \wedge$ $\neg holds(rain \circ sun, Z_1) \wedge$ $\neg holds(rain \circ snow \circ sun, Z_1), 0\rangle$
Normalized system in Prolog	`[[on(f,t)],0,[[rain],[snow],[sun]]]`
Normalized system description	$\langle Z_1 = on(f,t) \ \wedge \ \neg holds(rain, Z_1) \wedge$ $\neg holds(snow, Z_1) \wedge$ $\neg holds(sun, Z_1), 0\rangle$

Table B.11: Some tests: test 23.

Initial system in Prolog	`[[on(f,T1),tin(T1,m),X1],0,[[rin(f,m)]]]`
	`[[on(R2,T2),tin(T2,C2),rin(f,m),X2],10,[]]`
	`[[on(R3,T3),tin(T3,C3),X3],0,`
	`     [[rin(f,m)],[on(f,T3),tin(T3,m)]]]`
	`[[on(R4,T4),tin(T4,C4),rin(f,m),X4],10,[]]`
	`[[on(R5,T5),tin(T5,C5),X5],0,`
	`     [[rin(f,m)],[rin(f,m),rain]]]`

Initial system description	$\langle Z_1 = on(f, T_1) \circ tin(T_1, m) \circ X_1 \wedge$ $\quad \neg holds(rin(f, m), Z_1), 0\rangle$ $\langle Z_2 = on(R_2, T_2) \circ tin(T_2, C_2) \circ rin(f, m) \circ X_2, 10\rangle$ $\langle Z_3 = on(R_3, T_3) \circ tin(T_3, C_3) \circ X_3 \wedge$ $\quad \neg holds(rin(f, m), Z_3) \wedge$ $\quad \neg holds(on(f, T_3) \circ tin(T_3, m), Z_3), 0\rangle$ $\langle Z_4 = on(R_4, T_4) \circ tin(T_4, C_4) \circ rin(f, m) \circ X_4, 10\rangle$ $\langle Z_5 = on(R_5, T_5) \circ tin(T_5, C_5) \circ X_5 \wedge$ $\quad \neg holds(rin(f, m), Z_5) \wedge$ $\quad \neg holds(rin(f, m) \circ rain, Z_5), 0\rangle$
Normalized system in Prolog	`[[on(R4,T4),tin(T4,C4),rin(f,m),X4],10,[]]` `[[on(R5,T5),tin(T5,C5),X5],0,[[rin(f,m)]]]`
Normalized system description	$\langle Z_4 = on(R_4, T_4) \circ tin(T_4, C_4) \circ rin(f, m) \circ X_4, 10\rangle$ $\langle Z_5 = on(R_5, T_5) \circ tin(T_5, C_5) \circ X_5 \wedge$ $\quad \neg holds(rin(f, m), Z_5), 0\rangle$

Table B.12: Some tests: test 141.

Initial system	`[[on(f,T1),tin(T1,m),X1],0,[[rin(F,M)]]]`
	`[[on(R2,T2),tin(T2,C2),rin(f,m),X2],10,[]]`
	`[[on(R3,T3),tin(T3,C3),X3],0,`
	`[[rin(F,M)],[on(f,T3),tin(T3,m)]]]`
	`[[on(R4,T4),tin(T4,C4),rin(f,m),X4],10,[]]`
	`[[on(R5,T5),tin(T5,C5),X5],0,[[rin(f,m)],`
	`[rin(F,M)],[rin(f,m),rain]]]`

Initial system	$\langle Z_1 = on(f, T_1) \circ tin(T_1, m) \circ X_1 \wedge$ $\forall F, M.\ \neg holds(rin(F, M), Z_1), 0\rangle$ $\langle Z_2 = on(R_2, T_2) \circ tin(T_2, C_2) \circ rin(f, m) \circ X_2, 10\rangle$ $\langle Z_3 = on(R_3, T_3) \circ tin(T_3, C_3) \circ X_3 \wedge$ $\forall F, M.\ \neg holds(rin(F, M), Z_3) \wedge$ $\neg holds(on(f, T_3) \circ tin(T_3, m), Z_3), 0\rangle$ $\langle Z_4 = on(R_4, T_4) \circ tin(T_4, C_4) \circ rin(f, m) \circ X_4, 10\rangle$ $\langle Z_5 = on(R_5, T_5) \circ tin(T_5, C_5) \circ X_5 \wedge$ $\neg holds(rin(f, m), Z_5) \wedge$ $\forall F, M.\ \neg holds(rin(F, M), Z_5) \wedge$ $\neg holds(rin(f, m) \circ rain, Z_5), 0\rangle$

Normalized system	`[[on(R4,T4),tin(T4,C4),rin(f,m),X4],10,[]]`
	`[[on(R5,T5),tin(T5,C5),X5],0,[[rin(f,m)],`
	`[rin(F,M)],[rin(f,m),rain]]]`

Normalized system	$\langle Z_4 = on(R_4, T_4) \circ tin(T_4, C_4) \circ rin(f, m) \circ X_4, 10\rangle$ $\langle Z_5 = on(R_5, T_5) \circ tin(T_5, C_5) \circ X_5 \wedge$ $\neg holds(rin(f, m), Z_5) \wedge$ $\forall F, M.\ \neg holds(rin(F, M), Z_5) \wedge$ $\neg holds(rin(f, m) \circ rain, Z_5), 0\rangle$

Table B.13: Some tests: test 142.

Appendix C

Statement

Hiermit versichere ich, daß die vorliegende Diplomarbeit von mir selbständig verfaßt wurde und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt wurden.

Dresden, the Olga Skvortsova

Bibliography

[1] F. Baader and W. Nutt. Basic description logics. In Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors, *The Description Logic Handbook: Theory, Implementation, and Applications*, pages 43–95. Cambridge University Press, 2003.

[2] F. Baader and U. Sattler. An overview of tableau algorithms for description logics. *Studia Logica*, 69:5–40, 2001.

[3] A. G. Barto, S. J. Bradtke, and S. P Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72(1-2):81–138, 1995.

[4] R. E. Bellman. *Adaptive Control Processes: A Guided Tour.* Princeton University Press, Princeton, NJ, USA, 1961.

[5] C. Boutilier, T. Dean, and S. Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999.

[6] C. Boutilier and R. Dearden. Using abstracts for decision-theoretic planning with time constraints. In *Proceedings of the 12th National Conference on Artificial Intelligence*, pages 1016–1022, Seattle, WA, 1994.

[7] C. Boutilier, R. Dearden, and M. Goldszmidt. Exploiting structure in policy construction. In *Proceedings of the 14th International Conference on Artificial Intelligence*, pages 1104–1111, Montreal, Canada, 1995.

[8] C. Boutilier, R. Reiter, and B. Price. Symbolic dynamic programming for first-order MDPs. In Bernhard Nebel, editor, *Proceedings of the Seventeenth International Conference on Artificial Intelligence (IJCAI-01)*, pages 690–700. Morgan Kaufmann, 2001.

[9] C. Boutilier, R. Reiter, M. Soutchanski, and S. Thrun. Decision-theoretic, high-level agent programming in the situation calculus. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-00)*, 2000.

[10] R. J. Brachman and J. G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216, 1985.

[11] R. J. Brachmann and H. J. Levesque. The tractability of subsumption in frame-based description languages. In *Proceedings of the 4th Nat. Conference on Artificial Intelligence (AAAI'84)*, pages 34–37. AAAI Press, 1984.

[12] T. Dean, J. Allen, and Y. Aloimonos. *Artificial Intelligence: Theory and Practice*. Benjamin Cummings, 1995.

[13] A. Großmann, S. Hölldobler, and O. Skvortsova. Symbolic dynamic programming within the fluent calculus. In *Proceedings of the International Conference on Artificial and Computational Intelligence (ACI-2002)*, Tokyo, Japan, September 2002.

[14] S. Hanks, S. Russel, and W. Wellman, editors. *Decision Theoretic Planning: Proceedings of the AAAI Spring Symposium*, Menlo Park, CA, USA, 1998. AAAI Press.

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

[16] S. Hölldobler and H.-P. Störr. Solving the entailment problem in the fluent calculus with binary decision diagrams. In *Proceedings of the First International Conference on Computational Logic*, pages 747–761, 2000.

[17] S. Hölldobler and M. Thielscher. Computing change and specificity with equational logic programs. *Annals of Mathematics and Artificial Intelligence*, 14:99–133, 1995.

[18] B. Hollunder and F. Baader. Qualifying number restrictions in concept languages. In *Proceedings of the 2nd International Conference on the Principles of Knowledge Representation and Reasoning (KR91)*, pages 335–346. Morgan Kaufmann, 1991.

[19] D. Kapur and P. Narendran. NP-completeness of the set unification and matching-problems. In *Proceedings of the 8th International Conference on Automated Deduction(CADE)*, pages 489–495, Oxford, July 1986.

[20] R. Küsters. *Non-standard inferences in Description Logics*. PhD thesis, RWTH-Aachen, 2000.

[21] R.E. Ladner. The computational complexity of provability in systems of modal propositional logic. *SIAM Journal of Computing*, 6(3):467–480, 1977.

[22] R. MacGregor. *The evolving technology of classification-based knowledge representation systems*, pages 385–400. Morgan Kaufmann, 1991.

[23] E. Mays, R. Dionne, and R. Weida. K-REP system overview. *SIGART Bulletin*, 2(3), 1991.

[24] B. Nebel. *Reasoning and Revision in Hybrid Representation Systems*, volume 422 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin, Heidelberg, New York, 1990.

[25] B. Nebel. Terminological reasoning is inherently intractable. *Artificial Intelligence*, 43:235–249, 1990.

[26] C. Peltason. The BACK system – an overview. *SIGART Bulletin*, 2(3):114–119, 1991.

[27] R. Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, San Diego, CA, 1991.

[28] R. Reiter. Proving properties of states in the situation calculus. *Artificial Intelligence*, 64:337–351, 1993.

[29] S. Russell and P. Norvig. *Artificial intelligence: A modern approach*. Prentice Hall, 1995.

[30] M. Schmidt-Schauß. Subsumption in KL-ONE is undecidable. In *Proceedings of the 1st International Conference on the Principles of Knowledge Representation and Reasoning (KR'89)*, pages 421–431. Morgan Kaufmann, 1989.

[31] M. Schmidt-Schauß and G Smolka. Attribute concept descriptions with complements. *Artificial Intelligence*, 48(1):1–26, 1991.

[32] M. Thielscher. AC1-unifikation in der linearen logischen programmierung. TASSO-Report 42, Intellektik, Informatik, Darmstadt, Germany, October 1992. (In German).

[33] M. Thielscher. Introduction to the fluent calculus. *Electronic Transactions on Artificial Intelligence*, 2(3-4):179–192, 1998.

[34] M. Thielscher. Programming of reasoning and planning agents with FLUX. In D. Fensel, D. McGuinness, and M.-A. Williams, editors, *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 435–446, Toulouse, France, April 2002. Morgan Kaufmann.