ASYNCHRONOUS MULTI-AGENT REASONING IN THE SITUATION CALCULUS

Ryan Francis Kelly

Submitted in total fulfilment of the requirements of the degree of Doctor of Philosophy

October 2008

Department of Computer Science and Software Engineering

The University of Melbourne

Abstract

This thesis develops several powerful extensions to the situation calculus for reasoning about multi-agent teams in asynchronous domains. We enrich the situation calculus with an explicit representation of the local perspective of each agent, upon which we then construct: a partially-ordered representation of actions for planning the joint execution of a shared task; a theory of individual-level knowledge that allows agents to consider arbitrarily-long sequences of hidden actions; and a formalism for group-level epistemic modalities that allows common knowledge to be reasoned about using a regression rule.

Planning in the situation calculus typically involves building a fully-ordered sequence of the actions to be performed, requiring constant synchronisation between agents if the plan is to be carried out cooperatively. We develop a partially-ordered representation of actions which we call a *joint execution.* These structures allow independent actions to be performed independently while ensuring that, when necessary, synchronisation can be achieved based on the local observations of each agent. Joint executions can be reasoned about using standard situation calculus techniques, allowing them to easily replace raw situation terms during planning.

Existing accounts of knowledge in the situation calculus assume that everyone always knows how many actions have occurred, demanding significant synchronicity among agents. In asynchronous domains agents must instead consider arbitrarily-long sequences of hidden actions, which cannot be reasoned about effectively using existing techniques. We develop a new reasoning technique called the *persistence condition* operator to augment the standard regression operator, and use it to build a new account of individual-level knowledge that correctly accounts for hidden actions while retaining an effective reasoning procedure. Our formalism allows agents to reason directly about their own knowledge using only their local information.

Common knowledge is traditionally modelled using an explicit second-order axiom, which precludes regression as an effective reasoning technique. We formulate a more powerful language of group-level knowledge using an epistemic interpretation of dynamic logic, and develop a regression rule that is sound and complete for this language. As a consequence, our formalism allows common knowledge to be reasoned about effectively using standard regression techniques.

The end result is a more robust and flexible account of knowledge and action in the situation calculus, suitable both for reasoning about, and for planning in, asynchronous multi-agent domains.

Declaration

This is to certify that:

(i) the thesis comprises only my original work towards the PhD except where indicated in the Preface,

(ii) due acknowledgement has been made in the text to all other material used,

(iii) the thesis is less than 100,000 words in length, exclusive of tables, maps, bibliographies, and appendices.

Ryan Francis Kelly

Preface

During the course of this research, a number of public presentations have been made which are based on the work presented in this thesis. They are listed here for reference.

- Ryan F. Kelly and Adrian R. Pearce. Towards High-Level Programming for Distributed Problem Solving. In *Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT'06)*, pages 490–497, 2006
- Ryan F. Kelly and Adrian R. Pearce. Property Persistence in the Situation Calculus. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 1948–1953, 2007
- Ryan F. Kelly and Adrian R. Pearce. Knowledge and Observations in the Situation Calculus. In *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'07)*, pages 841–843, 2007
- Ryan F. Kelly and Adrian R. Pearce. Complex Epistemic Modalities in the Situation Calculus. In *Proceedings of the 11th International Conference on Principles of Knowledge Representation and Reasoning (KR'08)*, pages 611–620, 2008

Acknowledgements

First, my heartfelt thanks to my supervisor, Adrian Pearce, for his support and guidance throughout this research: for constantly reminding me to remember the bigger picture; for encouraging me to explore any number of wild tangents; and for asking the deep questions about how they all join back together. Thank you also for trusting me to work in my own time, in my own way, and for always remaining confident that we would get there in the end.

To my advisory committee, Leon and James, thank you for providing the clear heads needed to help keep the overall story in focus. It seems like I entered each of our progress meetings a little unsure about how all the pieces fit together, but left each with a renewed focus and a clear picture of the road ahead.

I must also acknowledge the generous support of the University of Melbourne and the Australian Federal Government in providing my Australian Postgraduate Award scholarship, without which this research would not have been possible.

To my fellow students and colleagues at the University of Melbourne – particularly my office-mates Alan, Terence, Michelle, Arif, and my CSSEPG accomplices Alauddin, Kapil, Archana, Andrea, Macros – thanks for making the whole postgrad experience so rewarding. My thanks also to Sebastian Sardiña for his valuable feedback and many helpful discussions.

For my parents Don and Judy, my brother Matt, and my grandparents Val and George, thank you for your unwavering support – financial, physical and emotional – throughout all of my studies. Despite the constant jokes about me being ready for retirement by the time I graduate (those *were* just jokes...right?) I have always known that I could count on your unconditional love and support.

Finally, and of course most importantly, to my wife Lauren: thank you for your unending love and understanding; for your tolerance for the vagaries of the PhD lifestyle; for keeping me firmly grounded in the real world; and for simply being so excited about the whole endeavour.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

The situation calculus, along with the programming language Golog that is built upon it, is a powerful formalism for reasoning about dynamic worlds and specifying the behaviour of autonomous agents. It combines a rich language for expressing domain features with techniques for effective reasoning and a straightforward implementation using logic programming. But while powerful, the situation calculus currently suffers some major limitations that make it unsuitable for reasoning about asynchronous multi-agent domains.

We begin by noting that there is a significant body of work on modelling and implementing multi-agent systems in the situation calculus, including: the cognitive agent specification language and verification environment [110]; theories of coordination and ability [35]; entries in AI competitions such as robot soccer [27] and robot rescue [26]; reasoning about the epistemic feasibility of plans [56]; analysing multi-player games [7]; and the cooperative execution of Golog programs [27, 44]. This literature shows the power and flexibility of the situation calculus, but also highlights three current weaknesses when working with rich multi-agent domains.

First, each of these works has been limited to *synchronous* domains – domains in which each agent's local perspective on the world is updated in lock-step with the global perspective available to the system designer. This restriction ensures that, while an agent may not know the full state of the world, it will always know precisely how many actions have been performed. The agents can therefore perform effective automated reasoning using *regression*, a process that depends on systematically removing action terms from a query. In many cases this synchronicity restriction is enforced using a blanket assumption that all actions are publicly observable.

Second, the fundamental unit of reasoning, and the output of the Golog execution planning process, is the *situation*: a complete, ordered history of all actions that

are to be performed. Having to execute a totally-ordered sequence of actions is far from ideal in a multi-agent setting, as it requires constant synchronisation between the agents. While this may be acceptable for systems that are already restricted to synchronous domains, it does not take advantage of the concurrency inherent in a multi-agent team. In asynchronous domains the required synchronisation of actions may be impossible to achieve, and a partially-ordered representation must be used instead. But there have been no formal accounts of such a representation in the existing situation calculus literature.

Third, the situation calculus lacks a comprehensive treatment of group-level epistemic modalities such as *common knowledge*, which are fundamental to coordination in multi-agent domains. While common knowledge can easily be modelled using an explicit second-order axiom, this precludes the use of regression for effective automated reasoning. In synchronous domains agents can often coordinate their activities without reasoning explicitly about common knowledge, but in more general multi-agent domains the lack of an effective reasoning procedure for common knowledge can be a serious shortcoming.

In each of these cases, the problem is not in modelling asynchronous multi-agent domains, but in combining such rich domain models with effective reasoning techniques. The standard technique for effective reasoning in the situation calculus, and the core of the Golog execution planning procedure, is *regression.* It operates by systematically removing action terms from a query until it is in a form that is easy to answer. The traditional restriction of the situation calculus to synchronous domains allows all agents to know the number of actions performed, so they can directly use existing regression-based techniques for reasoning and planning.

This thesis lifts that restriction by developing reasoning and planning techniques for *asynchronous* domains – domains in which the state of the world can change without updating the local perspective of each agent. We begin by explicitly representing this local perspective: whenever an action occurs, each agent makes a corresponding set of *observations* that are local to that agent. By allowing the set of observations to be empty, we formalise the case where asynchronicity means some actions are completely hidden from some agents.

On top of this seemingly simple extension, we construct a planning process based on partially-ordered action sequences, a new technique for effective inductive reasoning to augment standard regression, and a principled axiomatisation of both individual and group-level knowledge coupled with an effective reasoning procedure. These contributions greatly extend the reach of the situation calculus, enabling its use for specifying, simulating, and implementing more realistic multi-agent systems.

1.1 Motivation

To make things more concrete, let us introduce two motivating examples that will be used throughout the thesis, along with an overview of the challenges they pose to existing techniques from the situation calculus literature.

Example 1: The Cooking Agents

Cathy is hosting a dinner party. A brilliant engineer but a mediocre cook, she has built a team of robotic chefs to help her prepare the meal, and must now program them to carry out their duties. She needs a powerful formalism with which the agents can plan their actions, and a programming language flexible enough to specify the major steps in each recipe while leaving the precise details of execution for the agents to plan amongst themselves. Moreover, she wants to specify the tasks to be performed as a single shared program, and have the agents automatically distribute the work amongst themselves in such a way that they can operate independently where possible and synchronise their actions only when necessary.

The situation calculus offers a compelling approach for this example domain: each recipe can be represented as a Golog program, and the agents can cooperate to plan and perform the concurrent execution of these shared programs. However, existing Golog implementations generate raw situation terms as the output of their planning process. These are fully-ordered sequences of the actions to be performed, requiring constant synchronisation if the agents are to execute them cooperatively.

In synchronous domains this is not a problem, as the agents will always know how many steps of execution have already been performed, and thus what actions should be performed next. But it is still desirable to take advantage of the natural concurrency present when planning for a team of agents. The Golog planning process must be modified to reason about and allow such concurrency.

In asynchronous domains, an agent may not necessarily know how far execution has progressed, and may thus be unsure when or if to perform its next action. The planning process should account for this by only calling for agents to perform an action if they will know, based on their local information, that the action should be performed. But the situation calculus currently has no means of representing this kind of partially-ordered action structure, or of determining whether it is executable.

Rather than demand that Cathy equip her robots with a global synchronisation mechanism of some kind, we will extend the situation calculus and the semantics of Golog execution planning to better handle concurrency and inter-agent synchronisation in asynchronous, partially-observable domains.

Example 2: The Party Invitation

Alice and Bob have heard about Cathy's party but don't know where it is, and have just received an invitation telling them the location. Having suffered decades of trouble with their communications, they like to reason about each other's knowledge by directly observing each other's actions. If Alice reads the invitation, she will know the location of the party. But will Bob know that she knows this? What if he temporarily leaves the room, meaning Alice is able to read the invitation in secret? And most importantly, can they achieve common knowledge of the party's location in order to coordinate their travel plans for the evening?

The situation calculus permits an elegant axiomatisation of this domain, but its standard account of knowledge uses regression for effective reasoning. Existing regression techniques cannot handle two important aspects of this example.

First, the standard account of knowledge requires that whenever an action occurs, all agents *know* that an action has occurred. In domains such as this example, where some actions may be completely hidden, each agent must also establish that its knowledge will *persist* after the occurrence of arbitrarily-many hidden actions. Formulating this requirement involves a second-order induction axiom, which precludes the use of regression for effective automated reasoning.

Second, the situation calculus lacks a comprehensive treatment of group-level epistemic modalities such as common knowledge. The difficulty here is that the standard group-level knowledge operators are not powerful enough to express the regression of common knowledge. Instead, common knowledge is typically handled using a separate second-order definition, again precluding the use of regression for effective automated reasoning.

Rather than demand that Alice and Bob perform open-ended second-order theorem proving, we will develop a new technique for inductive reasoning in the situation calculus, use it to formalise an account of knowledge in the face of hidden actions, and provide a new formulation of group-level epistemic modalities which is powerful enough to capture a regression rule for common knowledge.

1.2 Contributions

As a launching point for our investigations we develop a new multi-agent variant of Golog, with which a team of agents can plan the cooperative execution of a shared task. The language, dubbed *MIndiGolog*, is implemented on the Mozart programming platform, using its powerful distributed logic programming capabil-

ities to share the planning workload between all members of the team. But this initial implementation is limited to synchronous, full-observable domains due to the underlying reasoning machinery of the situation calculus.

To extend the approach to asynchronous domains, we construct an explicit representation of the local *observations* made by each agent when an action is performed. We introduce a function $Obs(agt, a, s)$ returning the set of observations made by an agent when action a is performed in situation s. Each situation then corresponds to an agent-local view, denoted $View(agt, s)$, which is the sequence of observations made by the agent in situation s. Crucially, the agent's view excludes cases where $Obs(agt, a, s)$ is empty, allowing some actions to be completely hidden.

With a principled axiomatisation of the local perspective of each agent in hand, we then construct new formalisms and techniques for effective use of the situation calculus in asynchronous multi-agent domains. Specifically, we provide:

- A partially-ordered representation of the actions to be performed by a team of agents, that ensures synchronisation is always possible based on the local information available to each agent.

- A new procedure for effective inductive reasoning about a restricted form of query, using a meta-level fixpoint calculation on top of the standard regression operator. This allows certain second-order aspects of our axiomatisation to be "factored out" of the reasoning process when formulating regression rules.

- A new formalism for individual-level knowledge based explicitly on the agent's local view, with accompanying regression rules that use our new technique for inductive reasoning to handle arbitrarily-long sequences of hidden actions.

- A comprehensive treatment of group-level epistemic modalities such as common knowledge, using an epistemic interpretation of dynamic logic to gain the expressive power needed to formulate a regression rule for common knowledge.

These contributions provide a powerful fundamental framework for the situation calculus to represent and reason about asynchronous multi-agent domains.

While our results significantly extend the capabilities of the situation calculus, they are also firmly grounded in its existing theory and practice. The new concepts are axiomatised in a way that is compatible with standard basic action theories, as well as with common extensions such as concurrent actions and continuous time. We are therefore confident that our results can be integrated smoothly with existing theories and systems based on the situation calculus.

1.3 Scope

Before proceeding with the main body of the thesis, it is worth pausing to clarify the precise scope of our investigation and hopefully avoid any confusion over the details of our terminology and contributions.

1.3.1 Asynchronicity

The term "asynchronous domain" has come to mean slightly different things in different research fields, so our use of the term needs to be carefully delineated. Several common definitions frame asynchronicity as a property specific to inter-agent communication. For example, Fischer et al. [30] describe asynchronous domains as those having unbounded message delivery times, while Halpern and Moses [39] use the term to mean domains with delayed message delivery and no global clock.

The definition used in this thesis is more primitive than these, and corresponds to the account given by van Benthem and Pacuit [118]: a domain is *synchronous* if at any time, all agents know precisely how many actions have been performed. The internal state of each agent is thus updated in lock-step with the global state of the world. By contrast, in asynchronous domains agents must allow for potentially arbitrarily-many hidden actions which may or may not have occurred.

Where a communication-specific definition of asynchronicity is used, as in [30, 39], the potential for such hidden actions is implicitly assumed. The situation calculus typically assumes the exact opposite, limiting itself to strictly synchronous domains. We therefore argue that modelling richer notions of asynchronous communication in the situation calculus first requires a robust account of the kind of foundational asynchronicity treated in this thesis. While we do not specifically investigate asynchronous communication in the style of [30, 39], Chapter 4 shows that it has quite a natural formulation in our framework.

1.3.2 Common Knowledge

One of our contributions may, at first glance, seem to suggest that our definition of asynchronicity is flawed: our work on reasoning about common knowledge. Based on the famous paper of Halpern and Moses [39], it has become something of a "grand theorem" in epistemic reasoning that common knowledge cannot be obtained in asynchronous domains. So how can this thesis devote an entire chapter to reasoning about common knowledge in such domains?

The results of [39] apply specifically to obtaining common knowledge via asynchronous *communication*. They show that what is required to obtain common knowl-

edge is not synchronicity, per se, but *simultaneity* – the co-presence of the agents at the occurrence of a common event, which they all simultaneously observe. Asynchronous communication is not such a common event, and hence it cannot be used to obtain common knowledge.

Our approach, in essence, offers an explicit axiomatisation of this notion of simultaneity. By reasoning about their observations and the observations of others, the agents can deduce whether the occurrence of an action is simultaneously observed and thus whether it can contribute to common knowledge. Indeed, it is straightforward to model systems with no simultaneous events in our framework, and correspondingly straightforward to demonstrate that the agents cannot obtain common knowledge in such systems. Our contributions thus complement the standard results on asynchronicity and common knowledge, capturing them in the situation calculus and providing regression-based reasoning procedures with which they can be explored.

1.3.3 The Situation Calculus

As the title suggests, this thesis focuses exclusively on reasoning about multi-agent systems in the situation calculus. There are of course a range of related formalisms for reasoning about knowledge, action and change, which we overview briefly in Chapter 2 but otherwise do not directly consider.

Most closely related to the situation calculus are the fluent calculus of Thielscher [115] and the event calculus of Kowalski and Sergot [50]. There is also the family of approaches known as "dynamic epistemic logic" [8, 118, 119], which are based on modal logic and from which we draw some inspiration for our work in later chapters.

While there have been many attempts to combine the various action formalisms into a unifying theory of action, including [9, 51, 100, 113], there is yet to emerge a clear standard in this regard. In the meantime, we find the notation and metatheory of the situation calculus particularly suitable for expressing our main ideas, and find the Golog programming language to be a particularly powerful and flexible approach to specifying agent behaviour and programming shared tasks.

It is our hope that the strong underlying similarities between the major action formalisms will allow the ideas presented in this thesis to find some application or resonance beyond the specifics of the situation calculus.

1.4 Thesis Outline

The thesis now proceeds as follows:

- Chapter 2 presents the necessary background material on the situation calculus, Golog, and the Mozart programming system. More detailed reviews of the relevant literature are included in each subsequent chapter.
- Chapter 3 introduces *MIndiGolog*, a Golog variant suitable for planning the cooperative execution of a shared task. We demonstrate an implementation using distributed logic programming to share the planning workload, and discuss why current situation calculus techniques limit it to synchronous domains.
- Chapter 4 formalises the notion of a "local perspective" by reifying the *observations* made by each agent as the world evolves. We show that this formalism generalises existing approaches in which this perspective is modelled implicitly.
- Chapter 5 develops *joint executions*, a restricted kind of event structure where synchronisation is based on observations, and shows how to use them in planning the asynchronous execution of a shared MIndiGolog program.
- Chapter 6 develops a new technique for effective inductive reasoning, capable of handling a limited form of query that universally quantifies over situation terms. Dubbed the *persistence condition* operator, it uses a restricted fixpoint calculation to replace a second-order induction axiom.
- Chapter 7 develops a formalism for individual knowledge in the face of hidden actions, by specifying an agent's knowledge in terms of its local view. The persistence condition operator is used to augment the traditional regression rule for knowledge to account for arbitrarily-long sequences of hidden actions.
- Chapter 8 introduces common knowledge by using the syntax of dynamic logic to formulate a more expressive epistemic language than existing situation calculus theories. We formulate a regression rule for these complex epistemic modalities and formally relate them to our account of individual knowledge.
- Chapter 9 concludes with a summary of our achievements and our plans for ongoing work based on these results.
- The Appendices provide detailed proofs for those theorems where only a proof sketch is given in the main thesis body, describe how to obtain the software developed for this thesis, and specify axioms for the "cooking agents" example domain.

Chapter 2

Background

This chapter covers general background material for the thesis and provides a brief overview of the related literature. We defer more specific technical details and discussion of related work to the individual chapters that follow, where it can be presented in the appropriate context.

Readers familiar with the situation calculus are encouraged to briefly review this chapter. While it does not present any new results, it does introduce some novel notation and definitions which will be needed later in the thesis. They are introduced here to maintain consistency of the presentation. The introductory material on the Mozart programming platform may also be helpful.

We begin by introducing the base language of the situation calculus in Section 2.1, illustrated using examples from the "cooking agents" domain. Section 2.2 introduces the Golog family of programming languages, which are the standard formalism for representing complex tasks in the situation calculus. Reasoning about the knowledge of an agent, or *epistemic reasoning*, is covered in Section 2.3. Related formalisms for reasoning about action and change are briefly discussed in Section 2.4. Finally, Section 2.5 introduces the Mozart programming system, which will be used to implement our multi-agent Golog variant. Basic familiarity with formal logic is assumed throughout; readers requiring background on such material may find a gentle introduction in [43] and a more detailed treatment in [31].

2.1 The Situation Calculus

The situation calculus is a powerful formalism for describing and reasoning about dynamic worlds. It was first introduced by McCarthy and Hayes [70] and has since been significantly expanded and formalised [85, 92]. We use the particular variant

due to Reiter et. al. at the University of Toronto, sometimes called the "Toronto school" or "situations-as-histories" version. The formalisation below is based on the standard definitions from [59, 85, 91], but has been slightly generalised to accommodate several existing extensions to the situation calculus, as well as our own forthcoming extensions, in a uniform manner.

Readers familiar with the situation calculus should therefore note some modified notation: the unique names axioms $\mathcal{D}_{una}$ are incorporated into a general background theory $\mathcal{D}_{bg}$; the *Poss* fluent is subsumed by a general class of *action description predicates* defined in $\mathcal{D}_{ad}$; we parameterise the "future situations" predicate $s \sqsubset s'$ to assert that all intermediate actions satisfy a given predicate using $s <_\alpha s'$; and we use the single-step variant of the regression operator, with corresponding definitions of regressable formulae.

2.1.1 Notation

The language $\mathcal{L}_{sitcalc}$ of the situation calculus is a many-sorted language of first-order logic with equality, augmented with a second-order induction axiom, containing the following disjoint sorts:

- ACTION terms are functions denoting individual instantaneous events that can cause the state of the world to change;
- SITUATION terms are histories of the actions that have occurred in the world, with the initial situation represented by S_0 and successive situations built using the function $do : Action \times Situation \rightarrow Situation$;
- OBJECT terms represent any other object in the domain.

Fluents are predicates or functions that represent properties of the world that may change between situations, and so take a situation term as their final argument. Predicates and functions that do not take a situation term are called *rigid*. We use the term *primitive fluent* to describe fluents that are directly affected by actions, rather than being defined in terms of other fluents. No functions other than S_0 and *do* produce values of sort SITUATION.

For concreteness, let us present some formulae from an example domain that will be used throughout the thesis. In the "cooking agents" domain a group of robotic chefs inhabit a kitchen containing various ingredients and utensils, and they must cooperate to prepare a meal. Some example statements from this domain include "Joe does not have the knife initially", "Jim has the knife after he acquires it" and

"It is only possible to acquire an object if nobody else has it". Formally:

$$\neg HasObject(Joe, Knife1, S_0)$$
$$HasObject(Jim, Knife1, do(acquire(Jim, Knife1), S_0))$$
$$Poss(acquire(agt, obj), s) \equiv \neg\exists agt_2 : HasObject(agt_2, obj, s)$$

Here $HasObject$ is a primitive fluent, while $Poss$ is defined in terms of it.

$\mathcal{L}_{sitcalc}$ contains the standard alphabet of logical connectives, constants $\top$ and $\bot$, countably infinitely many variables of each sort, countably infinitely many predicates of each arity, etc; for a complete definition, consult the foundational paper by Pirri and Reiter [85]. We follow standard naming conventions for the situation calculus: upper-case roman names indicate constants; lower-case roman names indicate variables; greek characters indicate meta-variables or formula templates. All axioms universally close over their free variables at outermost scope. The notation $\bar{t}$ indicates a vector of terms of context-appropriate arity and type. The connectives $\wedge$, $\neg$, $\exists$ are taken as primitive, with $\vee$, $\rightarrow$, $\equiv$, $\forall$ defined in the usual manner.

In multi-agent domains it is customary to introduce a distinct sort AGENT to explicitly represent the agents operating in the world, and we will do so here. As seen in the example formulae above, the first argument of each action term gives the performing agent, which can be accessed by the function $actor(a)$.

Complex properties of the state of the world are represented using *uniform formulae*. These are basically logical combinations of fluents referring to a common situation term.

Definition 1 (Uniform Terms)**.** *Let σ be a fixed situation term, r an arbitrary rigid function symbol, f an arbitrary fluent function symbol, and x a variable that is not of sort* SITUATION*. Then the terms uniform in σ are the smallest set of syntactically-valid terms satisfying:*

$$\tau ::= x \mid r(\bar{\tau}) \mid f(\bar{\tau}, \sigma)$$

Definition 2 (Uniform Formulae)**.** *Let σ be a fixed situation term, R an arbitrary rigid predicate, F an arbitrary primitive fluent predicate, τ an arbitrary term uniform in σ, and x an arbitrary variable that is not of sort* SITUATION*. Then the formulae uniform in σ are the smallest set of syntactically-valid formulae satisfying:*

$$\phi ::= F(\bar{\tau}, \sigma) \mid R(\bar{\tau}) \mid \tau_1 = \tau_2 \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \exists x : \phi$$

We will call a formula *uniform* if it is uniform in some situation. The important aspect of this definition is that the formula refers to no situation other than σ, which appears as the final argument of all fluents in the formula. In particular, uniform formulae cannot quantify over situations or compare situation terms, and cannot contain non-primitive fluents.

The meta-variable ϕ is used throughout to refer to an arbitrary uniform formula. Since they represent some aspect of the state of the world, it is frequently useful to evaluate uniform formulae at several different situation terms. The notation $\phi[s']$ represents a uniform formula with the particular situation s' inserted into all its fluents. We may also completely suppress the situation term to simplify the presentation, using ϕ^{-1} to represent a uniform formula with the situation argument removed from all its fluents. For example, given:

$$\phi = HasObject(Jim, Knife1, s) \wedge HasObject(Joe, Bowl2, s)$$

Then we have:

$$\begin{aligned} \phi[s'] &= HasObject(Jim, Knife1, s') \wedge HasObject(Joe, Bowl2, s') \\ \phi^{-1} &= HasObject(Jim, Knife1) \wedge HasObject(Joe, Bowl2) \end{aligned}$$

Note that these are strictly meta-level operations, corresponding to possibly quite complex sentences from the underlying logic. They are *not* terms or operators from the logic itself.

2.1.2 Axioms

The dynamics of a particular domain are captured by a set of sentences from $\mathcal{L}_{sitcalc}$ called a *basic action theory*. Queries about the behaviour of the world are posed as logical entailment queries relative to this theory.

Definition 3 (Basic Action Theory)**.** *A basic action theory, denoted $\mathcal{D}$, is a set of situation calculus sentences (of the specific syntactic form outlined below) describing a particular dynamic world. It consists of the following disjoint sets: the foundational axioms of the situation calculus (Σ); action description axioms defining preconditions etc for each action ($\mathcal{D}_{ad}$); successor state axioms describing how primitive fluents change between situations ($\mathcal{D}_{ssa}$); axioms describing the value of primitive fluents in the initial situation ($\mathcal{D}_{S_0}$); and axioms describing the static background facts of the domain ($\mathcal{D}_{bg}$):*

$$\mathcal{D} = \Sigma \cup \mathcal{D}_{ad} \cup \mathcal{D}_{ssa} \cup \mathcal{D}_{S_0} \cup \mathcal{D}_{bg}$$

These axioms must satisfy some simple consistency criteria to constitute a valid domain description; see [85] for the details. This definition is slightly broader than the standard definitions found in the literature [59, 85, 91] and is designed to accommodate a variety of extensions to the situation calculus in a uniform manner.

We assume an arbitrary, but fixed, basic action theory $\mathcal{D}$.

Background Axioms

The set $\mathcal{D}_{bg}$ characterises the static aspects of the domain, and contains all axioms defining rigid predicates or functions. In particular, it must contain a set of unique names axioms asserting that action terms with different types or arguments are in fact different, e.g.:

$$acquire(agt, obj) \neq release(agt, obj)$$
$$acquire(agt_1, obj_1) = acquire(agt_2, obj_2) \rightarrow agt_1 = agt_2 \wedge obj_1 = obj_2$$

It also contains domain closure axioms for the sorts ACTION, AGENT and OBJECT, and defines the function $actor(a)$ to give the agent performing an action. The background axioms are a generalisation of the set $\mathcal{D}_{una}$ commonly found in the literature, which contains only the unique names axioms.

Successor State Axioms

The set $\mathcal{D}_{ssa}$ contains one *successor state axiom* for each primitive fluent in the domain. These axioms provide an elegant monotonic solution to the frame problem for that fluent [92] which has been instrumental to the popularity and utility of the situation calculus. They have the following general form:

$$F(\bar{x}, do(a, s)) \equiv \Phi_F(\bar{x}, a, s)$$

Here Φ_F is uniform in s. While we will make no assumptions about the internal structure of Φ_F, it typically takes the form shown below, which may help elucidate the purpose of these axioms:

$$F(\bar{x}, do(a, s)) \equiv \Phi_F^+(\bar{x}, a, s) \vee F(\bar{x}, s) \wedge \neg\Phi_F^-(\bar{x}, a, s)$$

Here Φ_F^+ and Φ_F^- are formulae uniform in s, representing the positive and negative effect axioms for that fluent. This may be read as "F is true after performing a if a made it true, or it was previously true and a did not make it false". For example,

the dynamics of the $HasObject$ fluent may be specified using:

$$HasObject(agt, obj, do(a, s)) \equiv a = acquire(agt, obj) \\ \vee \; HasObject(agt, obj, s) \wedge a \neq release(agt, obj)$$

For functional fluents, $\mathcal{D}_{ssa}$ contains a similar axiom to specify the value v of the fluent after an action has occurred:

$$f(\bar{x}, do(a, s)) = v \equiv \Phi_f(v, \bar{x}, a, s)$$

Action Description Predicates

The set $\mathcal{D}_{ad}$ generalises the standard *action precondition axioms* [85] to define fluents that describe various aspects of the performance of an action, which we call *action description predicates*. These are the only non-primitive fluents permitted in a basic action theory. The predicate $Poss(a, s)$ is the canonical example, indicating whether it is possible to perform an action in a given situation. The set $\mathcal{D}_{ad}$ contains a single axiom of the following form, defining the complete set of preconditions for the action variable a, where Π_{Poss} is a formula uniform in s:

$$Poss(a, s) \equiv \Pi_{Poss}(a, s)$$

Note that this is a slight departure from the standard approach of [85], in which the preconditions for each action type are enumerated individually. The more restrictive approach presented here embodies a domain-closure assumption on the ACTION sort. If there are finitely many action types then Π_{Poss} is simply the completion of the precondition axioms for each action type. The single-axiom form is necessary when quantifying over "all possible actions" and has been widely used in the literature [96, 124].

In principle, any number of predicates and functions can be defined in this way; a common example is the sensing-result function $SR(a, s)$ which we will describe in Chapter 4. The general notion of an action description predicate allows us to treat all of them in a uniform manner. We will use the meta-variable α to represent an arbitrary action description predicate, and allow the action and situation arguments to be suppressed in a similar way to situation-suppressed uniform formulae.

In preparation for the coming material on extensions to the situation calculus in Section 2.1.4, let us introduce an action description predicate *Legal* that identifies actions that can be legally executed in the real world. In the basic situation calculus,

it is simply equivalent to $Poss$:

$$Legal(a, s) \equiv Poss(a, s)$$

As shown by the above, it is often useful to define new action description predicates in terms of simpler existing ones, rather than directly in terms of the primitive fluents of the domain. As long as these definitions are well-founded they can be expanded down to primitive fluents when constructing the basic action theory.

Foundational Axioms

The foundational axioms Σ ensure that situations form a branching-time account of the world state. There is a distinguished situation S_0 called the *initial situation*. Situations in general form a tree structure with the initial situation at the root and $do(a, s)$ constructing the successor situation resulting when the action a is performed in situation s. All situations thus produced are distinct:

$$do(a_1, s_1) = do(a_2, s_2) \rightarrow a_1 = a_2 \wedge s_1 = s_2$$

We abbreviate the performance of several successive actions by writing:

$$do([a_1, \ldots, a_n], s) \stackrel{\text{def}}{=} do(a_n, do(\ldots, do(a_1, s)))$$

There is also a second-order induction axiom asserting that all situations must be constructed in this way, which is needed to prove statements that universally quantify over situations [89]:

$$\forall P : [P(S_0) \wedge \forall s, a : (P(s) \rightarrow P(do(a, s)))] \rightarrow \forall s : P(s)$$

The relation $s \sqsubset s'$ indicates that s' is in the future of s and is defined as follows:

$$\neg(s \sqsubset S_0)$$
$$s \sqsubset do(a, s') \equiv s \sqsubseteq s'$$

Here $s \sqsubseteq s'$ is the standard abbreviation for $s \sqsubset s' \vee s = s'$. This notion of "in the future of" can be extended to consider only those futures in which all actions satisfy a particular action description predicate. We define as a macro the relation $<_\alpha$ for an arbitrary action description predicate α, with the following definition:

$$s <_\alpha s' \stackrel{\text{def}}{=} s \sqsubset s' \wedge \forall a, s'' : \big(s \sqsubset do(a, s'') \sqsubseteq s' \rightarrow \alpha[a, s'']\big)$$

It is straightforward to demonstrate that this macro satisfies the following properties, which are analogous to the definition of $\sqsubset$:

$$\neg\left(s <_\alpha S_0\right)$$
$$s <_\alpha do(a, s') \equiv s \leq_\alpha s' \wedge \alpha[a, s']$$

The *legal situations* are those in which every action was legal to perform in the preceding situation. These are of fundamental importance, as they are the only situations that could be reached in the real world:

$$Legal(s) \stackrel{\text{def}}{=} S_0 \leq_{Legal} s$$

Initial State Axioms

The set $\mathcal{D}_{S_0}$ describes the actual state of the world before any actions are performed. It is a collection of sentences uniform in S_0 stating what holds in the initial situation. In many domains the initial state can be completely specified, so $\mathcal{D}_{S_0}$ is often in a closed form suitable for efficient automated reasoning.

Note that, unlike [59, 85, 91], we include static facts about the domain in $\mathcal{D}_{bg}$ rather than $\mathcal{D}_{S_0}$. This is entirely a cosmetic change to allow us to talk about these static facts separately from the initial database.

2.1.3 Reasoning

An important feature of the situation calculus is the existence of effective reasoning procedures for certain types of query. These are generally based on syntactic manipulation of a query into a form that is more amenable to reasoning, for example because it can be proven without using some of the axioms from $\mathcal{D}$.

Types of Reasoning

In the general case, answering a query about a basic action theory $\mathcal{D}$ is a theorem-proving task in second-order logic (denoted SOL) due to the induction axiom included in the foundational axioms:

$$\mathcal{D} \models_{SOL} \psi$$

This is clearly problematic for effective automated reasoning, but fortunately there exist particular syntactic forms for which some of the axioms in $\mathcal{D}$ are not required.

If a query only performs *existential* quantification over situation terms, it can be answered without the induction axiom (denoted I) and thus using only first-order logic (FOL) [85]:

$$\mathcal{D} \models_{SOL} \exists s : \psi(s) \text{ iff } \mathcal{D} - \{I\} \models_{FOL} \exists s : \psi(s)$$

While this is a substantial improvement over requiring a second-order theorem prover, it is still far from an effective technique. Effective reasoning requires that the set of axioms be reduced as much as possible.

In their work on state constraints, Lin and Reiter [66] show how to reduce the task of verifying a state constraint to a reasoning task we call *static domain reasoning*, where only the background axioms need to be considered:

$$\mathcal{D}_{bg} \models_{FOL} \forall s : \phi[s]$$

Since the axioms in $\mathcal{D}_{bg}$ do not mention situation terms, the leading quantification in such queries has no effect – ϕ will be entailed for all s if and only if it is entailed for some s. This is a major improvement because universal quantification over situation terms usually requires the second-order induction axiom. Their work has shown that this requirement can be circumvented in some cases.

Simpler still are queries uniform in the initial situation, which can be answered using only first-order logic and a limited set of axioms:

$$\mathcal{D} \models_{SOL} \phi[S_0] \quad \text{iff} \quad \mathcal{D}_{S_0} \cup \mathcal{D}_{bg} \models_{FOL} \phi[S_0]$$

We call such reasoning *initial situation reasoning*. Since the axioms $\mathcal{D}_{S_0} \cup \mathcal{D}_{bg}$ often satisfy the closed-world assumption, provers such as Prolog can be employed to handle this type of query quite effectively.

Regression

The principle tool for effective reasoning in the situation calculus is the regression meta-operator $\mathcal{R}_{\mathcal{D}}$, a syntactic manipulation that encodes the preconditions and effects of actions into the query itself, meaning fewer axioms are needed for the final reasoning task [85]. The idea is to reduce a query about some future situation to a query about the initial situation only.

There are two styles of regression operator commonly defined in the literature: the single-pass operator as defined in [85] which reduces to S_0 in a single application, the the single-step operator as defined in [98] which operates one action at a time.

We use the single-step variant because it is the more expressive of the two – while it is straightforward to define the single-pass operator in terms of the single-step operator, the reverse is not the case.

Regression is only defined for a certain class of formulae, the *regressable formulae*.

Definition 4 (Regressable Terms)**.** *Let σ be an arbitrary situation term, x an arbitrary variable not of sort situation, r an arbitrary rigid function and f an arbitrary fluent function. Then the regressable terms are the smallest set of syntactically-valid terms satisfying:*

$$\nu ::= \sigma \mid x \mid f(\bar{\nu}, \sigma) \mid r(\bar{\nu})$$

Definition 5 (Regressable Formulae)**.** *Let σ be an arbitrary situation term, x an arbitrary variable not of sort situation, ν an arbitrary regressable term, R an arbitrary rigid predicate, F an arbitrary primitive fluent predicate, and α an arbitrary action description predicate. Then the regressable formulae are the smallest set of syntactically-valid formulae satisfying:*

$$\varphi ::= F(\bar{\nu}, \sigma) \mid \alpha(\bar{\nu}, a, \sigma) \mid R(\bar{\nu}) \mid \nu_1 = \nu_2 \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \exists x : \varphi$$

Regressable formulae are more general than uniform formulae. In particular, they can contain action description predicates and may mention different situation terms. They cannot, however, quantify over situation terms or compare situations using the $\sqsubset$ predicate.

The regression operator is then defined using a series of *regression rules* such as those shown below, which mirror the structural definition of regressable formulae.

Definition 6 (Regression Operator)**.** *Let R be a rigid predicate, α be an action description predicate with axiom $\alpha(\bar{\nu}, a, s) \equiv \Pi_\alpha(a, s)$ in $\mathcal{D}_{ad}$, and F be a primitive fluent with axiom $F(\bar{x}, s) \equiv \Phi_F(\bar{x}, s)$ in $\mathcal{D}_{ssa}$. Then the regression of ϕ, denoted $\mathcal{R}_\mathcal{D}(\phi)$, is defined according to the following structural rules:*

$$\mathcal{R}_\mathcal{D}(\varphi_1 \wedge \varphi_2) \stackrel{def}{=} \mathcal{R}_\mathcal{D}(\varphi_1) \wedge \mathcal{R}_\mathcal{D}(\varphi_2)$$

$$\mathcal{R}_\mathcal{D}(\exists x : \varphi) \stackrel{def}{=} \exists x : \mathcal{R}_\mathcal{D}(\varphi)$$

$$\mathcal{R}_\mathcal{D}(\neg\varphi) \stackrel{def}{=} \neg\mathcal{R}_\mathcal{D}(\varphi)$$

$$\mathcal{R}_\mathcal{D}(\alpha(\bar{\nu}, a, \sigma)) \stackrel{def}{=} \mathcal{R}_\mathcal{D}(\Pi_\alpha(\bar{\nu}, a, \sigma))$$

$$\mathcal{R}_\mathcal{D}(F(\bar{\nu}, do(a, \sigma))) \stackrel{def}{=} \Phi_F(\bar{\nu}, a, \sigma)$$

$$\mathcal{R}_\mathcal{D}(F(\bar{\nu}, s)) \stackrel{def}{=} F(\bar{\nu}, s)$$

$$\mathcal{R}_\mathcal{D}(F(\bar{\nu}, S_0)) \stackrel{def}{=} F(\bar{\nu}, S_0)$$

We have omitted some technical details here, such as the handling of functional fluents; consult [85] for the details. The key point is that each application of the regression operator replaces action description predicates with their definitions from $\mathcal{D}_{ad}$ and primitive fluents with their successor state axioms from $\mathcal{D}_{ssa}$, "unwinding" a single action from each $do(a, \sigma)$ situation term in the query. If the situation term is not constructed using do, it is left unchanged.

Since $\mathcal{D}$ is fixed, we will henceforth drop the subscript and simply write $\mathcal{R}$ for the regression operator. When dealing with situation-suppressed uniform formulae, we will use a two-argument operator $\mathcal{R}(\phi, a)$ to indicate the regression of ϕ over the action a. It should be read as a shorthand for $\mathcal{R}(\phi[do(a, s)])^{-1}$ using the situation-suppression operator from Section 2.1.1.

Let us briefly state some important properties of the regression operator. First, and most importantly, it preserves equivalence of formulae:

Proposition 1. *Let φ be a regressable formula, then $\mathcal{D} \models \varphi \equiv \mathcal{R}(\varphi)$*

Any formula uniform in $do(a, s)$ is regressable, and the result is uniform in s:

Proposition 2. *Let ϕ be uniform in $do(a, s)$, then $\mathcal{R}(\phi)$ is uniform in s*

Let $\mathcal{R}^*$ denote repeated applications of $\mathcal{R}$ until the formula remains unchanged. Such applications can transform a query about some future situation into a query about the initial situation only:

Proposition 3. *Let ϕ be uniform in $do([a_1 \dots a_n], S_0)$, then $\mathcal{R}^*(\phi)$ is uniform in S_0*

This last property is key to effective reasoning in the situation calculus, as it allows one to answer the *projection problem*. To determine whether ϕ holds in a given future situation, it suffices to determine whether $\mathcal{R}^*(\phi)$ holds in the initial situation. As discussed above, queries uniform in S_0 are much easier to answer. The axioms $\mathcal{D}_{ad}$ and $\mathcal{D}_{ssa}$ are essentially "compiled into the query" by the $\mathcal{R}^*$ operator. While an efficiency gain is not guaranteed, regression has proven a very effective technique in practice [62, 85].

Decidability

Even given the use of regression to reduce the number of axioms required, reasoning still requires first-order logic and is thus only semi-decidable in general. Practical systems implemented on top of the situation calculus typically enforce additional restrictions on the domain in order to gain decidability.

A common restriction is to assume that the ACTION and OBJECT domains are finite. This allows quantification over these variables to be replaced with finite

conjunctions or disjunctions, essentially "propositionalising" the domain [13, 20, 91]. Both static domain and initial situation reasoning can then be performed using propositional logic, which is decidable. This may also be combined with special-purpose decision procedures for particular objects in the domain, such as deciding linear constraints over the integers or reals [91, 93].

A similar, but slightly less onerous restriction, is to ensure that the construction of function terms is well-founded [13]. This prevents building the arbitrarily-nested terms from the Herbrand universe that cause non-termination in first-order theorem provers, again gaining decidability.

Recent work by Gu and Soutchanski [38] has shown how to model some situation calculus domains using to the two-variable fragment of first-order logic. Since this fragment is decidable in general, both static domain and initial situation reasoning are decidable in such domains.

Inductive Reasoning

One class of query that cannot be answered effectively using regression are formulae that universally quantify over situations. Examples of such queries include verifying state constraints ("for all situations, the constraint is satisfied") and determining the impossibility of a goal ("for all situations, the goal is not satisfied"). The difficulty here comes from the induction axiom.

Reiter [89] has shown why the induction axiom is necessary to prove statements that universally quantify over situation terms. This work demonstrates the use of the axiom in manual proofs, but offers no procedure for answering such queries automatically. Other work considering inductive reasoning has focused exclusively on verifying state constraints [11, 66]. While it is possible to automate this verification in some cases, there are currently no general-purpose tools for effectively handling queries that universally quantify over situation terms.

It is this limitation, more than any other, that has restricted the situation calculus to synchronous domains. In asynchronous domains agents must account for potentially arbitrarily-long sequences of hidden actions, which requires universal quantification over situation terms. In Chapter 6 we develop a new reasoning technique to help overcome this limitation.

Progression

While regression has proven quite an effective technique in practice, it has an obvious shortcoming in domains with long histories – the computation required to reason about the current state of the world increases with each action performed.

An alternative approach is *progression*, in which the initial state of the world $\mathcal{D}_{S_0}$ is updated with each action performed, to give a new set of axioms describing the state of the current situation. Although this increases the upfront complexity when an action is performed, this work is amortised over many queries about the updated state. Thielscher [114] makes a compelling case that progression gives better runtime performance in domains with many actions. Why, then, do we focus only on regression in this thesis?

The theoretical foundations of progression in the situation calculus were laid out by Lin and Reiter [67] and come with an important caveat: the progression of a first-order database is not always first-order definable. This conjecture was recently proven by Vassos and Levesque [124], who show that while it is possible to define first-order progressions of a database that are valid for restricted classes of query, a first-order progressed database cannot be complete in general. As such, work on progression in the situation calculus has focused on restricted queries or restricted databases for which first-order progressions exist [68, 123, 125]. By contrast, the regression operator is sound and complete for answering a broad range of queries.

In this thesis, we develop formalisms and reasoning techniques for problems which have not been approached before in the situation calculus. Our first priority must be a sound and complete reasoning tool, for which regression is a good match. Advanced techniques such a progression are considered future work at this stage.

2.1.4 Extensions

The base language of the situation calculus may seem simplistic, lacking many features that would be desirable for modelling rich multi-agent domains. However, it is possible to significantly enrich the domain features that can be modelled while maintaining the elegance and simplicity of the base situation calculus. We now discuss several such extensions that are important in multi-agent domains.

Concurrent Actions

In the basic situation calculus only a single action can occur at any instant. While suitable for most single-agent domains, this limitation is emphatically not suitable for multi-agent systems – several actions can easily occur simultaneously if performed by different agents. Modelling this *true concurrency* is necessary to avoid problems with conflicting or incompatible actions. There is also the potential to utilise concurrency to execute tasks more efficiently. Clearly a solid account of concurrency is required for reasoning about multi-agent teams.

The work of [65, 83, 93] adds true concurrency to the situation calculus by

replacing action terms with *sets* of actions that are performed simultaneously. The additional sort CONCURRENT is added to $\mathcal{L}_{sitcalc}$, and the appropriate axioms for set theory are added to $\mathcal{D}_{bg}$. All functions and predicates that take an ACTION term are are modified to take a CONCURRENT term instead. For example, $do(a, s)$ becomes $do(\{a_1, a_2, ...\}, s)$. Successor state axioms are modified to test for set membership rather than equality of action terms. For example, the successor state axiom for $HasObject$ would become:

$$HasObject(agt, obj, do(c, s)) \equiv acquire(agt, obj) \in c \\ \vee \ HasObject(agt, obj, s) \wedge release(agt, obj) \notin c$$

Since it operates solely by replacing formulae with their equivalents, the regression operator is unchanged by this extension and effective reasoning is still possible.

There is, however, a subtle complication in axiomatising action description predicates such as $Poss$: interaction between primitive actions. A combination of actions is not guaranteed to be possible even if each of the individual actions are. For example, two agents may not be able to acquire the same resource at the same time. This is known as the precondition interaction problem and has undergone extensive research [79, 80, 83]. We make no explicit commitment towards a solution for this problem. Rather, we assume that the axioms in $\mathcal{D}_{ad}$ contain the necessary logic to account for interaction for all action description predicates.

To avoid unintuitive behaviour, we assume that the domain entails the following consistency requirements for the empty set of actions:

Definition 7 (Empty Action Consistency)**.** *A basic action theory $\mathcal{D}$ using concurrent actions must entail the following consistency requirements for the empty set of actions:*

$$\mathcal{D} \models \forall s : \neg Legal(\{\}, s)$$
$$\mathcal{D} \models \forall s : \phi[do(\{\}, s)] \equiv \phi[s]$$

Since true concurrency is such an important aspect of multi-agent systems, we will assume concurrent actions are in use throughout the rest of the thesis.

Time

An explicit notion of time can make coordination between agents easier, as joint actions may be performed at a particular time. It also allows a richer description of the world, particularly in domains such as the cooking agents where time can play

an important part in tasks to be performed.

The standard approach to time in the situation calculus is that of [82, 83, 93]. An additional sort TIMEPOINT is introduced, which can be any appropriately-behaved sequence such as integers or reals. The axiomatisation of timepoints is added to $\mathcal{D}_{bg}$, and each action gains an extra argument indicating the time at which is was performed. The functions *time* and *start* are introduced to give the performance time of an action and the start time of a situation respectively. The start time of the initial situation may be defined arbitrarily, but is typically taken to be zero.

However, this approach does not integrate cleanly with concurrent actions: it requires an additional predicate *Coherent* to ensure that the performance time is the same for all members in a set of concurrent actions [93]. The legal situations must be restricted to those in which all actions are coherent.

To avoid this extra complexity, we follow the approach taken in the related formalism of the fluent calculus [69] and attach the temporal component to the set of concurrent actions itself, rather than to each individual action. A similar approach is used in [97] to avoid problems when combining knowledge and time.

Predicates and functions taking terms of sort ACTION are modified to take CONCURRENT#TIMEPOINT pairs, e.g. $do(c, s)$ becomes $do(c\#t, s)$. The new function *start* is added to the foundational axioms with the following definition:

$$start(do(c\#t, s)) = t$$

We must ensure that successor situations have later start times than their preceding situations, by modifying the definition of *Legal*:

$$Legal(c\#t, s) \equiv Poss(c\#t, s) \wedge start(s) < t$$

Introducing timepoints does not affect the regression operator, but does increase the complexity of reasoning as $\mathcal{D}_{bg}$ now contains the axioms of number theory. In practice, we limit predicates about time to express only *linear* relationships, and employ a linear constraint solver for decidable reasoning over the temporal component.

Natural Actions

Natural actions are a special class of exogenous actions, those actions which occur outside of an agent's control [93]. They are classified according to the following requirement: natural actions must occur if it is possible for them to occur, unless an earlier action prevents them. For example, a timer will ring at the time it was

set for, unless it is switched off. Such actions are used to model the predictable behaviour of the environment.

Natural actions are identified by the truth of the predicate $Natural(a)$. The times at which natural actions may occur are specified by the $Poss$ predicate. For example, suppose that the fluent $TimerSet(m, s)$ represents the fact that a timer is set to ring in m minutes in situation s. The possibility predicate would entail:

$$Poss(ringTimer\#t, s) \equiv \exists m : [TimerSet(m, s) \wedge t = start(s) + m]$$

The timer may thus ring only at its predicted time. To enforce the requirement that natural actions *must* occur whenever possible, the action description predicate $Legal(c\#t, s)$ is adjusted to ensure that $c\#t$ is not legal if natural actions could occur at some earlier time:

$$\begin{aligned} Legal(c\#t, s) \equiv\ & Poss(c\#t, s) \wedge start(s) < t \\ & \wedge\ \forall a, t' :\ \big[Natural(a) \wedge Poss(\{a\}\#t', s)\ \rightarrow\ \big(a \in c \vee t < t'\big)\big] \end{aligned}$$

Thus it is only legal to perform actions c at time t if no natural actions can occur in s at a time less than t.

Following this intuition, the *least natural time point* (or "LNTP") of a situation is defined as the earliest time at which a natural action may occur [91]. Rather than adding another axiom, this can be defined using a simple macro:

$$\begin{aligned} \mathbf{LNTP}(s, t) \stackrel{\text{def}}{=} \exists a : [Natural(a) \wedge Poss(\{a\}\#t, s)] \wedge \\ \forall a', t' : \big[Natural(a') \wedge Poss(\{a'\}\#t, s) \rightarrow t \leq t'\big] \end{aligned}$$

We assume that the theory of action avoids certain pathological cases identified in [91], so that absence of an LNTP implies that no natural actions are possible. That is to say, we assume the following is a consequence of the basic action theory:

$$\mathcal{D}\ \models\ [\exists a, t :\ Natural(a) \wedge Poss(\{a\}\#t, s)] \rightarrow [\exists t :\ \mathbf{LNTP}(s, t)]$$

The LNTP is important when planning in the presence of natural actions – one cannot plan to perform some actions at time t if t is greater than the least natural timepoint of the current situation. We also define a related concept, the set of *pending natural actions*, as the set of all natural actions that are possible at the least natural time point:

$$\mathbf{PNA}(s, c) \stackrel{\text{def}}{=} \exists t :\ \mathbf{LNTP}(s, t) \wedge \forall a : [Natural(a) \wedge Poss(\{a\}\#t, s) \equiv a \in c]$$

Long-Running Tasks

Although all actions in the situation calculus are instantaneous, it is still possible to model long-running tasks that have a finite duration. They are modelled by decomposing them into instantaneous $beginTask$ and $endTask$ actions, and a fluent $DoingTask$ indicating that a task is in progress [83].

In the presence of long-running tasks, a robust account of natural actions is very important – the $endTask$ action must be natural to ensure that any task that is initiated eventually terminates at the appropriate time.

Summary

As can be seen from the discussion above, it is possible to enrich the situation calculus with some very powerful domain features while still maintaining the basic structure of the language, and retaining regression as the principle tool for effective automated reasoning.

While we assume concurrent actions are in use through the rest of this thesis, we shall only refer explicitly to other rich domain features – such as time and natural actions – when we wish to make a special point about their treatment. By uniformly using the predicate $Legal$ to identify actions that can legally be performed in the world, rather than the base $Poss$ predicate, we ensure that our techniques are applicable regardless of the particular domain features being used.

2.2 Golog

Golog is a declarative agent programming language that is the standard approach to specifying complex behaviours in the situation calculus [62]. Testimony to its success are its wide range of applications and many extensions to provide additional functionality [17, 21, 27]. For simplicity, we use the general name "Golog" to refer to the standard family of languages based on this technique, including ConGolog [21] and IndiGolog [17].

2.2.1 Notation

To program an agent using Golog one specifies a situation calculus theory of action, and a program consisting of actions from the theory connected by programming constructs such as if-then-else, while loops, and nondeterministic choice. Table 2.1 lists the standard operators available in various incarnations of the language.

Readers familiar with dynamic logic will recognise some of these operators, but others are unique to first-order formalisms such as Golog. Many Golog operators are

Operator	Meaning
Nil	The empty program
a	Execute action a in the world
$\phi?$	Proceed if condition ϕ is true
$\delta_1; \delta_2$	Execute $\delta_1$followed by δ_2
$\delta_1 \vert \delta_2$	Execute either δ_1 or δ_2
$\pi(x, \delta(x))$	Nondet. select arguments for δ
$\delta*$	Execute δ zero or more times
if ϕ **then** δ_1 **else** δ_2	Exec. δ_1 if ϕ holds, δ_2 otherwise
while ϕ **do** δ	Execute δ while ϕ holds
proc$P(\overrightarrow{x})\delta(\overrightarrow{x})$**end**	Procedure definition
$\delta_1 \vert\vert \delta_2$	Concurrent execution
$\delta_1 \ll \delta_2$	Prioritised concurrency
$\delta^{\vert\vert}$	Concurrent iteration
$\Sigma(\delta)$	Plan execution offline

Table 2.1: Operators used in Golog and its descendants

nondeterministic and may be executed in several different ways. It is the task of the agent to plan a deterministic instantiation of the program, a sequence of actions that can legally be performed in the world. Such a sequence is called a *legal execution* of the program.

To get a feel for how these operators can be used, consider some example programs. Figure 2.1 shows a simple program for Jim to wash the dishes. It makes use of the nondeterministic "pick" operator to select and clean a dish that needs washing, and does so in a loop until no dirty dishes remain. The legal executions of this program are sequences of $clean(Jim, d)$ actions, one for each dirty dish in the domain, performed in any order.

while $\exists d : Dirty(d)$ **do**
$\pi(d, clean(Jim, d))$
end

Figure 2.1: A Golog program for washing the dishes

Figure 2.2 shows a program that we will return to in subsequent chapters, giving instructions for how to prepare a simple salad. The procedure $ChopTypeInto$ (not shown) directs the specified agent to acquire an ingredient of the specified type, chop it, and place it into the indicated bowl. The procedure $MakeSalad$ nondeterministically selects an agent to do this for a lettuce, a carrot, and a tomato. Note the nondeterminism in this program: the agent assigned to handling each ingredient

is not specified (π construct), nor is the order in which they should be processed ($||$ construct). There is thus considerable scope for cooperation between agents to effectively carry out this task.

$$
\begin{array}{c}
\textbf{proc}\ MakeSalad(dest) \\
[\pi(agt, ChopTypeInto(agt, Lettuce, dest)) \,||\, \\
\pi(agt, ChopTypeInto(agt, Carrot, dest)) \,||\, \\
\pi(agt, ChopTypeInto(agt, Tomato, dest))]\ ; \\
\pi(agt, [acquire(agt, dest)\,; \\
beginTask(agt, mix(dest, 1))\,; \\
endTask(agt, mix(dest, 1))\,; \\
release(agt, dest)])\ \textbf{end}
\end{array}
$$

Figure 2.2: A Golog program for making a salad

2.2.2 Semantics

The original semantics of Golog were defined using macro-expansion [62]. The macro $\mathbf{Do}(\delta, s, s')$ was defined to be true if program δ could be successfully executed in situation s, leaving the world in situation s'. However, these semantics could not support the concurrent execution of two programs and were modified with the introduction of ConGolog [21] to use two predicates $Trans(\delta, s, \delta', s')$ and $Final(\delta, s)$ which are capable of representing single steps of execution of the program.

The predicate $Trans(\delta, s, \delta', s')$ holds when executing a step of program δ can cause the world to move from situation s to situation s', after which δ' remains to be executed. It thus characterises single steps of computation. The predicate $Final(\delta, s)$ holds when program δ may legally terminate its execution in situation s. We base our work on the semantics of IndiGolog [17], which builds on ConGolog [21] and is the most feature-full of the standard Golog variants. The full semantics are available in the references, but we present some illustrative examples below.

The transition rule for a program consisting of a single action is straightforward – it transitions by performing the action, provided it is possible in the current situation. Such a program may not terminate its execution since the action remains to be performed:

$$
\begin{aligned}
Trans(a, s, \delta', s') &\equiv Poss(a, s) \wedge \delta' = Nil \wedge s' = do(a, s) \\
Final(a, s) &\equiv \bot
\end{aligned}
$$

The transition rule for a test operator proceeds only if the test is true, leaving the situation unchanged, and likewise cannot terminate execution until the test has been satisfied:

$$\begin{aligned} Trans(?\phi, s, \delta', s') &\equiv \phi[s] \wedge \delta' = Nil \wedge s' = s \\ Final(?\phi, s) &\equiv \bot \end{aligned}$$

Now consider a simple nondeterministic operator, the "choice" construct that executes one of two alternate programs:

$$\begin{aligned} Trans(\delta_1|\delta_2, s, \delta', s') &\equiv Trans(\delta_1, s, \delta', s') \vee Trans(\delta_2, s, \delta', s') \\ Final(\delta_1|\delta_2, s) &\equiv Final(\delta_1, s) \vee Final(\delta_2, s) \end{aligned}$$

It is possible for this operator to transition in two different ways - by executing a step of execution from the first program, or a step of execution from the second program. Slightly more complicated, but of fundamental important in the next chapter, is the semantics of the concurrency operator:

$$\begin{aligned} Trans(\delta_1||\delta_2, s, \delta', s') &\equiv \exists\gamma : Trans(\delta_1, s, \gamma, s') \wedge \delta' = (\gamma||\delta_2) \\ &\quad \vee \exists\gamma : Trans(\delta_2, s, \gamma, s') \wedge \delta' = (\delta_1||\gamma) \\ Final(\delta_1||\delta_2, s) &\equiv Final(\delta_1, s) \wedge Final(\delta_2, s) \end{aligned}$$

This rule specifies the concurrent-execution operator as an *interleaving* of computation steps. It states that it is possible to single-step the concurrent execution of δ_1 and δ_2 by performing either a step from δ_1 or a step from δ_2, with the remainder γ left to execute concurrently with the other program

Clearly there are two notions of concurrency to be considered in the situation calculus: the possibility of performing several actions at the same instant (*true concurrency*), and the possibility of interleaving the execution of several programs (*interleaved concurrency*). Baier and Pinto [5] have modified ConGolog to incorporate sets of concurrent actions in an attempt to integrate these two forms of concurrency. However, their semantics may call for actions to be performed that are not possible and can also produce unintuitive program behaviour in some cases. A key aspect of our work in Chapter 3 is a robust integration of these two notions of concurrency.

We have omitted many details here that are not relevant to this thesis, such as the second-order axioms necessary to handle recursive procedure definitions. We will denote by $\mathcal{D}_{golog}$ the standard axioms defining $Trans$ and $Final$ [17, 21].

Algorithm 1 The Golog/ConGolog Execution Algorithm for program δ

Find a situation s such that:

$$\mathcal{D} \cup \mathcal{D}_{golog} \models \exists s : \mathbf{Do}(\delta, S_0, s)$$

for each action in the resulting situation term **do**
  execute that action
end for

2.2.3 Execution Planning

Planning an execution of a Golog program δ can be reduced to a theorem proving task as shown in equation (2.1). Here $Trans^*$ indicates the standard second-order definition for the reflexive transitive closure of $Trans$.

$$\mathcal{D} \cup \mathcal{D}_{golog} \models \exists s, \delta' : \left[Trans^*(\delta, S_0, \delta', s) \wedge Final(\delta', s)\right] \tag{2.1}$$

A constructive proof of this query would produce an instantiation of s, a situation term giving a sequence of actions constituting a legal execution of the program. These actions are then executed one-by-one in the world. Since the program remaining after termination is often not important, the macro **Do** is re-defined in terms of $Trans$ and $Final$ to specify only the resulting situation:

$$\mathbf{Do}(\delta, s, s') \stackrel{\text{def}}{=} \exists \delta' : Trans^*(\delta, s, \delta', s') \wedge Final(\delta', s')$$

In the original Golog and in ConGolog this forms the entirety of the execution planning process, as these variants require a full legal execution to be planned before any actions are performed in the world. This is referred to as *offline execution*. The Golog execution algorithm is presented in Algorithm 1.

By contrast, IndiGolog allows agents to proceed without planning a full terminating execution of their program, instead searching for a legal next step a in the current situation σ such that:

$$\mathcal{D} \cup \mathcal{D}_{golog} \models \exists a, \delta' : Trans^*(\delta, \sigma, \delta', do(a, \sigma))$$

This next step is then performed immediately, and the process repeats until a terminating configuration is reached. This is referred to as *online execution*. The IndiGolog execution algorithm is presented in Algorithm 2.

In order to incorporate planning into this execution algorithm, IndiGolog introduces an explicit "search" operator $\Sigma(\delta)$, which can only make a transition if the

Algorithm 2 The IndiGolog Execution Algorithm for program δ

$\sigma \Leftarrow S_0$
while $\mathcal{D} \cup \mathcal{D}_{golog} \not\models Final(\delta, \sigma)$ **do**
  Find an action a and program δ' such that:

$$\mathcal{D} \cup \mathcal{D}_{golog} \models Trans^*(\delta, \sigma, \delta', do(a, \sigma))$$

  Execute the action a
  $\sigma \Leftarrow do(a, \sigma)$
  $\delta \Leftarrow \delta'$
end while

program is guaranteed to eventually terminate successfully:

$$Trans(\Sigma(\delta), s, \delta', s') \equiv \exists s'', \delta'' : Trans(\delta, s, \delta'', s') \wedge \mathbf{Do}(\delta'', s', s'') \wedge \delta' = \Sigma(\delta'')$$

This approach gives the programmer powerful control over the amount of non-determinism in the program, and the amount of planning required to find a legal execution. It also allows the programmer to avoid planning over sensing actions, which can cause an exponential blowup in planning complexity. Sensing actions are simply performed outside the scope of a search operator.

2.2.4 Extensions

There have been a wide range of Golog extensions developed which we will not consider in this thesis. Among them have been extensions to include decision-theoretic [12] and game-theoretic aspects [28, 29], additional control operators such as partially-ordered sequences of actions [6] and hierarchical task networks [34, 111], synchronisation between the individual programs of a team of agents [26], and accounting for continuous change and event triggering [36].

While we will not consider these Gologs in any detail, we do note that each has involved relatively modest extensions to the underlying situation calculus theory and/or the semantics of the Golog operators, and as a result there has been rich cross-pollination between these different works. We therefore hope that our work may in turn be combined with some of these extensions to provide an even richer formalism.

2.3 Epistemic Reasoning

Reasoning about the knowledge of an agent and the combined knowledge of a group of agents, referred to in general as *epistemic reasoning*, is a fundamental aspect of

reasoning and planning in multi-agent domains. An excellent explanation of the importance of epistemic reasoning is the classic paper by Halpern and Moses [39].

2.3.1 Epistemic Reasoning in General

The standard semantics of epistemic reasoning are given by the Kripke structures of modal logic, and are based on the idea of *possible worlds*. Intuitively, if an agent is unsure about the state of the world, then there must be several candidate states of the world that it considers possible. The agent is then said to *know* a proposition if it is true in all worlds considered possible. In modal epistemic logics this is typically written as $[K_{agt}]\phi$, but we will consistently use the standard situation calculus notation of $\mathbf{Knows}(agt, \phi, s)$.

While this works well for a static knowledge base, there is also the question of how each agent's knowledge changes over time as actions are performed in the world. Various formal systems have been devised to represent the interaction between knowledge and action, including those of Fagin et al. [25], Parikh and Ramanujam [74] and Batlag et al. [8]. Recent work on the foundations of epistemic dynamic logic has shown these different perspectives to be essentially equivalent [73, 118], and they can be briefly summarised as follows.

A system is considered to have a set of possible *events* which could occur at any given instant, and the system *state* at any time is determined by the sequence of events that have occurred. For each agent there is some subset of these events that it is capable of observing, and it thus has a restricted local *view* of the state of the system. From the agent's perspective, the system may be in any one of the states that would be compatible with its current local view. If some proposition holds in all such states, then the agent *knows* that proposition.

We will not comment on these related formalisms for epistemic reasoning in any further detail, except to note that the approach we develop in Chapters 7 and 8 deliberately parallels the intuitions and notation of this wider field.

2.3.2 Epistemic Reasoning in the Situation Calculus

Epistemic reasoning was first introduced to the situation calculus by Moore [71], and formalised extensively by Scherl and Levesque [98] whose paper is now the canonical reference. Their work has been extended to include concurrent actions [97] and multiple agents [106]. Our work further extends these techniques.

The semantics of knowledge are based on a reification of the "possible worlds" semantics of modal logic, using situation terms rather than abstract worlds. A special fluent $K(agt, s', s)$ is used to indicate that "in situation s, the agent agt

considers the alternate situation s' to be possible". The macro **Knows** is then defined as a shorthand for the standard possible-worlds definition of knowledge, stating that an agent knows ϕ when ϕ is true in all situations considered possible:

$$\mathbf{Knows}(agt, \phi, s) \stackrel{\text{def}}{=} \forall s' : K(agt, s', s) \rightarrow \phi[s'] \tag{2.2}$$

Readers familiar with modal logic will recognise $K(agt, s', s)$ as the situation calculus analogue of the modal reachability relation K_{agt}, and the macro **Knows** as the equivalent of the modal box operator $[K_{agt}]\phi$. The definitions of uniform and regressable formulae are updated to permit statements of the form $\mathbf{Knows}(agt, \phi, \sigma)$.

In preparation for developments later in the thesis, we break slightly with standard situation calculus notation and introduce an additional fluent $K_0(agt, s', s)$ to model the *initial* epistemic uncertainty of the agents, with the corresponding macro $\mathbf{Knows_0}(agt, \phi, s)$ defined in the obvious way. In synchronous domains these are always equivalent to K and **Knows**, but we introduce them here to maintain consistency of the presentation.

To model initial epistemic uncertainty, the foundational axioms Σ are modified to permit multiple initial situations, identified by the predicate $Init(s)$. S_0 then represents the *actual* initial situation, while other initial situations represent alternatives that the agents consider possible. We have the following straightforward extensions of the foundational axioms presented in Section 2.1.2:

No situation precedes an initial situation:

$$Init(s) \rightarrow \neg(s' \sqsubset s)$$

Situation terms in general are always *rooted* at a particular initial situation:

$$Init(s) \rightarrow root(s) = s$$
$$root(do(c, s)) = root(s)$$

The definition of legal situations permits any root situation, not just S_0:

$$Legal(s) \stackrel{\text{def}}{=} root(s) \leq_{Legal} s$$

Finally, the initial epistemic uncertainty of the agents is restricted to initial situations only, so that they initially know that no actions have been performed:

$$K_0(agt, s', s) \rightarrow Init(s) \wedge Init(s')$$

With this infrastructure in place, the initial situation axioms $\mathcal{D}_{S_0}$ can now contain sentences of the form $\mathbf{Knows_0}(agt, \phi, S_0)$ specifying what the agents initially know.

Since we intend to replace the standard axioms of knowledge later in the thesis, we assume that the dynamics of the K fluent are specified in a separate set of axioms $\mathcal{D}_K$, which must be included when reasoning about knowledge queries. While we defer a detailed discussion of the dynamics of K until Chapter 7, note at this stage that its behaviour is specified by a successor state axiom like the following:

$$K(agt, do(a, s'), do(a, s)) \equiv K(agt, s', s) \wedge Legal(a, s')$$

That is, each agent's knowledge is updated to account for every action that occurs. The situations considered possible *after* an action occurrence are the legal successors of situations considered possible *before* that action. As various types of knowledge-producing action are introduced to the situation calculus, the successor state axiom for K is modified to encode the intended semantics [54, 77, 106–108].

In multi-agent settings, one must also consider group-level knowledge. We briefly review the various group-level epistemic modalities commonly found in the literature; an excellent overview and discussion can be found in [39]. Let G be a finite group of agents. The basic group-level modality is "everyone knows ϕ", which is defined as:

$$\mathbf{EKnows}(G, \phi, s) \stackrel{\text{def}}{=} \forall agt \in G : \mathbf{Knows}(agt, \phi, s)$$

Since G is a finite set, this can be written equivalently as a finite conjunction:

$$\mathbf{EKnows}(G, \phi, s) \stackrel{\text{def}}{=} \bigwedge_{agt \in G} \mathbf{Knows}(agt, \phi, s)$$

To assert more complete knowledge by members of the group, one can say "everyone knows that everyone knows ϕ" by nesting **EKnows** operators. In general we have "everyone knows to depth n" defined by:

$$\mathbf{EKnows}^1(G, \phi, s) \stackrel{\text{def}}{=} \mathbf{EKnows}(G, \phi, s)$$
$$\mathbf{EKnows}^n(G, \phi, s) \stackrel{\text{def}}{=} \mathbf{EKnows}(G, \mathbf{EKnows}^{n-1}(G, \phi), s)$$

The higher the value of n, the stronger an assertion is made about the knowledge of the group. The strongest group-level modality is "it is common knowledge that ϕ". Intuitively this indicates that everyone knows ϕ, everyone knows that everyone knows

ϕ, and so on ad infinitum. Formally, it can be defined as the infinite conjunction:

$$\mathbf{CKnows}(G, \phi, s) \stackrel{\text{def}}{=} \bigwedge_{n \in \mathbb{N}} \mathbf{EKnows}^n(G, \phi, s)$$

Equivalently, it can be defined as a fixpoint or transitive closure of the **EKnows** relation. Common knowledge is an extremely powerful form of knowledge that has deep implications for coordinated group behaviour. For example, in the famous "Coordinated Attack" problem, the proof that the generals cannot coordinate an attack depends heavily on their inability to obtain common knowledge [39].

As we shall see in Chapter 8, it is surprisingly difficult to reason effectively about common knowledge, as it is not directly amenable to a standard regression rule. Our work is the first to provide an effective reasoning procedure for common knowledge in the situation calculus.

2.4 Related Formalisms

There are a range of related formalisms for reasoning about knowledge, action and change, which we do not directly consider in this thesis. Most closely related to the situation calculus are the fluent calculus of Thielscher [115] and the event calculus of Kowalski and Sergot [50].

The fluent calculus is based explicitly on the use of progression for solving the projection problem, and so maintains an explicit representation of the state of the world which is updated as actions are performed. It can be derived from a restricted variant of the situation calculus by transforming successor state axioms into state update axioms that explicitly add and remove fluents from the state [116]. Notably, it is relatively straightforward to interpret Golog programs on top of the fluent calculus [99]. As discussed in Section 2.1.3, it would be interesting to translate our regression-based ideas into a progression-based formalism such as this, but we do not consider it in this thesis.

The event calculus is slightly further removed, in that it contains a single linear timeline rather than the branching time structure of the situation calculus. This makes it more suitable for representing some domains and posing some queries, but less suitable for others; a detailed comparison can be found in [10, 51]. Like the situation calculus, it has found significant applications in a logical approach to planning and agent control [104]. One particular strength of the event calculus is in planning with partially-ordered sequences of events, highlighted by a reimplementation of Golog using the event calculus that supports partially-ordered plans [24]. We add a similar ability to the situation calculus in this thesis.

There is also the family of approaches known as "dynamic epistemic logic", which are based on modal logic [8, 118, 119] While these formalisms are typically propositional rather than first-order, and focus more on reasoning about knowledge and communication than on modelling a changing dynamic world, there are still strong similarities with the situation calculus [117]. In Chapter 8 we will adapt and extend some reasoning techniques from these formalisms to model common knowledge in the situation calculus.

There have been several attempts to combine the various action formalisms into a unifying theory of action, including [9, 51, 100, 113], but there is yet to emerge a clear standard in this regard. In the meantime, we find the notation and meta-theory of the situation calculus particularly suitable for expressing our main ideas, and find the Golog programming language to be a particularly powerful and flexible approach to specifying agent behaviour and programming shared tasks.

It is our hope that the strong underlying similarities between the major action formalisms will allow the ideas presented in this thesis to find some application or resonance beyond the specifics of the situation calculus.

2.5 Mozart/Oz

One of the main advantages of the situation calculus and Golog are their straightforward implementation as a logic program. As the dominant implementation of the logic programming paradigm, Prolog is typically used for such implementations. In this thesis we use Mozart, a multi-paradigm programming system with some unique features that are particularly suited to our work.

The Mozart system [121] is an implementation of the Oz programming language [120] with strong support for logic programming and distributed computing. While a full explanation of its features is well outside the scope of this thesis, we provide a short introduction to the subset of its features we will be using – in particular, doing Prolog-style logic programming in Oz. Familiarity with logic programming in the style of Prolog is assumed.

Terms, variables and unification in Oz work similarly to Prolog, although arguments in compound terms are separated by whitespace rather than a comma. Predicates are implemented as ordinary procedures, so all clauses for a predicate must be contained in a single procedure. Figure 2.3 shows an Oz implementation of a classic Prolog example predicate, naive list reverse.

```
proc {Reverse LIn LOut}
  case LIn of nil then
      LOut = nil
  [] H|Ts then Tr in
      {Reverse Ts Tr}
      {List.append Tr [H] LOut}
  end
end
```

Figure 2.3: Naive List Reverse implemented in Mozart/Oz

Some things to note about this example include:

- The syntax for procedure definition is **proc** $\{Name\ Arg\ \ldots\}$ $Body$ **end**
- The syntax for procedure calls is $\{Name\ Arg\ \ldots\}$
- The **case** statement is used to pattern-match the contents of a variable
- Local variables must be explicitly introduced using the keyword **in**
- Mozart separates functionality into modules, such as *List*

Procedures in Oz are deterministic by default, and there is no default search strategy for exploring different alternatives. Instead, Oz provides independent facilities for creating choicepoints and for exploring procedures that contain choicepoints. The result is a much more flexible, although sometimes syntactically more cumbersome, approach to logic programming [94].

The creation of choice points is explicit in Oz, and performed using the **choice** keyword. To demonstrate, consider another classic Prolog example: the nondeterministic list member predicate shown in Figure 2.4. In the case of the empty list, $Member$ simply fails. For a non-empty list, $Member$ explicitly creates a *choice point* with two options – either bind E to the head of the list, or bind E to a member of the tail of the list.

It is at this point that the use of Mozart for logic programming differs most from Prolog. If the $Member$ procedure is invoked directly, it will suspend its execution when the **choice** statement is reached. To resolve the nondeterminism, one must execute the procedure inside an explicit *search object*. These objects are responsible for exploring the various choicepoints until a non-failing computation is achieved. They operate by executing the procedure in a separate *computation space* through which the state of the underlying computation can be managed [101].

```
proc {Member E List}
  case List of nil then
      fail
  [] H|Ts then
      choice
          E = H
      []
          {Member E Ts}
      end
  end
end
```

Figure 2.4: Nondeterministic List Member implemented in Mozart/Oz

```
proc {Pair List1 List2 P}
  E1 E2
in
  {Member E1 List1}
  {Member E2 List2}
  P = E1#E2
end

proc {AllPairs List1 List2 AllP}
  FindP = proc {$ P}
            {Pair List1 List2 P}
          end
in
  AllP = {Search.base.all FindP}
end
```

Figure 2.5: Finding all pairs in Mozart/Oz

As a demonstration, Figure 2.5 uses the *Member* procedure to define a procedure *Pairs*, which nondeterministically selects a pair of elements from a pair of lists. The procedure *AllPairs* then uses the builtin *Search.base.all* object to find all solutions from this procedure, returning a list of all possible pairs from the two lists. By encapsulating the calls to nondeterministic procedures inside a search object, *AllPairs* will not expose any choicepoints to code that calls it.

Also of note in Figure 2.5 is the use of a *closure* over the procedure *Pair* to create the one-argument procedure *FindP*. Search objects work with a one-argument procedure, which is expected to bind its argument to a result. The dollar symbol is used to translate a statement (in this case the **proc** definition) into an expression. The value that would be bound to the dollar symbol by the statement becomes the return value of the expression, so $FindP = proc\,\{\$\,P\}$ is equivalent to $proc\,\{FindP\,P\}$.

```
proc {P_AllPairs List1 List2 AllP}
  functor FindP
  import
    MyList
  export
    Script
  define
    proc {Script P}
      E1 E2
    in
      {MyList.member E1 List1}
      {MyList.member E2 List2}
      P = E1#E2
    end
  end
  Searcher = {New Search.parallel
               init(mango:1#ssh rambutan:2#ssh)}
in
  AllP = {Searcher all(FindP $)}
end
```

Figure 2.6: Finding all pairs in parallel in Mozart/Oz

The power of this decoupled approach to nondeterminism and search becomes apparent when defining new search strategies, which can then be used to evaluate any procedure. For example, it is straightforward to implement breadth-first or iterative-deepening strategies to replace the standard depth-first traversal of the *Search.base* object [101].

Coupled with Mozart's strong support for distributed computing, these programmable search strategies offer a unique opportunity – it becomes possible to implement a parallel search object which can automatically distribute work between several networked machines. Moreover, this parallel search can be applied without modification to any nondeterministic procedure. Mozart comes with a built-in *ParallelSearch* object, which is described in detail in [102] and which is our main motivation for the use of Oz in this thesis.

To demonstrate the power of the approach, consider Figure 2.6, which describes a parallel-search version of the $AllPairs$ procedure. In this instance we define $FindP$ as a *functor*, an Oz abstraction for code that is portable between machines. This functor imports the module $MyList$ containing the procedures we defined earlier, and exports a one-argument procedure *Script* which will be executed by the parallel search object. The parallel search object *Seacher* launches one instance of Mozart on the machine "mango" and two instances on the machine "rambutan", then is asked to enumerate all solutions for $FindP$.

In Chapter 3 we will use this parallel search object to automatically share the workload of planning a Golog execution amongst a team of cooperating agents.

As a multi-paradigm programming language with significant research history, there is much more to Oz than we have described here. However, these brief examples should be sufficient for a reader well-versed in Prolog to understand the Oz code used throughout this thesis. For more information and further examples, consult the general Oz tutorial [41] or the specialised tutorial on logic programming in Oz [94], which are both available online.

Chapter 3

MIndiGolog

This chapter develops a Golog variant specifically designed for cooperative execution in multi-agent domains. Motivated by the "cooking agents" example domain, we want to allow a team of agents to cooperatively plan and perform the execution of a shared Golog program. As this chapter will demonstrate, the existing features of the situation calculus go a long way towards achieving this goal, but are ultimately limited to execution in synchronous domains.

We begin by integrating three existing extensions to the situation calculus into the semantics of Golog to better represent the dynamics of a multi-agent team. Key among these is true concurrency of actions, which is combined with the standard interleaved concurrency of ConGolog to give a flexible account of concurrent execution. An explicit notion of time is incorporated to enrich the world model and to assist in coordination between agents. Finally, the concept of natural actions is tightly integrated into the language, allowing agents to predict the behaviour of their teammates and environment. We name the resulting language "MIndiGolog" for "Multi-Agent IndiGolog".

In addition to these new Golog semantics, we develop an innovative implementation of our language using the distributed logic programming features of the Mozart platform. Utilising the parallel search facility as described in Section 2.5, the agents can transparently share the workload of planning a program execution. The ability to utilise off-the-shelf techniques such as parallel search highlights a significant advantage of building our system on the situation calculus: it has a straightforward encoding as a logic program.

Concluding the chapter is a discussion of the limitations of this first incarnation of MIndiGolog, which derive from the effective reasoning procedures of the situation calculus. Specifically, it can only operate in fully-observable, synchronous domains.

Subsequent chapters of this thesis develop new extensions to the situation calculus that work towards overcoming this limitation. So while the semantics and initial implementation of MIndiGolog are of independent interest, this chapter should also be seen as motivating the use of the situation calculus in rich multi-agent domains, and therefore the techniques developed later in this thesis.

The chapter proceeds as follows: Section 3.1 provides some more detailed background material; Section 3.2 uses the example of the cooking agents to motivate the changes we will make to the standard Golog semantics; Section 3.3 introduces the MIndiGolog semantics incorporating time, true concurrency of actions, and natural actions, and proves the legality of our modifications; Section 3.4 discusses our implementation in Mozart/Oz and shows an example execution produced using the MIndiGolog semantics; Section 3.5 discusses the use of Mozart's parallel search functionality to share the planning workload; and finally Section 3.6 discusses both the achievements and limitations of this first incarnation of the language.

3.1 Background

There are two basic approaches to the use of Golog in a multi-agent setting. The first, and most common, is to assign each agent its own individual Golog program. The behaviour of the overall system is then defined as the concurrent execution of the individual agent's programs:

$$\delta = \delta_{agt1} \,||\, \delta_{agt2} \,||\, \ldots \,||\, \delta_{agtN}$$

This is the approach followed by TeamGolog [26] and the Cognitive Agents Specification Language [110], along with earlier work in a similar vein [58]. In such a setting, the agents do not necessarily cooperate or coordinate their actions, and it is assumed that any legal execution of the combined agent programs is a possible evolution of the entire system.

The second approach, and the one we follow here, is to have all agents cooperate to plan and perform the joint execution of a single, shared program. This program would typically be the concurrent execution of several shared tasks:

$$\delta = \delta_{task1} \,||\, \delta_{task2} \,||\, \ldots \,||\, \delta_{taskN}$$

This is the approach taken by the Golog variant "ReadyLog" developed by Ferrein et al. [27] to control the behaviour of a RoboCup soccer team.

The one-program-per-agent approach can be considered a special case of the

Algorithm 3 The ReadyLog Execution Algorithm for program δ

$\sigma \Leftarrow S_0$
while $\mathcal{D} \cup \mathcal{D}_{golog} \not\models Final(\delta, s)$ **do**
  Find an action a and program δ' such that:

$$\mathcal{D} \cup \mathcal{D}_{golog} \models Trans^*(\delta, \sigma, \delta', do(a, \sigma))$$

  if the action is to be performed by me **then**
    Execute the action a
  else
    Wait for the action to be executed
  end if
  $\sigma \Leftarrow do(a, \sigma)$
  $\delta \Leftarrow \delta'$
end while

shared-task approach, one which does not require coordination or cooperation between team members. So while we focus exclusively on the cooperative execution of a shared task in this and subsequent chapters, the techniques we develop are likely to have application in the case of multiple individual control programs as well.

While Ferrein et al. focus on decision-theoretic planning rather than the rich domain extensions we consider below, the execution algorithm they have developed for ReadyLog provides an excellent introduction to the cooperative execution of Golog programs. It is presented in Algorithm 3.

The aim of the ReadyLog execution algorithm is to allow agents to coordinate their actions without the need for explicit communication. Each agent is given their own individual copy of the shared program, and they each independently execute the IndiGolog planning process to determine the next step of execution. When an agent finds a next step where the action is to be performed by the agent itself, it executes the action immediately. When the next action is to be performed by another agent, it waits for its teammate to execute the action before proceeding to the next step.

Coordination arises in this setting by ensuring that the agents use identical theorem provers (in the case of [27], identical Prolog interpreters) to determine each program step, which will generate candidate solutions in the same order for each agent. So although each agent plans the program execution steps independently, they are guaranteed to plan the *same* execution steps and their actions will therefore be coordinated without needing to communicate.

However, the semantics of ReadyLog remain largely single-agent and do not address concerns such as the possibility of performing actions concurrently, sharing the computational workload of planning, or predicting the behaviour of team members

and the environment in the face of many concurrently-executing tasks.

The semantics of MIndiGolog that we develop in this chapter follow a similar approach to ReadyLog, but we focus on incorporating rich domain features such as concurrency and continuous time. While these features have been added to Golog individually in existing works [5, 84, 90, 91], our work is the first to provide a combined integration. We also rectify some subtle problems with the semantics presented in existing work, to provide a more robust combination of rich domain features.

There are, of course, a variety of other systems and formalisms that could be used to model and implement domains such as our "cooking agents" example. A popular choice is to represent tasks using a variant of the Hierarchical Task Networks formalism, which has been employed in systems such as STEAM [112], SharedPlans [37], and TAEMS [22].

The potential advantages of Golog over HTN-based approaches are discussed in [6], where HTN-like constructs are added as operators to the Golog language. They identify the following advantages: powerful control over and composition of nondeterministic operators; a more natural representation of many tasks thanks to common programming constructs; and a more sophisticated logic of action. This chapter will add to the list: the ability to utilise off-the-shelf techniques for distributed logic programming to automatically share the execution planning workload.

The purpose of this chapter, though, is not to propose the MIndiGolog approach as the ultimate solution for programming cooperative behaviour. Rather, it serves to highlight the *potential* of the situation calculus and Golog for both modelling and implementing rich multi-agent systems.

3.2 Motivation

Recall the "cooking agents" example domain from Chapter 1 – several agents inhabit a kitchen along with various ingredients and utensils, and they must cooperate to prepare a meal. A full definition of this domain is given in Appendix C, but the specific details are not important for the purposes of this this chapter.

Specifying tasks for the cooking agents requires an interesting combination of features. There is much procedural knowledge about recipes that should be encoded as precisely as possible, while at the same time there are a lot of details of precisely who performs which steps, or precisely when they are performed, that should not be explicitly specified by the programmer.

The Golog family of languages provide a compelling formalism for specifying tasks in this domain, as the controlled nondeterminism they provide can be used to

$$
\begin{array}{c}
\textbf{proc}\, MakeSalad(dest) \\
[\pi(agt, ChopTypeInto(agt, Lettuce, dest)) \,||\, \\
\pi(agt, ChopTypeInto(agt, Carrot, dest)) \,||\, \\
\pi(agt, ChopTypeInto(agt, Tomato, dest))] \,; \\
\pi(agt, [acquire(agt, dest) \,; \\
beginTask(agt, mix(dest, 1)) \,; \\
endTask(agt, mix(dest, 1)) \,; \\
release(agt, dest)])\ \textbf{end}
\end{array}
$$

Figure 3.1: A Golog program for making a salad

$$
\begin{array}{c}
\textbf{proc}\, ChopTypeInto(agt, type, dest) \\
[\pi(board,\ IsType(board, Board)? \,; \\
\pi(obj,\ IsType(obj, type)? \,; \\
acquire(agt, board) \,; \\
acquire(agt, obj) \,; \\
placeIn(agt, obj, board) \,; \\
beginTask(agt, chop(board)) \,; \\
endTask(agt, chop(board)) \,; \\
acquire(agt, dest) \,; \\
transfer(agt, board, dest) \,; \\
release(agt, board) \,; \\
release(agt, dest)))]\ \textbf{end}
\end{array}
$$

Figure 3.2: A Golog program for chopping an ingredient

elide certain details from the program while keeping its procedural nature intact. Consider how we might specify the task of making a simple salad, shown in Figure 3.1. Using the high-level nondeterministic operators of Golog, this program says, in essence, "somebody chop a lettuce, somebody chop a carrot, and somebody chop a tomato. Then, somebody mix them together".

Note that the explicit concurrency operators allow the three ingredients to be chopped independently, while the nondeterministic "pick" operators allow any available agent to perform each sub-task. Expanding on this example, the procedure *ChopTypeInto* could be specified as shown in Figure 3.2. Here the agent must se-

lect and acquire an object of the specified type, as well as an available chopping board. He then places the object on the board, chops it, and transfers it to the destination container.

Notice that the programs do no specify which agent is to perform which task – in fact they make no assertions at all about the particular agents operating in the world. A library of procedures such as this can be combined very flexibly to specify the behaviour of the cooking agents, and the resulting program could be given to any team of agents for execution. The agents can prepare several dishes concurrently:

$$MakeSalad() \,||\, MakePasta() \,||\, MakeCake()$$

They can even plan to have different dishes ready at different times:

$$([MakeSalad() \,||\, MakePasta()] \;;\; ?(time < 7:30)) \\ ||\; (MakeCake() \,;\, ?(8:15 < time < 8:30))$$

A legal execution of these programs must select appropriate ingredients and utensils, and ensure that are acquired in an appropriate order, so that it can proceed to completion without the agent's actions interfering with each other or coming into conflict over shared resources. Following the ReadyLog execution algorithm, each agent plans to avoid such conflict by virtue of finding a legal execution.

In short, the situation calculus and Golog provide an extremely powerful formalism for specifying cooperative agent behaviour in domains such as this.

However, executing these kinds of program using a standard Golog variant is far from ideal in a multi-agent setting. To illustrate this we have executed the *MakeSalad* program using the standard semantics of IndiGolog [17], augmented with an explicit temporal component in the style of [90]. The resulting execution is shown in Figure 3.3.

In this domain there are three agents *Joe*, *Jon* and *Jim*. While the resulting execution is legal, it is suboptimal in several ways. The most obvious problem is that it does not take advantage of the concurrency inherent in a team of agents. Only a single agent acts at any one time, while the other agents are idle. Ideally the execution planner should exploit *true concurrency* where possible, by calling for multiple actions to be performed at each timestep.

Another shortcoming of the standard semantics in this setting is that they have no support for predicting natural actions. This means we must specify unnecessary

```
do([acquire(jim lettuce1)] at 1)
do([acquire(jim board1)] at 2)
do([placeIn(jim lettuce1 board1)] at 3)
do([beginTask(jim chop(board1))] at 4)
do([acquire(joe tomato1)] at 5)
do([acquire(joe board2)] at 6)
do([endTask(jim chop(board1))] at 7)
do([acquire(jim bowl1)] at 8)
do([transfer(jim board1 bowl1)] at 9)
do([release(jim bowl1)] at 10)
do([release(jim board1)] at 11)
do([placeIn(joe tomato1 board2)] at 12)
do([beginTask(joe chop(board2))] at 13)
do([acquire(jim carrot1)] at 14)
do([acquire(jim board1)] at 15)
do([endTask(joe chop(board2))] at 16)
do([acquire(joe bowl1)] at 17)
do([transfer(joe board2 bowl1)] at 18)
do([release(joe bowl1)] at 19)
do([release(joe board2)] at 20)
do([placeIn(jim carrot1 board1)] at 21)
do([beginTask(jim chop(board1))] at 22)
do([endTask(jim chop(board1))] at 25)
do([acquire(jim bowl1)] at 26)
do([transfer(jim board1 bowl1)] at 27)
do([release(jim bowl1)] at 28)
do([release(jim board1)] at 29)
do([acquire(jim bowl1)] at 30)
do([beginTask(jim mix(bowl1 1))] at 31)
do([endTask(jim mix(bowl1 1))] at 34)
do([release(jim bowl1)] at 35)
```

Figure 3.3: Execution of $MakeSalad$ program using IndiGolog semantics

details in our programs, such as including the $endTask$ action in the definition of $MakeSalad$. Since these actions are predictable, the planner should incorporate them automatically – particularly in a multi-agent setting where there may be many natural actions associated with long-running tasks.

The remainder of this chapter is dedicated to developing a robustly multi-agent Golog semantics to overcome these issues, as well as an implementation that can highlight the benefits of this approach.

3.3 Semantics of MIndiGolog

We have integrated three extensions to the situation calculus with the existing semantics of IndiGolog to better model the dynamics of a multi-agent team. These extensions allow agents to represent time, concurrently-occurring actions, and natu-

ral actions in a robust way. We use the extensions to the situation calculus discussed in Chapter 2 to model concurrent actions (Section 2.1.4) and explicit time (Section 2.1.4). The arguments to *do* will therefore be CONCURRENT#TIMEPOINT pairs.

Why do we focus on these three extensions in particular? We consider the ability to reason about true concurrency of actions to be fundamental in planning for multi-agent domains, if only to ensure that actions cannot come into conflict if performed concurrently. Once concurrent actions are being considered, it makes sense to then take advantage of them where possible for concurrent program execution.

Natural actions form an important part of rich multi-agent domains with many long-running tasks, as the $endTask$ action is a natural action. Since they are entirely predictable based on the theory of action, we contend that Golog should provide support for them at the semantic level to relieve the programmer from having to specify them explicitly.

An explicit temporal component is a prerequisite for supporting natural actions, and also provides for a much more convincing domain model, particularly with the cooking agents example domain.

3.3.1 Time

The semantics of IndiGolog are straightforwardly modified to accommodate an explicit temporal component. We follow the approach of [90, 91], in which the transition rule for a single-action program is modified to use $Legal$ instead of $Poss$. Recall that the original transition rule for this case is:

$$Trans(a, s, \delta', s') \equiv Poss(a, s) \wedge \delta' = Nil \wedge s' = do(a, s)$$

Modifying this to use CONCURRENT#TIMEPOINT pairs and $Legal$, we obtain:

$$Trans(a, s, \delta', s') \equiv \exists t : Legal(\{a\}\#t, s) \wedge \delta' = Nil \wedge s' = do(\{a\}\#t, s) \quad (3.1)$$

This basically ensures that the temporal component respects the ordering between predecessor and successor situations. The key aspect here is not the new transition rule, but the use of a linear constraint solver to reason about time. The situations produced by the Golog execution process are not fixed situation terms, but contain timepoint variables that are constrained relative to each other [90].

For example, consider the following simple program:

$$placeIn(Jim, Flour, Bowl)\,;\, placeIn(Jim, Sugar, Bowl)$$

One output from a temporal Golog execution planner for this program would be:

$$do(\{placeIn(Jim, Sugar, Bowl)\}\#t_2, do(\{placeIn(Jim, Flour, Bowl)\}\#t_1, S_0))$$

Here t_1 and t_2 are variables giving the execution times of each action. It would also output the following constraint on this solution:

$$t_2 > t_1$$

This solution thus represents a family of legal executions that respect the temporal ordering on situations. Since we intend for these executions to be performed cooperatively, the agents must agree in advance on a convention for grounding these variables, for example by setting them to their smallest possible value. To simplify the presentation, we have done this in the executions shown throughout the chapter.

3.3.2 Concurrency

To take advantage of true concurrency, we must first allow sets of concurrent actions to appear in a MIndiGolog execution. The modified $Trans$ clause for primitive actions shown in equation 3.1 already ensures that sets of concurrent actions are performed rather than raw action terms.

However, this is clearly not enough to truly exploit the potential for concurrency in a multi-agent team. As noted by Baier and Pinto [5], the concurrency operator should be modified to accept a concurrent transition from both programs. The concurrency semantics they propose for their variant "TConGolog" are shown below:

$$\begin{aligned} Trans(\delta_1||\delta_2, s, \delta', s') \equiv \quad & \exists\gamma : Trans(\delta_1, s, \gamma, s') \wedge \delta' = (\gamma||\delta_2) \\ & \vee \exists\gamma : Trans(\delta_2, s, \gamma, s') \wedge \delta' = (\delta_1||\gamma) \\ & \vee \exists c_1, c_2, \gamma_1, \gamma_2 : Trans(\delta_1, s, \gamma_1, do(c_1, s)) \\ & \wedge Trans(\delta_2, s, \gamma_2, do(c_2, s)) \wedge \delta' = (\gamma_1||\gamma_2) \wedge s' = do(c_1 \cup c_2, s) \end{aligned}$$

The first two lines are the standard rules for the concurrency operator, encoding the interleaving of steps from programs δ_1 and δ_2. The remaining lines permit the concurrent execution of a transition from both programs. While this modification will take advantage of the true concurrency of actions present in multi-agent domains, it introduces several complications that [5] does not address.

First, precondition interaction means that $c_1 \cup c_2$ may not be possible even if the individual actions are. The transition clause must ensure that the combination of the two sets of actions is possible. Another issue arises when two programs can

legitimately be transitioned by executing the same action. Consider the following programs which add ingredients to a bowl:

$$\delta_1 = placeIn(Jim, Flour, Bowl)\,;\, placeIn(Jim, Sugar, Bowl)$$
$$\delta_2 = placeIn(Jim, Flour, Bowl)\,;\, placeIn(Jim, Egg, Bowl)$$

Executing $\delta_1||\delta_2$ should result in the bowl containing two units of flour, one unit of sugar and an egg. However, an individual transition for both programs is $c_1 = c_2 = \{placeIn(Jim, Flour, Bowl)\}$. Naively executing $c_1 \cup c_2$ to transition both programs would result in only one unit of flour being added.

Alternately, consider two programs waiting for a timer to ring:

$$\delta_1 = ringTimer\,;\, acquire(Jim, Bowl)$$
$$\delta_2 = ringTimer\,;\, acquire(Joe, Board)$$

Both programs should be allowed to proceed using the same $ringTimer$ occurrence, because it is an environmental effect rather than a purposeful agent-initiated action.

In simple cases like these, it is easy for the programmer to see the potential for such undesirable interaction and adjust their programs accordingly. But in more complex cases with liberal use of nondeterminism, it may not be possible to predict what actions can potentially be executed concurrently. To avoid unintuitive (and potentially dangerous) behaviour, concurrent execution must not be allowed to transition both programs using the same *agent-initiated* action. Natural actions can safely transition two concurrent programs.

Taking these factors into account, and including an explicit temporal component, we develop the improved transition rule for concurrency in equation (3.2). The first two lines are the original interleaved concurrency clause from ConGolog, while the remainder characterises the above considerations to take advantage of true concurrency.

$$\begin{aligned} Trans(\delta_1||\delta_2, s, \delta', s') \equiv \quad & \exists\gamma : Trans(\delta_1, s, \gamma, s') \wedge \delta' = (\gamma||\delta_2) \\ & \vee \exists\gamma : Trans(\delta_2, s, \gamma, s') \wedge \delta' = (\delta_1||\gamma) \\ \vee \exists c_1, c_2, \gamma_1, \gamma_2, t : Trans(\delta_1, s, \gamma_1, do(c_1\#t, s)) & \wedge Trans(\delta_2, s, \gamma_2, do(c_2\#t, s)) \\ \wedge Legal((c_1 \cup c_2)\#t, s) & \wedge \forall a : [a \in c_1 \wedge a \in c_2 \rightarrow Natural(a)] \\ \wedge \delta' = (\gamma_1||\gamma_2) & \wedge s' = do((c_1 \cup c_2)\#t, s) \end{aligned} \quad (3.2)$$

There are two other Golog operators that relate to concurrency: the prioritised concurrency operator $\delta_1 \ll \delta_2$, and the concurrent iteration operator $\delta^{||}$. MIndiGolog leaves both of these operators unmodified. For the concurrent iteration operator this is clearly the right thing to do, since its standard semantics are defined in terms of the base concurrency operator and will automatically inherit our new ability to take advantage of true concurrency.

For the prioritised concurrency operator, we note that it is mainly used to implement interrupt-handling by blocking the execution of the higher-priority program until a test condition is satisfied, at which point the lower-priority program is blocked until the interrupt handler is complete. It makes no sense to allow concurrent execution of both programs in this case, which would destroy these blocking semantics.

3.3.3 Natural Actions

While planning with natural actions has previously been done in Golog [84], the programmer was required to explicitly check for any possible natural actions and ensure that they appear in the execution. We significantly lower the burden on the programmer by guaranteeing that all legal program executions result in legal situations, inserting natural actions into the execution when they are predicted to occur. MIndiGolog agents will thus plan for the occurrence of natural actions without having them explicitly mentioned in the program. They may optionally be included in the program as a synchronisation device, instructing the agents to wait for the action to occur before proceeding.

This is achieved using a new $Trans$ clause for the case of a single action a, as shown in equation (3.3). If s has an LNTP t_n and corresponding set of pending natural actions c_n, a transition can be made in one of three ways: perform a at a time before t_n (fourth line), perform it along with the natural actions at t_n (fifth line), or wait for the natural actions to occur (sixth line). If there is no LNTP, then c may be performed at any legal time.

$$
\begin{aligned}
&Trans(a, s, \delta', s') \equiv \\
&\qquad \exists t, t_n, c_n : \mathbf{LNTP}(s, t_n) \wedge \mathbf{PNA}(s, c_n) \wedge \\
&\qquad \big[t < t_n \wedge Legal(\{a\}\#t, s) \wedge s' = do(\{a\}\#t, s) \wedge \delta' = Nil \\
&\qquad \vee Legal((\{a\} \cup c_n)\#t_n, s) \wedge s' = do((\{a\} \cup c_n)\#t_n, s) \wedge \delta' = Nil \\
&\qquad \vee s' = do(c_n\#t_n, s) \wedge \delta' = a\big] \\
&\vee \neg\exists t_n : \mathbf{LNTP}(s, t_n) \wedge \exists t : Legal(\{a\}\#t, s) \wedge s' = do(\{a\}\#t, s) \wedge \delta' = Nil
\end{aligned}
\tag{3.3}
$$

The occurrence of natural actions may also cause test conditions within the program to become satisfied, so a new $Trans$ clause for the test operator is also required as shown in equation (3.4). This permits a program consisting of a single test condition to make a transition if the test is satisfied, or a natural action occurs:

$$\begin{aligned} Trans(\phi?, s, \delta', s') \;\equiv\; & \phi[s] \wedge \delta' = Nil \wedge s' = s \\ & \vee\; \exists t_n, c_n : \mathbf{LNTP}(s, t_n) \wedge \mathbf{PNA}(s, c_n) \wedge \delta' = \phi? \wedge s' = do(c_n \# t_n s) \end{aligned} \quad (3.4)$$

A MIndiGolog execution will thus contain all natural actions that will occur, regardless of whether they were considered explicitly by the programmer. Contrast this with the standard handling of exogenous events in Golog, which is achieved by executing a concurrent program that generates exogenous actions:

$$\delta_{main} \,||\, (\pi(a)(Exog(a)?\,;\, a))^*$$

This allows the program to make legal transitions that contain exogenous actions, but does nothing to *predict* what exogenous actions will occur. Our approach allows the agents to directly predict the natural actions that will occur and automatically include them in a planned execution. Of course, the standard Golog approach is still required if there are also unpredictable exogenous actions to be accounted for.

3.3.4 Legality of the Semantics

Let us denote by $\mathcal{D}_{mgolog}$ the standard Golog axioms $\mathcal{D}_{golog}$, modified according to equations (3.2), (3.3) and (3.4). All legal executions of a MIndiGolog program derived from such a theory of action produce legal situations.

Lemma 1. *The semantics of MIndiGolog entail:*

$$\mathcal{D} \cup \mathcal{D}_{mgolog} \models \forall s, s', \delta, \delta' :\; Legal(s) \wedge Trans(\delta, s, \delta', s') \rightarrow Legal(s')$$

Proof. By induction on the structure of δ. The $Trans$ clauses of equations (3.2), (3.3) and (3.4) all assert that $Legal(c\#t, s)$ before constructing a new situation term $do(c\#t, s)$. The other $Trans$ clauses either assert that $s = s'$, or set s' to a new situation term constructed by one of these three base cases. Since we are given that s is legal, the lemma holds for each $Trans$ clause in the semantics. □

Theorem 1. *The semantics of MIndiGolog entail:*

$$\mathcal{D} \cup \mathcal{D}_{mgolog} \models \forall s', \delta, \delta' : Trans^*(\delta, S_0, \delta', s') \rightarrow Legal(s')$$

Thus, all legal executions of a MIndiGolog program produce legal situations.

Proof. By induction on situation terms. For the base case, S_0 is always legal by definition. Lemma 1 immediately provides legality for the inductive case by the transitivity of $Trans^*$. □

The MIndiGolog semantics are thus a powerful but robust extension to the standard semantics of the Golog language family. Our integration of concurrent actions allows the language to more accurately reflect the concurrency present in multi-agent teams while ensuring actions are performed in a definite order if they are not legal to perform concurrently. Natural actions are automatically predicted and inserted into the execution at appropriate times.

3.4 Implementation

With these new semantics in place, it is now possible to build a multi-agent execution planning system utilising MIndiGolog to specify the tasks to be performed. We have followed the style of [17, 21] to build an interpreter for our language in Oz on the Mozart programming platform [122]. We summarise our implementation below; details on obtaining the full source code are available in Appendix B.

Programs and actions are represented in Oz as record terms in a similar way to Prolog data terms. For example, the program:

$$\pi(agt, [acquire(agt, Bowl1); acquire(agt, Lettuce1)])$$

is represented as follows:

```
pick(agt seq(acquire(agt bowl1) acquire(agt lettuce1)))
```

Since "do" is a reserved keyword in Oz, we represent situation terms as records of the form $res(C\ T\ S)$. $Trans$ and $Final$ have a straightforward encoding as Oz procedures, using the **case** statement to encode each individual clause using pattern matching, and the **choice** statement to explicitly introduce choice points. The following are a selection of the operators as they appear in our implementation:

```
proc {Trans D S Dp Sp}
  case D of nil then fail
  []   test(Cond) then choice {Sitcalc.holds Cond S}
                              Sp=S Dp=nil
                       []     Tn = {Sitcalc.lntp S}
                              Cn = {Sitcalc.pna S} in
                              Dp=D Sp=res(Cn Tn S)
                       end
  []   seq(D1 D2) then choice D1r in {Trans D1 S D1r Sp}
                                     Dp=seq(D1r D2)
                       []            {Final D1 S}
                                     {Trans D2 S Dp Sp}
                       end
  []   choose(D1 D2) then choice {Trans D1 S Dp Sp}
                          []     {Trans D2 S Dp Sp}
                          end
  [] ... <additional cases ommitted> ...
  end
end
```

```
proc {Final D S}
  case D of nil then skip
  [] test(Cond) then fail
  [] seq(D1 D2) then {Final D1 S}
                     {Final D2 S}
  [] choose(D1 D2) then choice {Final D1 S}
                        []     {Final D2 S}
                        end
  [] ... <additional cases ommitted> ...
  end
end
```

The calls to *Sitcalc.holds* etc here perform standard regression-based theorem proving in the style of Prolog-based Golog implementations [21, 62]. Of particular interest is our implementation of the concurrency operator, reflecting the new semantics from equation (3.2). First, we introduce a procedure *Step* which calculates a series of transitions to produce a single next action:

```
proc {Step D S Dp Sp}
  choice Sp=res(_ _ S) {Trans D S Dp Sp}
  []     Dr in {Trans D S Dr S} {Step Dr S Dp Sp}
  end
end
```

Then we can encode the semantics of concurrency using the following case:

```
proc {Trans D S Dp Sp}
  [] ...
  []   conc(D1 D2) then
          choice D1r D2r C1 C2 Cu T in
             {Step D1 S D1r res(C1 T S)}
             {Step D2 S D2r res(C2 T S)}
             {LP.neg proc {$} A in
               {LP.member A C1} {LP.member A C2}
               {LP.neg proc {$} {Domain.isNatural A} end}
             end}
             {LP.union C1 C2 Cu}
             {Sitcalc.legal Cu T S}
             Sp=res(Cu T S) Dp=conc(D1r D2r)
          []     D1r in {Trans D1 S D1r Sp}
                        Dp=conc(D1r D2)
          []     D2r in {Trans D2 S D2r Sp}
                        Dp=conc(D1 D2r)
          end
  [] ...
end
```

The first option presented by the choicepoint is the case for true concurrency of actions, while the two other choices represent interleaved concurrency. By putting the true-concurrency case first, a depth-first search will try to find a concurrent step before looking for a step from only one of the programs. Using *Step* instead of $Trans$ allows this search to consume empty transitions of each program, such as test conditions, to find a valid concurrent step. This is valid since the empty transitions consumed by *Step* are also generated by the interleaved concurrency cases.

These simple implementation details are enough to ensure a high degree of concurrency in the generated executions, as we shall demonstrate in the next section.

A procedure $Do(\delta, s, s') \equiv Trans^*(\delta, s, \delta', s') \wedge Final(\delta', s')$ is defined that determines a complete legal execution Sp for a given program D. As discussed in Section 2.2.3, this is used to define the semantics of the search operator.

```
proc {TransStar D S Dp Sp}
  choice  Dp=D Sp=S
  []  Dr Sr in {Trans D S Dr Sr} {TransStar Dr Sr Dp Sp}
  end
end

proc {Do D S Sp}
   Dp in
   {TransStar D S Dp Sp}
   {Final Dp Sp}
end
```

Our implementation of the search operator avoids recalculating the complete legal execution by translating it into a direct list of actions to be performed. More sophisticated implementations can further instrument this case to perform failure detection and re-planning [57], but we do not include these techniques in our work.

```
proc {Trans D S Dp Sp}
  [] ...
  []   search(D1) then Sr Dr in
                  {Trans D1 S Dr Sp}
                  {Do Dr Sp Sr}
                  Dp = dosteps({Sitcalc.toStepsList Sp Sr})
  []   dosteps(Steps) then C T Steps2 in
                  Steps = (C#T)|Steps2
                  Sp = res(C T S)
                  Dp = dosteps(Steps2)
  [] ...
end
```

Using this implementation, a team of agents can plan and perform the execution of a shared MIndiGolog program by following the ReadyLog execution algorithm from Algorithm 3. First we define procedures to detect program termination and to plan a next program step, which use the built-in search object to resolve choicepoints:

```
proc {IsFinal D S B} F in
  F = {Search.base.one proc {$ R}
          {MIndiGolog.final D S} R=unit
      end}
  if F == nil then B=false else B=true end
end

proc {NextStep D S Dp Sp}
  [Dp#Sp] = {Search.base.one proc {$ R} DpR SpR in
                {MIndiGolog.step D S DpR SpR}
                R = DpR#SpR
            end}
end
```

The main control loop is then implemented in the procedure *Run* as shown below. As in ReadyLog, each agent individually executes this control loop. When the next step contains an action that is to be performed be the agent, they execute it at the indicated time. Otherwise, they wait for the actions to be executed by their teammates before proceeding to the next iteration.

```
proc {Run D S}
  if {IsFinal D S} then
    {Control.log succeeded}
  else Dp Sp C T in
      try {NextStep D S Dp Sp}
          Sp = res(C T S)
          T = {Time.min T}
          {Control.execute C T}
          {Run Dp Sp}
      catch _ then
        {Control.log failed}
      end
  end
end
```

The effect of our new semantics can be seen in Figure 3.4. This shows the execution generated by our implementation for the *MakeSalad* program from Figure 3.1, using the new semantics of MIndiGolog in a domain with three agents. Note the execution of several actions at each timestep, demonstrating the integration of true concurrency into the language. Since there are three agents but only two chopping boards, in this execution *Jon* must wait until a board becomes available at time 12 before he can begin processing his ingredient. This execution is clearly more suited to a multi-agent domain than that produced by the standard IndiGolog semantics.

```
do([acquire(jim lettuce1) acquire(joe tomato1) acquire(jon carrot1)] at 1)
do([acquire(jim board1) acquire(joe board2)] at 2)
do([placeIn(jim lettuce1 board1) placeIn(joe tomato1 board2)] at 3)
do([beginTask(jim chop(board1)) beginTask(joe chop(board2))] at 4)
do([endTask(jim chop(board1)) endTask(joe chop(board2))] at 7)
do([acquire(jim bowl1)] at 8)
do([transfer(jim board1 bowl1)] at 9)
do([release(jim bowl1)] at 10)
do([release(jim board1) acquire(joe bowl1)] at 11)
do([transfer(joe board2 bowl1) acquire(jon board1)] at 12)
do([release(joe bowl1) placeIn(jon carrot1 board1)] at 13)
do([release(joe board2) beginTask(jon chop(board1))] at 14)
do([endTask(jon chop(board1))] at 17)
do([acquire(jon bowl1)] at 18)
do([transfer(jon board1 bowl1)] at 19)
do([release(jon bowl1)] at 20)
do([release(jon board1)] at 21)
do([acquire(jim bowl1)] at 22)
do([beginTask(jim mix(bowl1 1))] at 23)
do([endTask(jim mix(bowl1 1))] at 26)
do([release(jim bowl1)] at 27)
```

Figure 3.4: Execution of *MakeSalad* program using MIndiGolog semantics

3.5 Distributed Execution Planning

One powerful feature of Mozart is the ability to use several networked computers to search for solutions to a logic program in parallel. Since the task of planning a MIndiGolog execution is encoded as a logic program, this immediately allows a team of agents to distribute the execution planning workload.

There are non-trivial computational and communication overheads involved in such a search, so it must be used judiciously. We argue that parts of the program appearing outside the scope of a search operator are intended for on-line execution, and so will tend to require little deliberation by the agent. By contrast, program components enclosed in a search operator are intended to require significant planning to generate a legal execution. We therefore modify the search operator to perform distributed execution planning, but leave the rest of the code intact.

To coordinate the parallel search, we designate one agent to be the *team leader*, who will be responsible for managing the search process; the choice of agent is arbitrary and is simply a coordination device. The code below shows our implementation of the search operator using parallel search. If the executing agent is the team leader, it executes the procedure *ParallelDo* and, when a plan is found, sends the details to the other members of the team. Subordinate team members simply wait for the plan to be received before continuing.

```
proc {Trans D S Dp Sp}
  [] ...
  [] search(D1) then Sr Dr in
          if Control.teamMember == Control.teamLeader then
               try
                 {Trans D1 S Dr Sp}
                 {ParallelDo Dr Sp Sr}
                 Dp = dosteps({Sitcalc.toStepsList Sp Sr})
                 {Control.sendMessage Dp#Sp}
               catch failure then
                 {Control.sendMessage plan_failed} fail
               end
           else Msg in
               {Control.waitForMessage Msg}
               if Msg == plan_failed then fail
               else (Dp#Sp) = Msg end
           end
  [] ...
end
```

Note that this modification is completely transparent to the rest of the implementation. While some details of sending messages are not shown, the core imple-

mentation of the ParallelDo procedure is shown below.

```
{IParallelDo D S Sp}
  PDo PSearch Ds Ss Machines
in
  Ds = {LP.serialize D}
  Ss = {LP.serialize S}
  functor PDo
    import
      MG at '/path/to/MIndiGolog.ozf'
      LP at '/path/to/LP.ozf'
    export
      Script
  define
      proc {Script R}
        Dl Sl Spl
      in
        {LP.unserialize Ds Dl}
        {LP.unserialize Ss Sl}
        {MG.'do' Dl Sl Spl}
        R = {LP.serialize (Dl#Sl#Spl)}
      end
  end
  % this constructs a record like: init(jim:1#ssh joe:1#ssh)
  Machines = {Record.make init Control.agents}
  for Agt in {Record.arity Machines} do
    Machines.Agt = 1#ssh
  end
  PSearch = {New Search.parallel Machines}
  [(D#S#Sp)] = {LP.unserialize {PSearch one(PDo $)}}
end
```

This code packages up the task to be performed as a *functor*, a portable piece of code that can be shared across the network by all team members. Input terms are serialised to a textual representation since variables cannot be exported in functor definitions. It is then a simple matter of creating a new *ParallelSearch* object that spans all team members, and asking it for a solution to the functor. When this code is executed by the team leader, it will utilise the computational resources of all team members to plan the execution of the enclosed MIndiGolog program.

As a brief demonstration of the advantages provided by this technique, consider the suggestively-named program "HardToPlan" shown in Figure 3.5. This program asks the agents to nondeterministically select and acquire objects of a variety of types, and then tests whether certain specific objects have been acquired. It has been constructed so that a single bad choice in the early stages of execution planning

proc $HardToPlan()$
$[AcquireType(Joe, Carrot)$;
$AcquireType(Jon, Sugar)$;
$AcquireType(Jim, Lettuce)$;
$AcquireType(Joe, Flour)$;
$AcquireType(Jon, Flour)$;
$HasObject(Jon, Carrot3)?$;
$HasObject(Joe, Flour5)?$;
$HasObject(Jon, Sugar4)?]$ **end**

proc $AcquireType(Agt, Type)$
$[\pi(obj,\ IsType(obj, Type)?$;
$acquire(Agt, obj))]$ **end**

Figure 3.5: A Golog program for which execution planning is difficult

	Individual Search	**Parallel Search**	**Ratio Indiv/Para**
Run 1	29.30	13.67	2.14
Run 2	29.21	11.25	2.59
Run 3	29.08	11.61	2.50
Average	29.19	12.17	2.39

Table 3.1: Execution planning times for $HardToPlan$, in seconds

- for example, having Joe acquire $Carrot1$ instead of $Carrot3$ - can invalidate all choices subsequently made. Planning a legal execution of this program thus requires a significant amount of backtracking and should benefit greatly from parallelisation.

We used our MIndiGolog implementation to execute $\Sigma(HardToPlan)$ in two different ways: using parallel search as described above, and having the team leader search for a legal execution on its own. The program for team leader Jon was executed on an AMD Athlon 64 3000+, while the subordinate team members Jim and Joe each executed on one core an Intel Core2 Duo 1.8 GHz. Three test runs were performed for both parallel and individual search. The times required to find a legal execution are shown in Table 3.1 along with the speedup factor achieved by using parallel search.

These results shown an impressive decrease in execution planning time with the use of parallel search - close to a factor of three speedup using the computational resources of three agents. Of course, on programs where execution planning is less

difficult this advantage will be reduced, but on difficult problems it can clearly provide a significant benefit.

The ability to implement this distributed execution planning with so little code, and in a way that is completely transparent to the rest of the implementation, highlights one of the major advantages of using the situation calculus and Golog – the ability to encode both the domain and the execution planning problem as a simple logic program, which is then amenable to off-the-shelf techniques for distributed logic programming.

3.6 Discussion

Our work in this chapter has integrated several important extensions to the situation calculus and Golog to better model the dynamics of multi-agent teams. Specifically, MIndiGolog combines true and interleaved concurrency, an explicit account of time, and seamless integration of natural actions. As we have shown by comparison to IndiGolog, it defines legal executions of Golog programs that are much more suitable for cooperative execution by a multi-agent team.

We have also demonstrated an innovative implementation of MIndiGolog using Mozart/Oz instead of Prolog. Since the situation calculus and Golog have a straightforward encoding as a logic program, the off-the-shelf techniques for distributed logic programming provided by the Mozart platform can be used to transparently share the execution planning workload between team members.

The approach presented in this chapter is by no means a complete account of cooperative execution in multi-agent domains. It pre-supposes the existence of a fixed team of agents and their mutual commitment to executing a pre-specified shared task. Our implementation therefore focuses entirely on planning and performing the execution of such a task, without explicit mental attitudes such as cooperation or commitment. However, this technique could easily form a component of a larger multi-agent system based on the situation calculus, such as [48, 110].

The purpose of this chapter is not to propose the MIndiGolog approach as the ultimate solution for programming cooperative behaviour. Rather, it serves to highlight the *potential* of the situation calculus and Golog for both modelling and implementing rich multi-agent systems. Unfortunately, this potential has traditionally been limited by some restrictions on the effective reasoning procedures of the situation calculus, and our implementation of MIndiGolog has inherited those limitations.

Most fundamentally, the output of our execution planning process assumes that the domain is synchronous and that all actions are publicly observable. This assumption is necessary to allow reasoning using standard regression techniques, as it

means the agents do not need to consider arbitrarily-long sequences of hidden actions. It also means that simple situation terms suffice as the output of the planning process; agents have all the information they need to coordinate the performance of concurrent actions, and to ensure that actions are performed in the correct order. For example, the execution in Figure 3.4 calls for Jim to release $Bowl1$ and then for Joe to acquire it in the next timestep, implicitly assuming that the two agents are able to coordinate and synchronise these actions.

Consider, by contrast, an asynchronous domain in which some actions are not public. If Joe is not be able to observe the occurrence of Jim releasing the bowl, he would not know when to proceed with acquiring it and execution of the plan would fail. To be sure that the plans produced by our system can be executed in the real world, we must assume that the agents execute their actions in lock-step and always know the current state of execution – in other words, that there is some form of constant synchronisation between the agents.

Another limitation is that MIndiGolog does only *linear* planning, and has no support for sensing actions. In the single-agent case the execution of Golog programs that include sensing actions is well understood [17, 52], but there is no straightforward way to adapt these techniques to the implicit coordination scheme used by MIndiGolog. The difficulty arises from the execution algorithm's crucial assumption that all agents have access to the same information. Introducing sensing actions to MIndiGolog will require more explicit reasoning about coordination based on the local information available to each agent.

Existing situation calculus techniques for reasoning about the local perspective of each agent are based on explicit notions of knowledge, as described in Section 2.3. But even these formalisms are limited to synchronous domains, requiring agents to always know how many actions have occurred so that standard regression techniques can be applied. The situation calculus currently offers no tools to extend the MIndiGolog approach into asynchronous domains. The remainder of this thesis is devoted to developing the foundations for such tools, based on a formal characterisation of the local perspective of each agent that explicitly deals with the hidden actions inherent in an asynchronous domain.

While we will revisit the cooking agents again in Chapter 5, our implementation of a new, asynchronous version of MIndiGolog is still ongoing. At the conclusion of this thesis, we will discuss the challenges remaining to be faced in bringing our new techniques for representing and reasoning about asynchronous domains together with a practical implementation of a system such as MIndiGolog.

Chapter 4

Observations and Views

This chapter develops an explicit formalisation of the local perspective of each agent, representing it using concrete terms in the logic, so that we can approach reasoning and planning in asynchronous domains in a systematic way.

Existing work on multi-agent domains in the situation calculus has left this agent-local perspective largely implicit; for example, it is customary to introduce different kinds of sensing or communication actions by directly modifying the axioms that define the dynamics of knowledge. We choose instead to reify the local perspective of each agent by explicitly talking about what it has observed, independent of how this information will be used by the rest of the action theory.

The basic idea is as follows: each occurrence of an action results in an agent making a set of *observations*. Every situation then corresponds to a local *view* for that agent: the sequence of all its observations, excluding cases where the set of observations was empty. These form agent-local analogues to standard action and situation terms, which represent the global state of the world. Allowing the set of observations to be empty lets us model truly asynchronous domains, in which an agent's local view is not always updated when the state of the world is changed.

By having views as explicit terms in the logic, we are then in a position to ensure that agents only reason and act based on their local information. Having factored out the precise details of each agent's local view, we can develop reasoning techniques and tools that can be applied in a variety of different domains, rather than depending on any particular details of how actions are perceived by each agent.

To demonstrate the appeal of this decoupling, we show how a variety of domain dynamics can be modelled using our approach. The techniques we subsequently develop using this foundation of observations and views - our planning formalism using joint executions, our account of knowledge with hidden actions, our regression

rule for common knowledge - can be used unmodified in any of these domains.

The chapter begins with additional background material in Section 4.1, discussing how an agent's local information is traditionally modelled in the situation calculus. We then define the notion of observations and views in Section 4.2, and discuss how they can be specified within the structure of a basic action theory. Section 4.3 discusses how some common domain dynamics can be modelled using observations and views, while Section 4.4 demonstrates the power and flexibility of the approach by axiomatising more complex domains that have not previously been approached in the situation calculus. Section 4.5 discusses how agents can reason using their local view, and Section 4.6 concludes with some general discussion.

4.1 Background

In many single-agent applications of the situation calculus, there is no need to consider the local perspective of the agent – since the agent has complete knowledge and is the only entity acting in the world, its local information is precisely equivalent to the global information captured by the current situation term.

If the agent has incomplete knowledge about the state of the world, it may need to perform *sensing actions* to obtain additional information [17, 98]. To represent such actions, a new sort RESULT is added to $\mathcal{L}_{sitcalc}$, along with an action description function $SR(a, s) = r$ that specifies the result returned by each action. The agent's local perspective on the world is then given by a *history*, a sequence of $a\#r$ pairs giving each action performed and its corresponding sensing result.

When sensing actions are used in IndiGolog [17], the agent must plan its execution using this history rather than a raw situation term. This is accomplished without any modifications to the underlying theory of action, by handling the history as a purely meta-level structure and modifying the way queries are posed.

First, a pair of macros are defined to convert a history into proper sentences of the situation calculus that capture the information it contains. The macro **end** gives the situation term corresponding to a history, while the macro **Sensed** produces a formula asserting that each action produced the given sensing result. Let ϵ be the empty history, then the definitions are:

$$\begin{aligned}
\mathbf{end}[\epsilon] &\stackrel{\text{def}}{=} S_0 \\
\mathbf{end}[h \cdot (a\#r)] &\stackrel{\text{def}}{=} do(a, \mathbf{end}[h]) \\
\mathbf{Sensed}[\epsilon] &\stackrel{\text{def}}{=} \top \\
\mathbf{Sensed}[h \cdot (a\#r)] &\stackrel{\text{def}}{=} \mathbf{Sensed}[h] \wedge SR(a, \mathbf{end}[h]) = r
\end{aligned}$$

Then, instead of asking whether a query holds at the current situation σ:

$$\mathcal{D} \models \phi[\sigma]$$

The agent asks whether the query holds given its current history h:

$$\mathcal{D} \models \mathbf{Sensed}[h] \rightarrow \phi[\mathbf{end}[h]]$$

This approach works well for a single agent, but we are aware of no works extending this meta-level handling of histories to the multi-agent case.

While the meta-level approach of [17] allows an agent to reason based on its local perspective, it is cumbersome for reasoning *about* that local perspective. To determine whether an agent *knows* that a formula holds in a given situation, we must explicitly specify the agent's history of sensing results for that situation, which may not be available until run-time. This meta-level approach is not suitable for rich epistemic reasoning, where we may want to reason offline about what the agent (or a group of agents) will or will not know after a series actions.

This kind of reasoning requires an explicit representation of an agent's knowledge, as described in Section 2.3. We will review such epistemic reasoning in more detail in Chapter 7, where we extend existing approaches to handle asynchronous domains based on the formalism developed in this chapter. For now we briefly discuss its use in axiomatising the local perspective of each agent.

The effect of sensing results on an agent's knowledge is axiomatised in [98] by directly specifying it in the successor state axiom for the knowledge fluent K. Agents discard situations that do not agree with the obtained sensing result:

$$K(agt, do(a, s'), do(a, s)) \equiv K(agt, s', s) \wedge Legal(a, s') \wedge SR(a, s') = SR(a, s)$$

As a variety of richer domains have been modelled on top of this formalism, their particular accounts of the local information available to each agent have been specified by progressively modifying this successor state axiom.

For example, when multiple agents are introduced, the successor state axiom for K is modified to ensure that an agent knows the results produced by its own actions, but not by the actions of others [106]. When communication actions are introduced, the successor state axiom for K is modified to ensure that the agent's knowledge is updated to include the communicated information [107, 108]. When concurrent actions and time are introduced, the successor state axiom for K is modified to ensure that the agent knows how much time has passed since the last action, while

not inadvertently learning the real value of the current time unless this was already known [97].

In these works there is no explicit representation of the local perspective of each agent – rather, the information each agent receives from an action is specified only in terms of its effect on the agent's knowledge. The formalism developed in this chapter will allow us to decouple the dynamics of knowledge from the specific details of how each action affects the agent's local perspective. As we will show in Chapter 7, this can produce a much more general and robust formalism for knowledge.

The observation-based approach will also allow us to work directly with the local information available to each agent without needing to make explicit statements about knowledge. For example, when planning the cooperative execution of a task, we can formulate a reactive plan in which each agent can act based directly on its local observations, without having to introspectively reason about what it knows.

A related approach to ours is the work by Pinto [81] on axiomatising narratives in the situation calculus. Here the term "observation" is used in a more general sense to mean some partial information about the state of the world, such as an action occurring or a fluent holding at a particular time. These are asserted using predicates such as $occurs(a, t)$ and $holds(F, t)$, and situations are defined as *proper* if they respect the asserted *occurs* and *holds* facts. Although the focus of [81] is on reasoning from a single omniscient perspective, it could easily be extended to reason about the local perspective of multiple agents.

The crucial difference between [81] and the approach presented in this chapter is that we provide an explicit axiomatisation not just of *observations* but of *observability.* We provide a complete account of what each agent would observe if any particular action occurred in any particular state of the world. By virtue of not having made particular observations, agents in our formalism can conclude that certain actions cannot have occurred. By contrast, the use of *occurs* and *holds* in [81] specifies only what *must* have happened, not what *cannot* have happened. This distinction will play an important role for effective reasoning for our formalism.

4.2 Definitions

In this section we formally define an explicit representation of the local information available to each agent, and do so in a manner that is independent of how that information will eventually be used. Our approach can be seen as a generalisation of the history-based approach of [17], explicitly representing the information available to each agent. However, we encode this information as terms in the logic rather

than in the meta-level reasoning machinery. This allows us to use this explicit local perspective in more general ways.

We begin by defining the notion of an *observation*, which is fundamental to the entire approach. At the simplest level, this is an internal notification that an agent receives when some action has occurred.

Definition 8 (Observations)**.** *An observation is a notification event received by an agent, making it aware of some change in the state of the world. When an agent receives such a notification, we say that it "observed", "made" or "perceived" that observation.*

Since "observation" is quite a loaded term, it is worth re-iterating this point: our observations are instantaneous *events* generated internally by each agent in response to actions occurring in the world. We make no commitment as to how these events are generated, preferring a clean delineation between the task of observing change and the dynamics of knowledge update based on those observations.

Since the state of the world may only change in response to some action, observations can only be made as a result of some action. For simplicity we assume that agents perceive observations instantaneously, i.e. in the same instant as the actions that cause them; see Section 4.4.4 for a suggestion on how delayed observations can be modelled within this framework.

To make this idea concrete, let us introduce an additional sort OBSERVATION to the language $\mathcal{L}_{sitcalc}$, for the moment without any particular commitment towards what this sort will contain. We then add an action description function of the following form to $\mathcal{D}_{ad}$:

$$Obs(agt, c, s) = o$$

This function returns a set of observations, and should be read as "when actions c are performed in situation s, agent agt will perceive observations o". Using a set of observations allows an agent to perceive any number of observations in response to an action occurrence – perhaps several observations, perhaps none. When $Obs(agt, c, s)$ is empty, the agent makes no observations and the actions c are completely hidden.

The concept of a *view* follows naturally - it is the sequence of all the observations made by an agent as the world has evolved.

Definition 9 (Views)**.** *An agent's view in a given situation* s *is the corresponding sequence of observations made by the agent as a result of each action in* s*, excluding those actions for which no observations were made.*

We introduce another sort VIEW consisting of sequences of sets of observations, with ϵ being the empty sequence and the functional fluent $View$ giving the history

of observations associated with a particular situation. Since these definitions will not change from one domain to another, they are added to the foundational axioms:

$$\begin{aligned} Init(s) &\rightarrow View(agt, s) = \epsilon \\ Obs(agt, c, s) = \{\} &\rightarrow View(agt, do(c, s)) = View(agt, s) \\ Obs(agt, c, s) \neq \{\} &\rightarrow View(agt, do(c, s)) = Obs(agt, c, s) \cdot View(agt, s) \end{aligned} \tag{4.1}$$

Observations and views can be seen as localised analogues of actions and situations respectively. An action is a global event that causes the state of the world to change, while an observation is an internal event that causes an agent's knowledge of the state of the world to change. Similarly, a situation represents a complete, global history of all the actions that have occurred in the world, while a view is an agent's local history of all the observations it has made. The situation is an omniscient perspective on the world, the view a local perspective. This distinction will be fundamental to the new techniques we develop throughout this thesis.

To provide a global account of the results returned by each action, we can define a *history* in a similar way to IndiGolog, but represented explicitly as a term in the language. First, we define the *outcome* of an action as a mapping from agents to the observations they made. To represent this mapping we use a set of AGENT#OBSERVATION pairs:

Definition 10 (Outcomes)**.** *The outcome of an action c is the set of $agt\#Obs(agt, c)$ pairs generated by that action:*

$$Out(c, s) = y \equiv \forall agt, o : agt\#o \in y \equiv Obs(agt, c, s) = o$$

Then we can build the global history of a situation as a sequence of actions paired with their respective outcomes:

Definition 11 (Histories)**.** *The history of a situation s is the sequence of action#outcome pairs corresponding to each action in s:*

$$\begin{aligned} Init(s) &\rightarrow History(s) = \epsilon \\ History(do(c, s)) = h &\equiv h = (c\#Out(c, s)) \cdot History(s) \end{aligned}$$

We can also introduce an analogous function Sit that translates from a history term back into a raw situation:

$$Sit(\epsilon) = S_0$$
$$Sit((c\#y) \cdot h) = do(c, Sit(h))$$

Histories will be useful for planning the cooperative execution of a shared task, when agents must explicitly reason about both the global state of the system and the local perspective of each agent. To this end, we introduce some suggestive notation for accessing the individual observations in an outcome:

$$Out(c, s)[agt] = o \equiv agt\#o \in Out(c, s)$$

To ensure that these definitions operate in an intuitively correct way, we also need a simple consistency constraint. Just as the empty set of actions is assumed to never be legal, so we should assume that it generates no observations – clearly the agents cannot observe anything if no action has taken place. Formally, we impose the following consistency requirement on basic actions theories containing *Obs*:

Definition 12 (Observation Causality Requirement)**.** *A basic action theory $\mathcal{D}$ specifying the Obs function must ensure that agents do not perceive observations that are not caused by some action:*

$$\mathcal{D} \models \forall agt, s : Obs(agt, \{\}, s) = \{\}$$

The key distinguishing feature of our approach is that the agent's view excludes cases where $Obs(agt, c, s)$ is empty, so the agent may not have enough information to determine how many actions have been performed in the world. As discussed in Chapter 1, this property is fundamental to modelling truly asynchronous domains. Mirroring the terminology of [118], we can explicitly define what it means for a domain to be *synchronous* in the situation calculus.

Definition 13 (Synchronous Action Theory)**.** *A basic action theory $\mathcal{D}$ is synchronous if every agent observes something whenever an action occurs:*

$$\mathcal{D} \models \forall agt, c, s : Legal(c, s) \rightarrow Obs(agt, c, s) \neq \{\}$$

As we shall see, such domains make reasoning from the local perspective of an agent significantly easier, as they do not need to consider arbitrarily-long sequences of hidden actions. Before proceeding with some example definitions of *Obs*, let us briefly foreshadow how observations and views will be used in the coming chapters.

In Chapter 5, we will define a partially-ordered branching action structure to be generated as the output of the MIndiGolog execution planning process. This structure, called a *joint execution,* represents many possible situations that are legal executions of the program. Since agents can only be expected to act based on their local information, we will require that if s and s' are two situations that could be reached while performing a joint execution, and $View(agt, s) = View(agt, s')$, then the agent's next action in both situations must be the same. Moreover, if the joint execution requires an agent to execute some action a_2 after another action a_1, we will require that $Obs(agt, a_1, s)$ is not empty, so that it will have some local observation to trigger the performance of a_2. These restrictions ensure that the joint execution can feasibly be carried out by the agents.

In Chapter 7, we formalise the intuition that an agent's knowledge should be based only on its local information. So if the agent believes that the world might be in situation s, then it must also consider possible any other situation s' such that $View(agt, s) = View(agt, s')$. By decoupling the axiomatisation of knowledge from the specific details of how each action affects the agent's local information, we develop a very general and robust formalism that can be applied without modification in a wide variety of domains.

4.3 Axiomatising Observations

We now show how observations and views can be used to model a variety of common domain dynamics from the situation calculus literature. We argue that these axiomatisations intuitively capture the "correct" information in each case, but defer a formal comparison between our approach and existing axiomatisations until we have developed our theory of knowledge in Chapter 7.

4.3.1 Public Actions

By far the most common assumption about the observability of actions is that "all actions are public", which can be rephrased as "when an action occurs, all agents will observe that action". Letting the OBSERVATION sort contain ACTION terms, this can be captured using the following axiom in the definition of *Obs*:

$$a \in Obs(agt, c, s) \equiv a \in c \tag{4.2}$$

When sensing actions are included, it is typically assumed that only the performing agent has access to the sensing results. This can be modelled by extending the OBSERVATION sort to contain ACTION#RESULT pairs, and including the following

in the definition for *Obs*:

$$a\#r \in Obs(agt, c, s) \equiv a \in c \wedge actor(a) = agt \wedge SR(a, s) = r \tag{4.3}$$

Note that since *Obs* is an action description function, technically we must specify it using a single axiom as described in Section 2.1.2. For the sake of clarity we specify these two cases independently, and assume that the final axiom defining *Obs* takes the completion of the individual cases in the standard way.

It should be clear that these definitions capture the intuition behind this most common model of action observability. When we develop our new axiomatisation of knowledge in Chapter 7, we will demonstrate that these definitions provide an equivalent account to the standard knowledge axioms of Scherl and Levesque [98].

This approach clearly leads to synchronous domains, since an agent's set of observations can only be empty if the set of actions is also empty, and the empty action set is never legal to perform.

4.3.2 Private Actions

Another common model for action observability is to assume that "all actions are private", which can be rephrased as "when an action occurs, only the performing agent will observe it". This can be modelled by simply dropping the public-observability axiom from equation 4.2, leaving the following definition of *Obs*:

$$a\#r \in Obs(agt, c, s) \equiv a \in c \wedge actor(a) = agt \wedge SR(a, s) = r$$

As noted in [54], this approach means that agents need to consider arbitrarily-long sequences of hidden actions which may or may not have occurred, and thus forego regression as an effective reasoning technique. By explicitly formalising this situation, we will be in a position provide the first formal account of effective reasoning in such asynchronous domains.

4.3.3 Guarded Sensing Actions

While the standard approach to sensing actions has the result $SR(a, s)$ returned unconditionally, it is also possible to model actions that return sensing information only when some additional conditions hold in the environment [77]. These can be modelled in our framework by adding the guard conditions Ψ to the definition of *Obs*:

$$a\#r \in Obs(agt, c, s) \equiv a \in c \wedge actor(a) = agt \wedge SR(a, s) = r \wedge \Psi(a, s)$$

For example, an action $sense_{\phi,\psi}$ that senses the truth of some formula ϕ when the guard condition ψ is true would require the following to be entailed by the definition:

$$sense_{\phi,\psi}\#T \in Obs(agt, c, s) \equiv sense_{\phi,\psi} \in c \wedge \phi(s) \wedge \psi(s)$$
$$sense_{\phi,\psi}\#F \in Obs(agt, c, s) \equiv sense_{\phi,\psi} \in c \wedge \neg\phi(s) \wedge \psi(s)$$
$$sense_{\phi,\psi} \in Obs(agt, c, s) \equiv sense_{\phi,\psi} \in c \wedge \neg\psi(s)$$

As noted in [77], guarded sensing actions can create difficulties when axiomatising the K fluent. The approach we develop in Chapter 7 will show that by explicitly representing the information returned by the action, rather than defining it implicitly in the axiom for K, these difficulties are avoided.

4.3.4 Speech Acts

Communication in the situation calculus is traditionally modelled using explicit communicative actions or "speech acts" [107, 108]. These actions are axiomatised as per standard actions, but special-case handling is introduced in the axioms for knowledge in order to model their communicative effects.

Instantaneous communication is modelled using actions such as $inform$, where $inform(agt_s, agt_r, \phi)$ means the sender agt_s informs the receiver agt_r of the truth of some formula ϕ. If we assert that only truthful speech acts are allowed, and all actions are publicly observable, then this requires no further axiomatisation:

$$Poss(inform(agt_s, agt_r, \phi), s) \equiv \phi[s]$$

The $inform$ action will be included in each agent's observations whenever it occurs, from which the agent can conclude that it was possible and thus that the contained formula holds in the world.

However, this simple approach can lead to third-party agents being aware of what was communicated, which is often not desirable. In [108] encrypted speech acts are introduced to overcome this limitation, ensuring that only the intended recipient of a message is able to access its contents by performing a special *decrypt* action. While it would be straightforward to copy this approach in our formalism, the problem it was introduced to solve no longer exists; we can directly limit the accessibility of

the message contents to the receiving agent without introducing another action:

$$inform(s, r) \in Obs(agt, c, s) \equiv \exists m : inform(s, r, m) \in c$$
$$inform(s, r, m) \in Obs(agt, c, s) \equiv inform(s, r, m) \in c \wedge (agt = r \vee agt = s)$$

Here all agents will observe that the communication occurred, but only the sender and recipient can access the contents of the message.

Non-instantaneous communication can be modelled using a message queue for each agent with separate *send* and *check* actions [54]. The *send* action adds a message to the queue, while the *check* action returns the details of pending messages as its sensing result. Since this approach uses the standard sensing-result machinery, it requires no special axiomatisation in our framework.

4.4 New Axiomatisations

From the above examples, it should be clear that our formalism can capture the information available to each agent under a variety of domain dynamics already modelled in the situation calculus. We now demonstrate some new axiomatisations of domains that have not previously been explored in the situation calculus.

4.4.1 Explicit Observability Axioms

Our approach offers a straightforward way to explore the middle ground between the two extremes of "public actions" and "private actions" discussed in the previous section. To axiomatise general *partial observability* of actions, we introduce a new action description predicate $CanObs(agt, a, s)$ that defines the conditions under which agent agt would observe action a being performed in situation s. If $CanObs(agt, a, s)$ is false, then that action will be hidden. We can then define Obs as follows:

$$a \in Obs(agt, c, s) \equiv a \in c \wedge CanObs(agt, a, s)$$

This permits a great deal of flexibility in the axiomatisation. Consider a domain in which the agents inhabit several different rooms, and are aware of all the actions performed in the same room as themselves:

$$CanObs(agt, a, s) \equiv InSameRoom(agt, actor(a), s)$$

It is also possible to allow partial observability of sensing results using an analo-

gous predicate $CanSense(agt, a, s)$ and the following definition of Obs:

$$a \in Obs(agt, c, s) \equiv a \in c \wedge CanObs(agt, a, s) \wedge \neg CanSense(agt, a, s)$$
$$a\#r \in Obs(agt, c, s) \equiv a \in c \wedge SR(a, s) = r \wedge CanObs(agt, a, s) \wedge CanSense(agt, a, s)$$

For example, consider an agent waiting for a train who activates a speaker to determine when it will arrive. The results of this sensing action would provide information to any other agent within earshot:

$$CanSense(agt, activateSpeaker(agt_2), s) \equiv CloseToSpeaker(agt)$$

We feel that this formulation provides a good balance between simplicity and expressiveness; it allows the observability of actions to vary according to the state of the world, but provides agents with a complete description of each action that they are capable of observing.

4.4.2 Observability Interaction

Reasoning about observability of concurrent actions raises the potential for *observability interaction*, in which some actions produce different observations when they are performed concurrently with another action. Like the precondition interaction problem for $Poss$ discussed in Section 2.1.4, we assume that the axiom defining Obs contains the appropriate logic to handle such interaction. A simple axiomatisation might have actions being "masked" by the co-occurrence of another action, and would appear like so:

$$a \in Obs(agt, c, s) \equiv a \in c \wedge CanObs(agt, a, s) \wedge \neg \exists a' \in c : Masks(a', a, s)$$

The important point is that, given an explicit account of the local perspective of each agent, such interaction can be axiomatised independently of the rest of the action theory.

4.4.3 Observing the Effects of Actions

In many domains it would be infeasible for an agent to observe all of the details of a particular action when it occurs, but it may observe some of the effects of that action. For example, suppose that an agent monitors the state of a light in its environment, such that it notices it changing from dark to light. While it knows

that *some* action must have occurred to produce that effect, it may not be sure precisely what action took place (e.g. precisely *who* turned on the light). This can be modelled by further extending the OBSERVATION sort to contain a special "effect observation" term $lightCameOn$, and axiomatising like so:

$$lightCameOn \in Obs(agt, c, s) \equiv \neg lightIsOn(s) \wedge \exists agt' : turnLightOn(agt') \in c$$

When the light is switched on, each agent's observation set will contain the term $lightCameOn$, and they will be able to deduce that this change has occurred without necessarily knowing the specific action responsible for the change. This is similar to the "fluent change" actions proposed by De Giacomo et al. [19], but embedded in the theory itself rather than generated by the agent when it discovers that it must update its beliefs.

4.4.4 Delayed Communication

Delayed communication can be modelled using separate *send* and *recv* actions. However, unlike the use of explicit communication channels discussed in the previous section, we do not want the receiving agent to have to poll the message queue. Rather, the *recv* action should occur automatically some time after the *send* action.

This is easily modelled by making *recv* a natural action. The *send*/*recv* pair can then be axiomatised mirroring the standard account of long-running tasks in the situation calculus. A fluent $PendMsg(s, r, m, t)$ indicates that some message is pending and will be delivered at time t. We have:

$$natural(recv(agt_s, agt_r, m))$$
$$send(agt_s, agt_r, m) \in Obs(agt, c\#t, s) \equiv send(agt_s, agt_r, m) \in c \wedge agt = agt_s$$
$$recv(agt_s, agt_r, m) \in Obs(agt, c\#t, s) \equiv recv(agt_s, agt_r, m) \in c \wedge agt = agt_r$$
$$Poss(recv(agt_s, agt_r, m)\#t, s) \equiv PendMsg(agt_s, agt_r, m, t, s)$$

$$\begin{aligned} PendMsg(s, r, m, t_m, do(c\#t, s)) \equiv\ & send(s, r, m) \in c \wedge t_m = t + delay(s, r, m, s) \\ & \vee PendMsg(s, s, m, t_m, s) \wedge (recv(s, r, m) \notin c \vee t \neq t_m) \end{aligned}$$

A *send* action thus causes the message to become pending, with its delivery time determined by the functional fluent *delay*. Once the delay time has elapsed, the natural action *recv* will be triggered and the message delivered. The *send* and *recv* messages are observed only by the sender and receiver respectively.

If the agents have incomplete information about the *delay* function, this can easily model domains in which the message delay is unpredictable or even unbounded, giving asynchronous communication in the style of [39].

4.5 Reasoning from Observations

With these definitions in place, we can now give a principled account of what it means for an agent to reason using its local information. Recall that in the single-agent setting of IndiGolog [17] a pair of macros is used to construct a query of the following form given the agent's current history h:

$$\mathcal{D} \models \mathbf{Sensed}[h] \rightarrow \phi[\mathbf{end}[h]]$$

This depends crucially on the assumption that all actions are publicly observable, so that the macro **end** can construct the precise situation term corresponding to a given history. The resulting query is in a form that can be answered effectively using the standard regression operator.

We can pose a similar query using the definition of a global history in our framework. First, define a history to be legal if it contains the correct sensing results for a legal situation:

$$Legal(h) \stackrel{\text{def}}{=} Legal(Sit(h)) \wedge History(Sit(h)) = h$$

Then an appropriate query using the current history h would be:

$$\mathcal{D} \models Legal(h) \rightarrow \phi[Sit(h)]$$

Since these are no longer macros, but are now actual functions in the logic, this query is not immediately amenable to standard regression techniques. However, since a history can always be converted into a unique corresponding situation term, we can easily provide special-purpose regression rules as follows:

$$\mathcal{R}(\phi[Sit(\epsilon)]) \stackrel{\text{def}}{=} \phi[S_0]$$
$$\mathcal{R}(\phi[Sit((c\#y) \cdot h)]) \stackrel{\text{def}}{=} \mathcal{R}(\phi, c)[Sit(h)]$$

These rules mirror the definition of **end** and preserve equivalence given the definition of the $History$ function. Since $Legal$ and $History$ are ordinary fluents they can be handled by standard regression rules. Agents can therefore use such a query to do regression-based reasoning about some hypothetical future state of the world,

for example for the purposes of planning.

An agent could not, however, answer queries about the actual current state of the world in this manner. They will not have access to the current history term, and must instead reason based only on their current view v. Since multiple situations can result in the same view, the appropriate query would be:

$$\mathcal{D} \models \forall s : View(agt, s) = v \rightarrow \phi[s]$$

Here the agent encounters a problem – this is a much more difficult query in general. Since it cannot tell how many actions have occurred based on its local view, it cannot re-write this query into a form suitable for regression. The agent must instead perform second-order theorem proving, using the induction axiom over situations, in order to reason based on its local view. The situation calculus currently offers no tools for effective reasoning about such queries.

However, suppose the domain is synchronous. Then combining Definition 13 with equation (4.1), we can prove that all situations matching a given view will contain the same number of actions. The agent can therefore construct a query like the following to reason about the the world using standard regression techniques:

$$\mathcal{D} \models \forall c_1, \ldots, c_n : View(agt, do([c_1, \ldots, c_n], S_0)) = v \rightarrow \phi[do([c_1, \ldots, c_n], S_0)]$$

Thus in synchronous domains, existing reasoning techniques of the situation calculus can be used by an agent to reason from its own local perspective in much the same way as in the single-agent case. In asynchronous domains, the induction axiom is required and no effective reasoning procedures currently exist. This restriction, more than any other, has limited the use of the situation calculus for modelling asynchronous multi-agent domains.

As we shall see in the coming chapters, for offline planning we can permit the agents to reason using the hypothetical global history rather than their local observations. For richer epistemic reasoning about the current state of the world, we will require a technique capable of performing inductive reasoning.

4.6 Discussion

In this chapter we have constructed an explicit representation of the local perspective of each agent, in terms of *observations* and *views*. This terminology has been deliberately chosen to mirror that used in other formalisms where representing this

local perspective is the norm, such as [39, 74]. As the examples in Sections 4.3 and 4.4 have demonstrated, this approach is able to capture a very wide variety of domain dynamics in a flexible way.

Some of our axioms in Sections 4.3 and 4.4 may seem rather ad-hoc, but we claim they are no more or less ad-hoc than the many adjustments made to the axioms defining the knowledge fluent K to accommodate different kinds of information-producing action [54, 77, 106–108]. The difference is that these adjustments can now be made separately from the rest of the theory, rather than in the fundamental axiom for reasoning about knowledge. This makes our formalism significantly more elaboration tolerant, a point we will return to in Chapter 7. It also means that for certain applications, we can reason about an agent's local view without the overhead of performing explicit epistemic reasoning.

Of course, we also pay a price for this extra expressive power: representational complexity. The theory of action must contain an explicit axiomatisation of the OBSERVATION sort and of our new *Obs* function. There is something of a tradition in the situation calculus of doing as much as possible at the meta-level, adding to the theory itself only when necessary [62]. As we will demonstrate in the remainder of this thesis, the advantages provided by our explicit representation of each agent's local perspective more than compensate for the added complexity it introduces to the theory of action.

From the perspective of the rest of the thesis, the key contribution of this chapter is to provide a uniform representation formalism. The domain-specific observability dynamics can now be specified independently from the rest of the theory. By "factoring out" the details in this way, we are in a position to construct formalisms and reasoning techniques that do not make any assumptions about action observability. In particular, we can explicitly represent and reason about asynchronous domains.

Chapter 5

Joint Executions

This chapter constructs a new representation for the actions to be performed by a team of agents during the cooperative execution of a shared task. Dubbed *joint executions*, they are partially-ordered branching sequences of events. Joint executions allow independent actions to be performed independently, while using each agent's local view to ensure that synchronisation is always possible when required.

The output of the standard Golog execution planning process is a raw situation term; a complete, ordered sequence of all actions that are to be performed. This is suboptimal for generating and representing plans in an asynchronous multi-agent setting in three ways:

- it does not permit branching to utilise information obtained at run-time
- it enforces a strict execution order on actions that are potentially independent, requiring inter-agent synchronisation when it is not actually necessary
- it requires a strict execution order on actions that may be unobservable, demanding inter-agent synchronisation that is not actually feasible

As we have demonstrated in Chapter 3, restricting the domain to be synchronous and completely known lets the agents make effective use of raw situation terms for planning. In asynchronous domains with incomplete knowledge they are no longer sufficient, and the Golog execution planner is required to generate a much richer representation of the actions to be performed.

To build such a representation, we take inspiration from a model of concurrent computation known as *prime event structures*, which are partially-ordered branching sequences of events [72]. A *joint execution* is defined as a particular kind of prime event structure that is rich enough to capture the concurrent execution of

independent actions and can branch on the results of sensing actions. We use our explicit account of an agent's local view to identify joint executions that can feasibly be executed based on the local information available to each agent at runtime.

Joint executions are formalised in a way that translates well into an implementation. They can be built up one action at a time in much the same way as ordinary situation terms. If the theory of action meets some simple restrictions, joint executions can also be reasoned about using standard regression techniques. We demonstrate an implementation that performs offline execution planning for an asynchronous, partially observable domain, and discuss the challenges faced when attempting a cooperative online execution in such domains.

Joint executions thus allow us to represent the actions that a team of agents are to perform in service of some shared task, without requiring constant synchronisation between the agents, and without assuming that agents know all the actions that have been performed, while utilising existing reasoning methods and planning machinery. This is a significant increase in power over existing approaches to planning for multi-agent teams in the situation calculus.

The chapter proceeds as follows: after some more detailed background information in Section 5.1, we formally define and axiomatise joint executions in Section 5.2. Section 5.3 then characterises the Golog execution planning problem in terms of joint executions rather than raw situation terms, and Section 5.4 identifies a restricted kind of joint execution that can be reasoned about effectively using standard regression techniques. In Section 5.5 we present an overview of our new MIndiGolog execution planner that generates joint executions, and show some examples of its output. Finally, Section 5.6 concludes with some general discussion and an outline of our ongoing work in this area.

5.1 Background

The above discussion highlights three important properties of a plan representation formalism intended for use in asynchronous multi-agent domains: it must be *partially-ordered* to allow agents to operate independently, *branching* to allow information to be collected at run-time, and *feasible to execute* based on the local information available to each agent. While each of these aspects have been studied in isolation in the situation calculus, our work is the first to combine them into a single formalism that is suitable for asynchronous multi-agent domains.

5.1.1 Partial Ordering

There has been little work on partial-order planning in the situation calculus, most likely because the use of situation terms heavily biases the reasoning machinery towards totally-ordered sequences of actions. While Baral and Son [6] allow the programmer to specify a partial order on actions by adding operators to the Golog language, the actual plans produced by their system are still ordinary situation terms. One exception is [86], which extends the situation calculus with explicit "aspects" and allows partial ordering between actions that affect different aspects of the world state. By contrast, we seek to leverage the existing meta-theory of the standard situation calculus.

Partial-order planning is the mainstay of the closely-related *event calculus* formalism [50]. In this formalism, actions are represented as occurring at specific times, rather than in a specific order as in the situation calculus. Constraints placed on the relative occurrence times of actions then determine a partial ordering. Shanahan [103] has shown that abductive theorem proving in the event calculate generates partially-ordered plans, and the mechanics of the theorem prover naturally mirror various concepts from the goal-based partial-order planning literature, such as conflicts, threats and links [75].

The close similarities between the situation and event calculi are well understood, as are the advantages of the event calculus when working with partially-ordered action sequences [10]. Indeed, it is possible to implement a Golog interpreter on top of the event calculus, and the execution plans it generates are partially-ordered sets of actions [24]. Perhaps we should simply adopt a formalism such as the event calculus that is naturally partially-ordered, rather than trying to construct a partially-ordered representation on top of the naturally sequential situation calculus?

Having a partial-order representation is important, but it is not the complete picture. While we don't want the agents to have to synchronise their actions unnecessarily, we also need to ensure the converse: that when an explicit ordering between actions is *necessary*, the required synchronisation is actually *feasible* based on the local information available to each agent. It is not clear how techniques such as [24] would extend to the asynchronous multi-agent case.

By taking advantage of our explicit account of the local information available to each agent, the formalism developed in this chapter enables this pair of dual requirements - that some actions don't need to be ordered, while other actions cannot be ordered - to be captured in an elegant way. Moreover, we do not need to step outside the bounds of existing situation calculus theory, and can utilise existing regression techniques for effective automated reasoning.

5.1.2 Branching

Several single-agent formalisms based on the situation calculus have introduced some form of branching into the structures returned by the planner, including the conditional action trees of of sGolog [52] and the branching IndiGolog plans of [95]. These structures typically branch based on the truth or falsehood of test conditions included in the program. For example, the structural definition of conditional action trees in [52] includes the following branching case:

$$c = [\phi, c_1, c_2]$$

This instructs the agent to execute the sub-tree c_1 if ϕ is true and the sub-tree c_2 if ϕ is false. An alternate approach, exemplified by the "robot programs" of Lin and Levesque [64], is to have the plan branch directly on the results returned by actions rather than on a test condition. Branching on the binary result of a sensing action is represented in this formalism by the following structure:

$$branch(action, \delta_1, \delta_2)$$

Here the agent continues execution with program δ_1 if the action returns true, and with δ_2 if the action returns false. Plans that branch directly on the results of actions are typically longer, but easier for the agent to execute reactively since it does not need to introspect its knowledge base to decide a test condition.

5.1.3 Feasibility

To allow an agent to execute a plan that depends on information collected at run-time, it is not sufficient to simply introduce branching into the plan representation formalism. One must also ensure that, at execution time, the agent will always *know* which branch of the plan to take. For example, suppose this simple branching plan will provably achieve a goal:

$$\textbf{if } \phi \textbf{ then } action_1 \textbf{ else } action_2$$

The agent can only execute this program if it knows whether or not ϕ holds; otherwise, although one of the branches is guaranteed to achieve the goal, the agent does not know which branch to take. Feasibility is typically guaranteed by including sensing actions to ensure that the test conditions become known when needed:

$$sense_\phi \text{ ; } \textbf{if } \phi \textbf{ then } action_1 \textbf{ else } action_2$$

This requirement that an agent "knows how" to execute a plan is formalised by various notions of *epistemic feasibility*, including those of [4, 55, 56, 64, 95].

One approach to ensuring feasibility, embodied by [4, 55, 95], is to represent plans by arbitrary programs formulated in a control language such as Golog. One then semantically characterises the class of epistemically feasible programs, using direct assertions about the knowledge of each agent at each stage of execution. While this allows for potentially very rich, very succinct plans, it is not clear how to systematically generate an epistemically feasible plan using such a general characterisation.

Another approach, advocated by [61, 64] and used in the implementation section of [95], is to restrict the structure of plans so that they are always epistemically feasible. For example, the "robot programs" of [64] are restricted to simple operators such as sequencing, branching and looping:

$$action$$
$$seq(\delta_1, \delta_2)$$
$$branch(action, \delta_1, \delta_2)$$
$$loop(branch(action, \delta, exit))$$

These programs do not contain test conditions, but rather branch and loop directly according to the sensing results returned from each action. There is therefore no potential for confusion when executing such programs; they are essentially equivalent to a kind of finite automaton that can be executed reactively. Nevertheless, Lin and Levesque [64] show that these programs are universal, in the sense that any achievable goal can be achieved by suitable a robot program. We are not aware of any work extending this approach to represent programs intended for cooperative execution by a team of agents.

These existing notions of epistemic feasibility can be broadly characterised as *knowing what*. At each stage of execution, each agent must know what its next action is. In synchronous domains with public actions, as typically studied in the situation calculus, this is sufficient to ensure the feasibility of executing a plan.

In asynchronous domains it is not enough for an agent to know *what* its next action is; it must also know *when* that action should be performed. For example, suppose that the following simple plan provably achieves a goal:

$$action_1(agt_1)\,;\, action_2(agt_2)$$

In a synchronous domain this plan can be executed directly. But suppose the domain is asynchronous, and agt_2 is unable to observe the occurrence of $action_1$.

Since agt_2 has no way of knowing whether or not $action_1$ has been performed yet, it will not know when to perform $action_2$ and the plan cannot be executed.

In this chapter we ensure plan feasibility by restricting the structure used to represent plans, in an approach similar to [64] but without looping constructs. We use the explicit account of an agent's local view developed in the previous chapter to ensure that each agent will always have enough information to determine what action to perform next, and when to perform it.

5.1.4 Event Structures

To tackle cooperative execution in a multi-agent setting, we have adopted a model of concurrent computation known as *event structures* [72]. The particular variant we are interested in are *prime event structures*, which are defined as follows.

Definition 14 (Prime Event Structure)**.** *A prime event structure is a four-tuple* $(\mathcal{V}, \gamma, \prec, \oplus)$ *where:* $\mathcal{V}$ *is a set of events;* γ *is a function assigning a label to each event;* $\prec$ *is the precedence relation, a strict partial order on events;* $\oplus$ *is the conflict relation, a binary symmetric relation indicating events that are mutually exclusive.*

The labels assigned by γ give the action associated with each event. By using a labelling scheme rather than identifying events directly with actions, multiple events can result in the same action being performed. The precedence relation restricts the order in which events can occur, so that if $e1 \prec e2$ then $e1$ must occur before $e2$. The conflict relation allows the structure to represent branching, by having the occurrence of some events preclude the occurrence of others.

Figure 5.1 shows a simple example of a prime event structure. The arrows represent the precedence relation, so in this diagram we have $e1 \prec e3 \prec e7$, but $e3 \not\prec e4$. The conflict relation is represented using a dotted line, so we have $e2 \oplus e3$ and only one of these two events is permitted to occur. Conflict is also inherited through the precedence relation, so $e6 \oplus e7$ in this diagram.

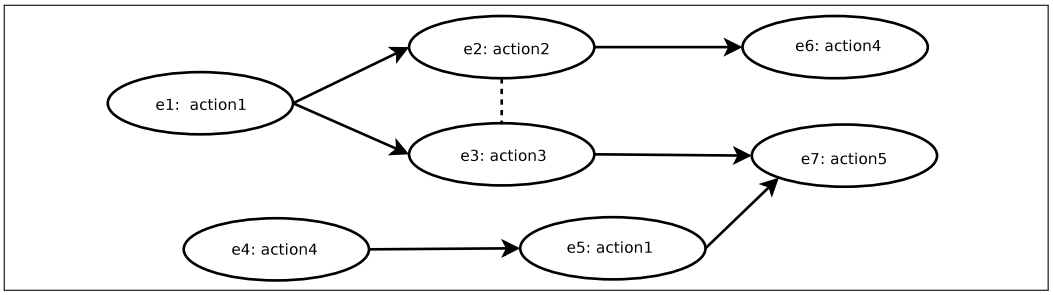

Figure 5.1: An example Prime Event Structure.

As it can be cumbersome to specify $\prec$ and $\oplus$ in their entirety, we will instead specify only the direct *enablers* and *alternatives* for each event, denoted by $ens(i)$ and $alts(i)$ respectively. Construction of $(\prec, \oplus)$ from $(ens, alts)$ is a straightforward transitive closure. Indeed, it only the enablers and alternatives that are represented explicitly in Figure 5.1, by arrows and dotted lines respectively.

A *configuration* is a sequence of events consistent with $\prec$ in which no pair of events conflict. Each configuration represents a potential partial run of execution of the system. Event structures thus form a directed acyclic graph of the events that could occur during execution of the system. As shown in [88], these structures are a canonical representation of a variety of formalisms for representing concurrent execution, and it is straightforward to execute them in a purely reactive fashion.

5.2 Joint Executions

This section defines *joint execution*s as a restricted kind of prime event structure suitable for representing the actions of a team of agents in an asynchronous domain. We begin with a high-level intuitive description to motivate these structures, and then formally define them using a set of axioms to be included in the theory of action $\mathcal{D}$. Since we intend for agents to synthesise joint executions as the output of a planning process, they must exist as concrete terms in the logic.

5.2.1 Motivation

To make things more concrete, consider again the "cooking agents" example domain from Chapter 3 and the $MakeSalad$ program shown in Figure 3.1. In a completely-known, synchronous domain, the execution found for this program by our MIndiGolog interpreter was a linear sequence of concurrent actions as shown in Figure 3.4 on page 57.

Let us now suppose that the cooking agents domain is asynchronous, and all actions other than *release* and *acquire* are private. The execution found by a MIndiGolog interpreter for such a domain cannot assume that the agents perform their actions in lock-step. Rather, it should allow the agents to process their respective ingredients independently, synchronising their actions only on the *release*/*acquire* sequence necessary to gain control of shared utensils.

An appropriate partially-ordered representation of the actions to be performed for $MakeSalad$ would then look something like the structure shown in Figure 5.2. For simplicity, we do not consider time or natural actions in this chapter, and have collapsed the "mix" and "chop" tasks into primitive actions. Without expanding on

the details at this stage, it should be clear that this structure captures the same basic workflow as the synchronous execution of *MakeSalad* from Chapter 3, but without imposing a strict ordering between the independent actions of different agents.

Indeed, Figure 5.2 is the joint execution produced for the *MakeSalad* program by our new execution planner detailed in Section 5.5, although with certain details suppressed for brevity. It may be helpful to keep this structure in mind as we develop the formal definitions contained in this section.

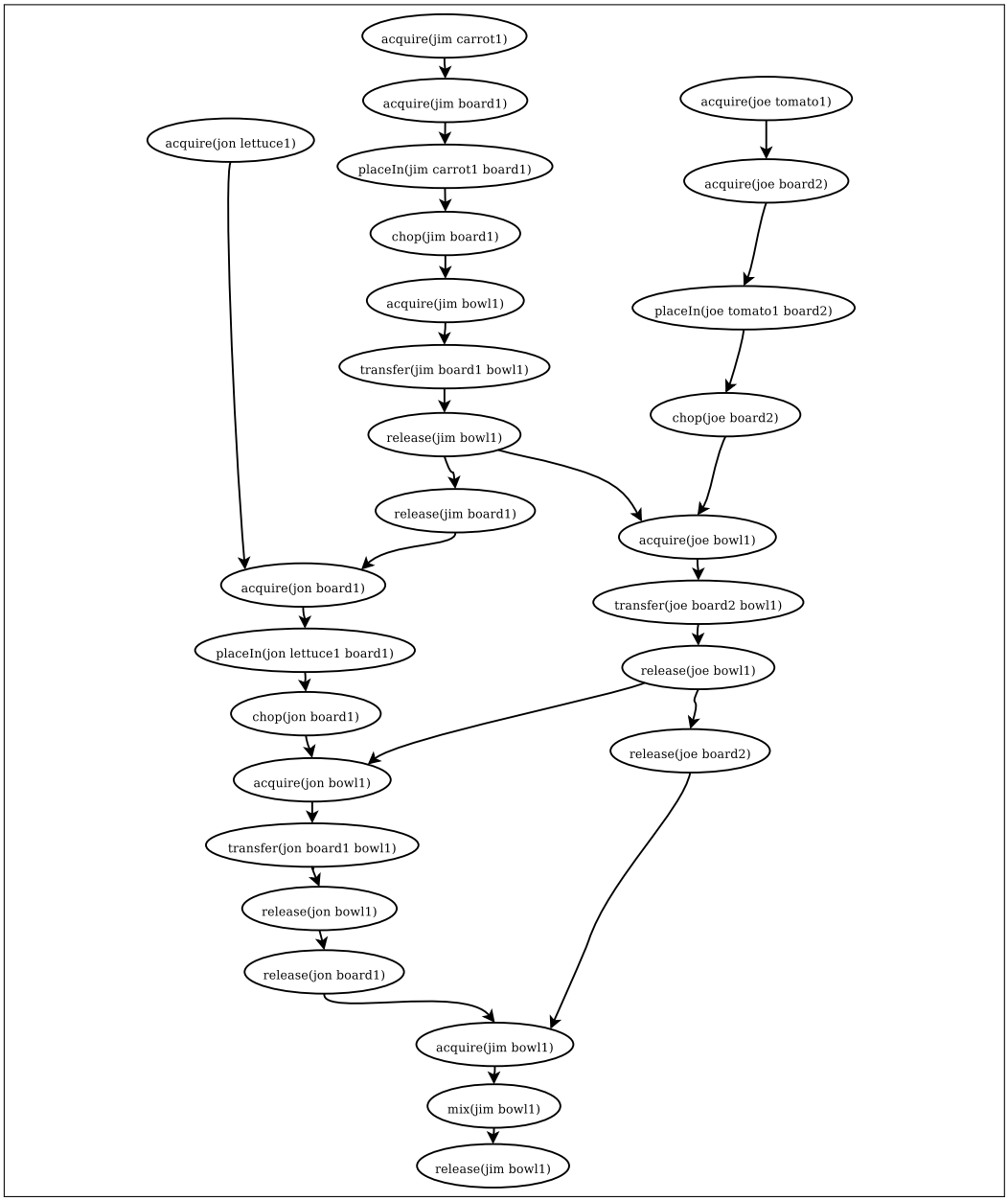

Figure 5.2: Joint Execution for the *MakeSalad* Program

5.2.2 Intuitions

We define a joint execution as a special kind of prime event structure as follows:

Definition 15 (Joint Execution)**.** *A joint execution is a tuple* $(\mathcal{A}, \mathcal{O}, ens, alts, \gamma, <)$ *where: action events* $\mathcal{A}$ *represent actions to be performed; outcome events* $\mathcal{O}$ *represent possible outcomes of actions;* $(\mathcal{A} \cup \mathcal{O}, ens, alts, \gamma)$ *forms a prime event structure with precedence relation* $\prec$*;* $<$ *is a total order on events that is consistent with* $\prec$*.*

A joint execution contains two disjoint sets of events: *action* events $\mathcal{A}$ representing the actions to be performed, and *outcome* events $\mathcal{O}$ representing the possible outcomes of each action. For each action event $i \in \mathcal{A}$, its enablers $ens(i)$ is a set of outcome events, its alternatives $alts(i)$ is empty, and its label $\gamma(i)$ is the action to be performed. For each outcome event $i \in \mathcal{O}$, $ens(i)$ is a single action event for which it is a possible outcome, $alts(i)$ is the set of all other outcome events j such that $ens(j) = ens(i)$, and $\gamma(i)$ is an outcome as produced by the $Out(a, s)$ function for the action $\gamma(ens(i))$.

Each action event thus represents a single action to be performed, which enables several alternative outcome events corresponding to the potential results returned by that action; since the action can only produce one actual outcome when it is executed, the enabled outcome events are all mutually conflicting. Each of these outcome events can then enable further action events, and so forth.

A simple example of a joint execution is shown in Figure 5.3, again using the "cooking agents" example domain. Here elliptical nodes are action events and box nodes are the resulting outcome events. The action *checkFor* senses the presence of a type of ingredient, returning either T or F, and thus producing two conflicting outcome events. In this example the agent Jim senses for the availability of eggs, and if this returns true he acquires one; otherwise, he acquires a tomato. Meanwhile agent Joe acquires a lettuce, independent of the actions Jim is performing.

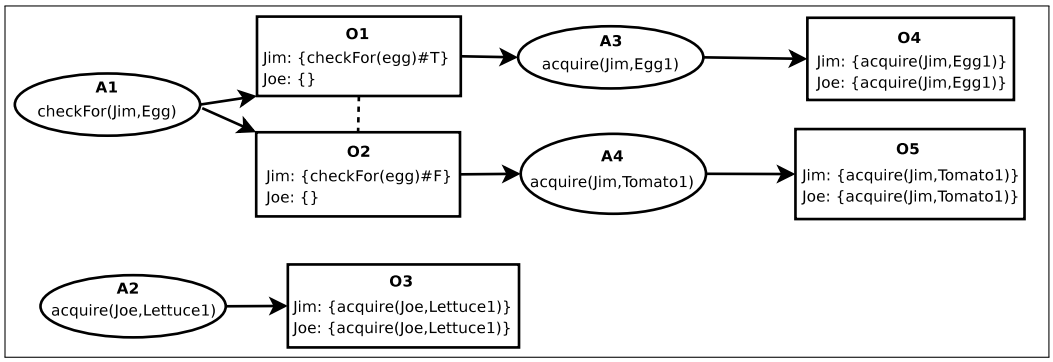

Figure 5.3: A simple joint execution.

Since we are explicitly considering concurrent actions, there are many different possible ways that the events in this structure could translate into action occurrences. The independent actions $checkFor(Jim, Egg)$ and $acquire(Joe, Lettuce1)$ could be performed in either order, or even concurrently.

These structures are clearly much richer than ordinary situation terms, as they permit branching and partial-ordering between actions. Still, they correspond to sets of ordinary situation terms in a straightforward way. Recall that a *configuration* is a partial run of execution of a prime event structure. Clearly any configuration ending in an outcome event corresponds to a unique situation term and also a unique history term, as it is a sequence of alternating actions and their outcomes.

We will call a set of unordered, non-conflicting outcome events a *branch*. A branch identifies a set of partial runs of the joint execution. In Figure 5.3, the sets $\{O3\}$, $\{O1, O3\}$ and $\{O5, O3\}$ are examples of branches. A *leaf* is a special case of a branch, where every event is either in the leaf, conflicts with something in the leaf, or precedes something in the leaf; it thus represents potential *terminating* runs of the joint execution execution. In Figure 5.3 there are two leaves, $\{O3, O4\}$ and $\{O3, O5\}$, generated by the two alternate outcomes of the $checkFor$ action.

A *history* of a branch is a history term (as defined in Chapter 4) that can be generated by performing actions and observing outcomes from the joint execution until all events in the branch have occurred. By these definitions, the set of histories of all leaves gives every possible history that could be produced by performing the joint execution through to a terminating configuration.

A joint execution has one additional component over a standard prime event structure: a *total* order on events $<$ that is consistent with the partial order $\prec$ induced by the enabling relation. We call this the *canonical ordering*, and it allows any branch to be unambiguously translated into a single *canonical history*. When we come to use joint executions for planning, we will use the canonical history to avoid having to reason about all the (potentially exponentially-many) histories of each leaf. The canonical ordering is essentially arbitrary; in practice it is determined by the order of insertion of events into the structure.

5.2.3 Structural Axioms

We introduce new sorts EVENT and JOINTEXEC to $\mathcal{L}_{sitcalc}$, and will collect the axioms defining joint executions in a separate axiom set $\mathcal{D}_{je}$. Events are opaque identifiers with which a joint execution associates a label, a set of enablers, and a set of alternatives. In practice we identify events with the integers, although our definitions require only a total ordering relation over events. Labels are either

ACTION or OUTCOME terms. A joint execution is then a term containing:

- a set of *events*, which are opaque ids having total order $<$
- a mapping from each event to a *label*, which is either an action or an outcome
- a mapping from each event to its *enablers*, a set of lower-numbered events
- a mapping from each event to its *alternatives*, a set of events

We will use the function $jexec$ as a constructor for joint execution terms, specifying each of the four features above as an argument, and using sets of $key\#value$ pairs to represent a mapping as in the previous chapter.

First, we require a unique names axiom to specify that a joint execution is uniquely defined by its four components, and a domain closure axiom to specify that all joint executions are constructed in this way. Assuming the variables are restricted to appropriate sorts by $\mathcal{L}_{sitcalc}$, the following axioms suffice:

$$\forall ex : \exists es, ls, ns, as : \ ex = jexec(es, ls, ns, as)$$

$$jexec(es, ls, ns, as) = jexec(es', ls', ns', as') \equiv$$
$$es = es' \wedge ls = ls' \wedge ns = ns' \wedge as = as'$$

We introduce four functions to access the components of a joint execution:

$$events(ex) = es \equiv \exists ls, ns, as : \ ex = jexec(es, ls, ns, as)$$
$$lblmap(ex) = ls \equiv \exists es, ns, as : \ ex = jexec(es, ls, ns, as)$$
$$ensmap(ex) = ns \equiv \exists es, ls, as : \ ex = jexec(es, ls, ns, as)$$
$$altsmap(ex) = as \equiv \exists es, ls, ns : \ ex = jexec(es, ls, ns, as)$$

We also define the following shortcut accessors to get the value from each mapping for a particular event i:

$$lbl(ex, i, l) \equiv i\#l \in lblmap(ex)$$
$$ens(ex, i, ns) \equiv i\#ns \in ensmap(ex)$$
$$alts(ex, i, as) \equiv i\#as \in altsmap(ex)$$

For notational convenience we will often write these as functions, e.g. $ens(ex, i) = ns$ rather than $ens(ex, i, ns)$, but this should be understood as an abbreviation since not every joint execution will contain every event.

We must also define the *precedes* and *conflicts* relations in terms of enablers and alternatives. These will be written as binary infix operators $\prec_{ex}$ and $\oplus_{ex}$ respectively. Since they are transitive closures they require a second-order axiomatisation. First, the precedence relation is defined as a simple transitive closure over enablers:

$$\forall P, ex, i, j : [(i \in ens(ex, j) \rightarrow P(i, j)) \wedge (\forall k : P(i, k) \wedge k \in ens(ex, j) \rightarrow P(i, j))] \\ \rightarrow (P(i, j) \rightarrow i \prec_{ex} j)$$

Then we can define the conflict relation by specifying that $i \oplus_{ex} j$ if they are alternatives to each other, or they have conflicting predecessors:

$$\forall P, ex, i, j : [(i \in alts(ex, j) \rightarrow P(i, j)) \\ \wedge \left(\forall i', j' : P(i', j') \wedge i' \preceq_{ex} i \wedge j' \preceq_{ex} j \rightarrow P(i, j)\right)] \\ \rightarrow (P(i, j) \rightarrow i \oplus_{ex} j)$$

Next we need axioms defining our terminology of *branches* and *leaves*. A branch is a set of unordered non-conflicting outcome events:

$$Branch(ex, br) \equiv \forall i, j \in br : IsOutcome(lbl(ex, i)) \wedge IsOutcome(lbl(ex, j)) \\ \wedge \neg(i \oplus_{ex} j) \wedge i \not\prec_{ex} j \wedge j \not\prec_{ex} j$$

A *leaf* is defined as a special case of a branch, so that every event in the joint execution is either in the leaf, precedes something in the leaf, or conflicts with something in the leaf:

$$Leaf(ex, lf) \equiv Branch(ex, lf) \\ \wedge \forall i \in events(ex) : i \in lf \equiv \neg(\exists i' \in lf : i \oplus_{ex} i' \vee i \prec_{ex} i')$$

Finally, we say a joint execution is *proper* if it respects the basic structural intuitions we discussed in the previous section. Every event must be proper according to its type, and events cannot be enabled by higher-numbered events:

$$Proper(ex) \equiv \forall i \in events(ex) : ProperAct(ex, i) \vee ProperOut(ex, i) \\ \wedge \forall i, j : (i \in events(ex) \wedge j \in ens(ex, i) \rightarrow j < i)$$

Note that this does not result in a loss of expressivity, since we we want event i to precede event j, then j cannot also precede i and we simply give j the higher event number. This restriction will play an important role in Section 5.4.

An action event is proper if it has no alternatives, enables at least one outcome event, and is enabled by a branch. Restricting the enablers to be a branch ensures that they do not contain any redundant or conflicting information.

$$ProperAct(ex,i) \equiv IsAction(lbl(ex,i))$$
$$\wedge Branch(ex, ens(ex,i)) \wedge alts(ex,i) = \{\} \wedge \exists j : ens(ex,j) = \{i\}$$

An outcome event is proper if it is enabled by a unique action event, and has as its alternatives the set of all other events enabled by that action.

$$ProperOut(ex,i) \equiv IsOutcome(lbl(ex,i))$$
$$\wedge \exists j : ens(ex,i) = \{j\} \wedge IsAction(lbl(ex,j))$$
$$\wedge \forall k : (k \in alts(ex,i) \equiv ens(ex,k) = \{j\} \wedge k \neq i)$$

These definitions enforce the basic structure of a joint execution according to the intuitions discussed in the previous section, but do not constrain it to be something that could actually be performed in the world – for example, outcomes can be enabled by actions that will never actually produce that outcome. Like situation terms, we focus first on getting the appropriate structure, and then specify additional conditions that joint executions must satisfy in order to be legal in the real world.

5.2.4 Performing Events

We introduce a predicate $Perform$ that axiomatises how events from a joint execution can be performed. Since we explicitly consider concurrent actions, this predicate selects a *set* of action events to be performed:

$$Perform(ex, es_a, es_o, ex') \equiv \quad es_a \neq \{\} \wedge es_o \neq \{\}$$
$$\wedge \forall i : (i \in es_a \rightarrow IsAction(lbl(ex,i)) \wedge ens(ex,i) = \{\})$$
$$\wedge \forall i : (i \in es_o \rightarrow \exists j : ens(ex,i) = \{j\} \wedge j \in es_a)$$
$$\wedge \forall i : (i \in es_a \rightarrow \exists j : j \in es_o \wedge ens(ex,j) = \{i\})$$
$$\wedge \forall i,j : (i \in es_o \wedge j \in es_o \rightarrow \neg(i \oplus_{ex} j))$$
$$\wedge \forall i : \big(i \in events(ex') \equiv i \notin es_a \wedge i \notin es_o \wedge \neg\exists j : (j \in es_o \wedge i \oplus_{je} j) \wedge\big)$$
$$\wedge \forall i, lb : \big(i\#lb \in lblmap(ex') \equiv lbl(ex,i) = lb \wedge i \in events(ex')\big)$$
$$\wedge \forall i, as : \big(i\#as \in altsmap(ex') \equiv alts(ex,i) = as \wedge i \in events(ex')\big)$$
$$\wedge \forall i, ns : \big(i\#ns \in ensmap(ex') \equiv (ens(ex,i) - es_o) = ns \wedge i \in events(ex')\big)$$

The first four lines of this definition select es_a and es_o as sets of action and outcome events respectively. The events in es_a are any subset of the action events in the joint execution that have no enablers, and are therefore possible to perform. The set es_o contains one outcome event enabled by each event in es_a.

The remaining four lines specify how the events remaining in the joint execution are updated: events that conflict with the performed events are removed, and the performed events are removed from all lists of enablers.

As an example, consider again the simple joint execution shown in Figure 5.3. The possible values of $es_a \# es_o$ generated by $Perform$ for this joint execution are:

$$\{A1\}\#\{O1\}$$
$$\{A1\}\#\{O2\}$$
$$\{A2\}\#\{O3\}$$
$$\{A1, A2\}\#\{O1, O3\}$$
$$\{A1, A2\}\#\{O2, O3\}$$

These correspond to all potential next steps of execution of this structure. The $Perform$ predicate is clearly quite non-deterministic, permitting any subset of the enabled events to be performed; the different choices it can make correspond to different potential orderings of events when performing the joint execution.

5.2.5 Histories

Every branch identifies a family of potential partial runs of the execution, which are given by the branch's *histories*. The predicate $History$ constructs a branch history by recursively performing events that do not conflict with the branch, until all events in the branch have been performed. This predicate depends on $Perform$ to identify an enabled set of action events es_a and outcome events es_o, the labels of which are translated into action and outcome terms c and y respectively.

$$\begin{aligned}
History(ex, br, h) \equiv \quad & b = \{\} \wedge h = \epsilon \\
\vee \big(& \exists ex', h', br', es_a, es_o, c, y : Perform(ex, es_a, es_o, ex') \\
& \wedge \forall i, j : (i \in br \wedge j \in (es_a \cup es_o) \rightarrow \neg(i \oplus_{ex} j)) \\
& \wedge \forall a : (a \in c \equiv \exists i : i \in es_a \wedge lbl(ex, i) = a) \\
& \wedge \forall agt, o : (o \in y[agt] \equiv \exists i : i \in es_o \wedge o \in lbl(ex, i)[agt]) \\
& \forall i : \big(i \in br' \equiv i \in br \wedge i \in events(ex') \big) \\
& \wedge History(ex', br', h') \wedge h = h' \cdot (c\#y) \big]
\end{aligned}$$

The second and third lines of this definition select sets es_a and es_o that do not conflict with the given branch. The fourth line constructs the concurrent action c as the union of each action in the set es_a, while the fifth line constructs the corresponding outcome y as the agent-wise union of the outcomes in the set es_o. Such pairs $c\#y$ are repeatedly selected until every event in the branch is performed.

Clearly, if there are many unordered events then there are many potential histories for a given branch; in fact there may be exponentially-many histories in general. In Section 5.4 we show how to avoid reasoning about each history individually, which is crucial if these structures are to be of practical use. Instead, we reason about only the *canonical history,* the unique history obtained by performing events in the strict order determined by the $<$ relation:

$$\begin{aligned}
CHistory(ex, br, h) \equiv \quad & b = \{\} \wedge h = \epsilon \\
\vee \big(\exists ex', h', br', es_a, es_o, c, y : \; & Perform(ex, es_a, es_o, ex') \\
\wedge \exists i : \; & es_a = \{i\} \wedge \forall j \in events(ex) : \; i < j \\
\wedge \forall i, j : \; & (i \in br \wedge j \in (es_a \cup es_o) \; \rightarrow \; \neg(i \oplus_{ex} j)) \\
\wedge \forall a : \; & (a \in c \; \equiv \; \exists i : \; i \in es_a \wedge lbl(ex, i) = a) \\
\wedge \forall agt, o : \; & (o \in y[agt] \; \equiv \; \exists i : \; i \in es_o \wedge o \in lbl(ex, i)[agt]) \\
\forall i : \; & \big(i \in br' \; \equiv \; i \in br \wedge i \in events(ex') \big) \\
\wedge \; & History(ex', br', h') \; \wedge \; h = h' \cdot (c\#y) \big]
\end{aligned}$$

For convenience, we also define a predicate Sit that gives the situation terms corresponding to the branch histories:

$$Sit(ex, br, s) \; \equiv \; \exists h : \; History(ex, br, h) \wedge Sit(h) = s$$

From these definitions it should be clear that joint executions constitute a plan of action that can be executed reactively in the world. It is simply a matter of picking some subset of the enabled actions, executing them and obtaining the corresponding outcomes, then rolling the joint execution forward according to the $Perform$ predicate. The set of histories of all leaves of a joint execution gives every possible situation that could be reached by performing it in the world.

Of course, this simple account of performing a joint execution assumes public observability of all actions and outcomes. For a team of agents to be able to feasibly execute it based only on their local information, we must enforce some additional restrictions on the structure of a joint execution.

5.2.6 Feasible Joint Executions

Since we intend for joint executions to be performed reactively by a team of agents in an asynchronous environment, we must formalise the relationship between the global histories of a joint execution and each agent's local view of those histories. First, we define the $View$ function over a history in the obvious way:

$$View(agt, \epsilon) = \epsilon$$
$$y[agt] = \{\} \rightarrow View(agt, (c\#y) \cdot h) = View(agt, h)$$
$$y[agt] \neq \{\} \rightarrow View(agt, (c\#y) \cdot h) = y[agt] \cdot View(agt, h)$$

A branch can *generate a view* if one of its histories corresponds to that view:

$$GeneratesView(ex, br, agt, v) \equiv \exists h : History(ex, br, h) \wedge View(agt, h) = v$$

Let $actor(ex, i)$ be the agent responsible for performing an action event i. Then that event is *enabled by a view* if there is a history of its enablers that can generate that view for the performing agent:

$$EnabledByView(ex, i, agt, v) \equiv$$
$$actor(ex, i) = agt \wedge GeneratesView(ex, ens(ex, i), agt, v)$$

Since an agent's view does not have complete information, $EnabledByView$ identifies events that the agent *might* be required to perform based on its local information. It is not sufficient to precisely identify a particular branch, and therefore cannot be used to determine for certain whether any particular event should be performed.

To ensure the feasibility of performing a joint execution based on each agent's local information, we must assert two additional structural restrictions to ensure each agent can always determine the action it is to perform.

The first restriction corresponds to the idea of *knowing when* to perform an action. If an action event i is enabled by an outcome event j, then j must not be hidden from the agent performing i. Otherwise, it has no way of enforcing the required ordering between the two events. This requirement is formalised by:

$$KnowsWhen(ex) \stackrel{\text{def}}{=}$$
$$\forall i, j \in events(ex) : IsAction(lbl(ex, i)) \wedge j \in ens(ex, i)$$
$$\rightarrow lbl(ex, i)[actor(ex, i)] \neq \{\}$$

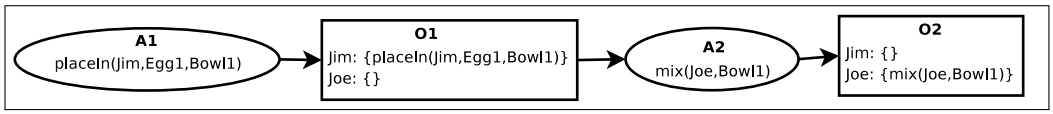

Figure 5.4: A joint execution that violates the $KnowsWhen$ restriction

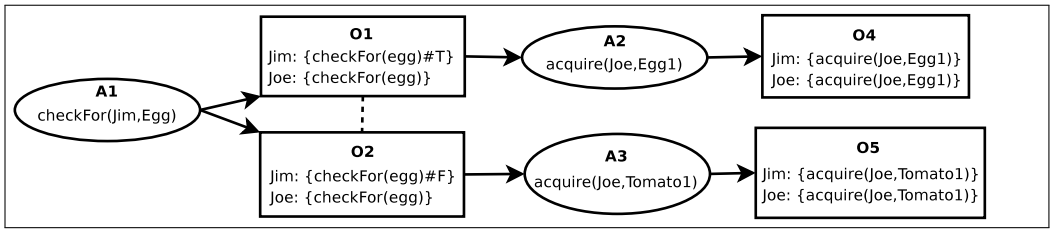

Figure 5.5: A joint execution that violates the $KnowsWhat$ restriction

Figure 5.4 shows an example of a joint execution that does not meet this requirement. This plan calls for Jim to place an egg in the bowl, and then for Joe to mix the bowl's contents. However, since Joe cannot observe the occurrence of Jim's action, he cannot enforce the ordering between these two events.

The second restriction corresponds to the idea of *knowing what* action to perform. For any given view v, there may be multiple branches with a history that could generate that view, and the agent has no means of knowing precisely which branch has been performed in the world. We require that if an event is enabled by a view, then *all* branches that could generate that view enable a similar event:

$$KnowsWhat(ex) \stackrel{\text{def}}{=}$$
$$\forall agt, v, i, br: \; EnabledByView(ex, i, agt, v) \wedge GeneratesView(ex, br, agt, v) \rightarrow$$
$$\exists j: \; ens(ex, j) = br \wedge lbl(ex, i) = lbl(ex, j)$$

While the agent may not know precisely which *event* is enabled, its local information is enough to determine the specific *action* that it is to perform. Figure 5.5 shows an example of a joint execution that does not meet this requirement. This plan calls for Jim to check for the availability of eggs, then for Joe to acquire an appropriate ingredient depending on whether they are available. But since Joe cannot distinguish between outcome events $O1$ and $O2$ after he observes $checkFor(egg)$, he doesn't know what action to perform and the plan cannot be executed.

We say a joint execution is *feasible* if it meets both of these restrictions:

$$Feasible(ex) \stackrel{\text{def}}{=} KnowsWhat(ex) \wedge KnowsWhen(ex)$$

5.2.7 Legal Joint Executions

So far, we have not restricted joint executions to correspond to any sort of *legal* run of execution. A joint execution may have action events enabling outcome events that they would never produce under the given theory of action. It may call for actions to be performed in situations where they are not legal, or allow actions to be performed concurrently that could be in conflict.

To avoid such undesirable cases, we identify *legal* joint executions as ones that are constrained enough to be performed in the real word. We will say that a particular leaf of a joint execution is legal if every history of that leaf is legal:

$$Legal(ex, lf) \stackrel{\text{def}}{=} \forall h : History(ex, lf, h) \rightarrow Legal(h)$$

This ensures that the leaf is constrained enough to prevent precondition interaction between independent action events, that its outcome events are correct for their corresponding actions, etc. However, the agents will generally not have enough information to determine whether a particular leaf is legal, since this would imply that they already know what sensing results will occur. We call an entire joint execution legal if is proper and contains a legal leaf:

$$Legal(ex) \stackrel{\text{def}}{=} Proper(ex) \wedge \exists lf : Leaf(ex, lf) \wedge Legal(ex, lf)$$

This definition does not require that we establish *which* leaf is legal, only that we are able to prove that *some* leaf must be legal. In practise this would be done by enumerating the possible outcomes of each action. Since the leaves of a joint execution represent all its possible terminating configurations, this requirement means that a legal joint execution can legally be performed to completion in the world.

The definition is also *permissive*, in that there may be leaves of the joint execution that are provably never be legal. Since the outcomes along these leaves will not occur in reality, the agents will never follow them at execution time. This permissiveness will therefore not affect an agent's ability to carry out the plan in practice.

5.2.8 Summary

This section has formally defined a *joint execution*, a partially-ordered branching action structure that we claim is particularly well suited for representing the actions to be performed by a team of agents in service of a shared task. The partially-ordered nature of joint executions allows them to explicitly account for inter-agent synchronisation of actions in the face of partial observability, while their branching

nature allows them to account for incomplete information that must be augmented with runtime sensing results.

In asynchronous domains, where raw situation terms cannot feasibly be executed in the world, joint executions are an ideal alternative as a plan representation structure for use by the Golog execution planning process. In the next section we identify precisely what such a planning process would entail.

5.3 Planning with Joint Executions

With the above definitions and axioms in place, we are now in a position to plan the cooperative execution of a shared Golog program using joint executions rather than raw situation terms. For the moment we focus on *offline* execution planning, in the style of the original Golog and ConGolog. Recall that the semantics of execution planning in Golog involve finding a situation term s satisfying:

$$\mathcal{D} \cup \mathcal{D}_{golog} \models \exists s : \mathbf{Do}(\delta, S_0, s)$$

Before extending this query to search for a joint execution, notice an important consequence of our definitions: two events can occur in either order if and only if they can also occur concurrently. Since the standard Golog/ConGolog semantics do not permit true concurrency, they would force all events to be ordered and we would gain no benefit from using joint executions. We must therefore adopt the concurrency semantics of MIndiGolog from Chapter 3, which permit true concurrency of actions.

The execution planning problem then reduces to the task of finding a *legal*, *feasible* joint execution such that for every leaf, if that leaf is legal, then it constitutes a legal execution of the program:

$$\begin{aligned}\mathcal{D} \cup \mathcal{D}_{mgolog} \cup \mathcal{D}_{je} \models \exists ex : \; & Legal(ex) \wedge Feasible(ex) \wedge \\ & \forall lf : Leaf(ex, lf) \wedge Legal(ex, lf) \rightarrow [\forall s : Sit(ex, lf, s) \rightarrow \mathbf{Do}(\delta, S_0, s)] \end{aligned} \quad (5.1)$$

This query neatly captures dual soundness and completeness requirements. For soundness, it requires that for every leaf of the joint execution, *if* that leaf is legal then it will be a legal execution of the program δ. For completeness, it requires that there must in fact be *some* leaf that is legal, so the joint execution can actually be performed in the world. The joint execution must contain enough branching to account for any incomplete knowledge the agents have about the state of the world.

Algorithm 4 presents a simple modification of the Golog offline planning algorithm that can be used by each agent to plan the execution of a shared program δ

Algorithm 4 Offline Execution Algorithm using Joint Executions

$v \Leftarrow \epsilon$
Find a joint execution ex such that:

$$\mathcal{D} \cup \mathcal{D}_{mgolog} \cup \mathcal{D}_{je} \models \exists ex : Legal(ex) \wedge Feasible(ex) \wedge \\ \forall lf : Leaf(ex, lf) \wedge Legal(ex, lf) \rightarrow [\forall s : Sit(ex, lf, s) \rightarrow \mathbf{Do}(\delta, S_0, s)]$$

while ex contains action events to be performed by me **do**
  Find an action a such that

$$\mathcal{D} \cup \mathcal{D}_{mgolog} \cup \mathcal{D}_{je} \models \exists i : EnabledByView(ex, i, agt, v) \wedge lbl(ex, i) = a$$

  if there is such an action **then**
    Execute action a
  end if
  Wait for a new observation o
  $v \Leftarrow o \cdot v$
end while

and then perform it in the world. A restricted version of this algorithm is used by our implementation that will be described in Section 5.5.

Since this algorithm is to be executed independently by each agent in the team, it must identify actions to perform using only the agent's local view. We restrict the joint execution to be feasible so that the $EnabledByView$ query is sufficient to identify what action to perform next. If we did not have this restriction, Algorithm 4 would not be correct.

If we turn our attention to *online* execution in the style of IndiGolog, things are not so straightforward. Although we have not presented the axioms for doing so, it is simple enough to extend the leaves of a joint execution one action at a time in the style of the IndiGolog execution algorithm presented in Algorithm 2. The difficulty comes in trying to coordinate this process across multiple agents when they have differing knowledge about the state of the world.

To demonstrate the issues involved, consider the hypothetical, ***incorrect*** online execution algorithm presented in Algorithm 5, which mirrors the ReadyLog execution algorithm used by our first MIndiGolog implementation. Since joint executions are a branching structure, the agent must extend each leaf of the joint execution with a new step of execution of the program; if any leaf cannot be extended then execution will potentially fail. To avoid this the agent discards leaves that it knows are not legal before planning the next step of execution.

However, the implicit coordination scheme used by ReadyLog and MIndiGolog

Algorithm 5 Hypothetical (Incorrect) Online Execution Algorithm

$v \Leftarrow \epsilon$
$ex \Leftarrow jexec(\{\}, \{\}, \{\}, \{\})$
while δ is not final according to my current view v **do**
  Discard any leaves of ex incompatible with v
  Extend each leaf of ex with a legal step of δ
  Find an action in ex that is enabled by v
  if there is such an action **then**
    Execute that action
  end if
  Wait for a new observation o
  $v \Leftarrow o \cdot v$
end while

depends on all agents generating the same "next step" at every iteration. It is therefore incorrect for the agent to discard leaves based only on its local information – it must retain any leaves that its teammates could still consider possible, in order to guarantee that they generate the same plan. Worse, it must also consider that its teammates will retain leaves that they think *it* could still consider possible, and so-on ad infinitum.

The difficulty here is the well-known correspondence between coordination and common knowledge [39]. In order to extend this execution algorithm to the case of incomplete information, the agents must plan based on what is *commonly known* at each step of execution, rather than based on their own individual view. Unfortunately the situation calculus currently offers no tools for reasoning about common knowledge, not even in synchronous domains.

Coordinating the online execution of a shared Golog program in asynchronous domains thus requires more explicit reasoning about the knowledge of each agent, and the common knowledge of the team. In the coming chapters of this thesis we will explore the foundations for such reasoning, but we are yet to incorporate it into our implementation. Our joint-execution based MIndiGolog planner is therefore currently limited to offline execution planning.

5.4 Reasonable Joint Executions

While joint executions can clearly provide a powerful formal account of execution planning for asynchronous multi-agent domains, in their current form they are not suitable for an effective implementation. The difficulty arises from the definition of $History(ex, br, h)$, which due to the partial ordering on events can generate an

exponentially-large number of possible histories. To verify that a joint execution is legal, the planner needs to examine each of these histories individually.

To overcome this difficulty and produce an effective implementation, we identify a restricted class of joint executions in which all possible histories of a branch are provably equivalent. Such executions can be reasoned about using the canonical ordering over events, rather than having to enumerate each possible distinct history.

5.4.1 Independent Actions

To construct families of situation terms that are all equivalent, we need a way to identify *independent actions.* Intuitively, we want independent actions to be able to be performed in either order, or even concurrently, without affecting what holds in the resulting situation, or the preconditions or outcomes of each action. This section formally identifies the conditions that independent actions must satisfy.

For simplicity, we identify actions that are independent regardless of the situation they are performed in. Let us assume that the theory of action $\mathcal{D}$ is equipped with a rigid predicate $indep(a, a')$ identifying actions that are independent. We identify sets of mutually-independent actions with this simple definition:

$$mIndep(c) \stackrel{\text{def}}{=} \forall a, a' : a \in c \wedge a' \in c \rightarrow indep(a, a')$$

We then restrict the theory of action to satisfy the following conditions:

Definition 16 (Independent Actions)**.** *A theory of action $\mathcal{D}$ correctly specifies independent actions when it contains a rigid predicate $indep(a, a')$ and entails the following, where $\mathcal{F}$ is a meta-variable ranging over fluents:*

1. $\mathcal{D} \models indep(a, a') \equiv indep(a', a)$
2. $\mathcal{D} \models Legal(\{a\}, s) \equiv Legal(\{a\}, do(\{a'\}, s))$
3. $\mathcal{D} \models Out(\{a\}, s) = Out(\{a\}, do(\{a'\}, s))$
4. $\mathcal{D} \models \mathcal{F}(do(\{a\}, do(\{a'\}, s))) \equiv \mathcal{F}(do(\{a'\}, do(\{a\}, s)))$
5. $\mathcal{D} \models mIndep(c) \rightarrow (Legal(c, s) \equiv \forall a \in c : Legal(\{a\}, s))$
6. $\mathcal{D} \models mIndep(c) \rightarrow (o \in Out(c, s)[agt] \equiv \exists a \in c : o \in Out(\{a\}, s)[agt])$
7. $\mathcal{D} \models mIndep(c) \rightarrow \forall a \in c : (\mathcal{F}(do(c, s)) \equiv \mathcal{F}(do(\{a\}, do(c - \{a\}, s))))$

The first restriction simply ensures that independence is symmetrical. The next three restrictions ensure that independent actions do not interfere with each other's

preconditions, outcomes or effects. The final three restrictions ensure that there is no interference between preconditions, outcomes or effects when independent actions are performed concurrently.

The following theorems are direct consequences of correctly specifying independent actions; indeed, they are the motivation for the restrictions in Definition 16.

Theorem 2. *Let h and h' be two histories of the same length, containing the same action#outcome pairs, and differing only by transposition of $(\{a\}\#y)$ and $(\{a'\}\#y')$. If $indep(a, a')$ holds, then h is legal if and only if h' is legal.*

Proof. Let h_p be the common prefix of these histories and h_s the common suffix:

$$\begin{aligned} h &= h_s \cdot (\{a\}\#y) \cdot (\{a'\}\#y') \cdot h_p \\ h' &= h_s \cdot (\{a'\}\#y') \cdot (\{a\}\#y) \cdot h_p \end{aligned}$$

By restrictions 2 and 3 from Definition 16, we have:

$$\begin{aligned} Legal(\{a\}, Sit(h_p)) &\equiv Legal(\{a\}, Sit((\{a'\}\#y') \cdot h_p)) \\ Out(\{a\}, Sit(h_p)) &= Out(\{a\}, Sit((\{a'\}\#y') \cdot h_p)) \end{aligned}$$

And vice-versa. If h_s is empty, this is sufficient to establish $Legal(h)$ iff $Legal(h')$ as desired. Alternately, suppose h_s contains n items, then we can apply regression n times to state the legality of the h_s component as a uniform formula evaluated at $Sit((\{a\}\#y) \cdot (\{a'\}\#y') \cdot h_p)$. By restriction 4, whether this formula holds will be unaffected by the order of $\{a\}$ and $\{a'\}$ and we have the equivalence as desired. □

Theorem 3. *Let h and h' be two histories that differ only by the concurrent execution of adjacent mutually-independent actions, and the corresponding agent-wise union of their outcomes. Then h is legal iff h' is legal.*

Proof. Assume that the histories differ by concurrent execution of a single action. Let h_p be the common prefix of these histories and h_s the common suffix:

$$\begin{aligned} h &= h_s \cdot (\{a\}\#y) \cdot (\{c\}\#y') \cdot h_p \\ h' &= h_s \cdot (c \cup \{a\}\#y'') \cdot h_p \end{aligned}$$

Furthermore, we're given that:

$$agt\#o \in y'' \equiv o \in y[agt] \vee o \in y'[agt]$$
$$mIndep(c \cup \{a\})$$

By mutual independence, and restrictions 2 and 5, we have:

$$Legal(\{a\}, Sit((\{c\}\#y') \cdot h_p)) \equiv Legal(\{a\}, Sit(h_p))$$
$$Legal(c \cup \{a\}, Sit(h_p)) \equiv Legal(\{a\}, Sit(h_p)) \wedge \forall a' \in c : Legal(\{a'\}, Sit(h_p))$$

Similarly for outcomes, using restrictions 3 and 6:

$$Out(\{a\}, Sit((\{c\}\#y') \cdot h_p)) = Out(\{a\}, Sit(h_p))$$
$$o \in Out(c \cup \{a\}, Sit(h_p))[agt] \equiv o \in (Out(\{a\}, Sit(h_p)) \cup Out(c, Sit(h_p)))\,[agt]$$

This is sufficient to establish $Legal(h)$ iff $Legal(h')$ if h_s is empty. Alternately, suppose h_s contains n items, then we can apply regression n times to state the legality of the h_s component as a uniform formula evaluated at $Sit((c \cup \{a\}\#y'') \cdot h_p)$. By restriction 7, whether this formula holds will be unaffected if $\{a\}$ is executed separately, and we have the equivalence as desired.

If the histories differ by concurrent execution of more than a single action, we can simply unfold them into a sequence of histories differing by only one action, with each being legal iff its adjacent history is legal. □

Note that we make no attempt to derive action independence from the theory of action, but simply assume an appropriate predicate $indep(a, a')$ is available for the purposes of planning. This predicate need not identify *all* independent actions, although the more actions that can be identified as independent, the better for our implementation.

5.4.2 Reasonability

We can now define a *reasonable* joint execution as one in which every pair of action events is either ordered, in conflict, or independent:

Definition 17 (Reasonable Joint Execution)**.** *A joint execution is reasonable if it satisfies the following restriction:*

$$\mathcal{D} \models \forall i, j \in events(ex) : IsAction(lbl(ex, i)) \wedge IsAction(lbl(ex, j)) \rightarrow i \prec_{ex} j \vee j \prec_{ex} i \vee i \oplus_{ex} j \vee indep(lbl(ex, i), lbl(ex, j))$$

We call such executions "reasonable" because a planner can reason about then effectively, using the unique canonical history of each leaf rather than enumerating every individual history.

Theorem 4. *Let ex be a reasonable joint execution, then:*

$$\mathcal{D} \cup \mathcal{D}_{je} \models \forall lf : Leaf(ex, lf) \rightarrow [Legal(ex, lf) \equiv \exists h : CHistory(ex, lf, h) \wedge Legal(h)]$$

Proof. By definition, a leaf is legal if every possible history of that leaf is legal, so the *if* direction is trivial. For the *only-if* direction, assume that the canonical history of the leaf if legal. The histories of a leaf can differ only by the ordering or concurrent execution of unordered action events, and all unordered action events in a reasonable execution are independent. Therefore every history of the leaf differs from the canonical history by transposition or concurrent execution of independent action events. So if the canonical history is legal, by Theorems 2 and 3 we have legality of every history and hence legality of the leaf as required. □

This result is key to our implementation of a MIndiGolog execution planner based on joint executions - by restricting its search to reasonable executions, it can verify the legality of each leaf by querying the legality of the canonical leaf history, which can be done using standard regression techniques.

We thus trade completeness for efficiency in our implementation. There can certainly be non-reasonable joint executions that are valid plans of execution according to equation (5.1), but it is computationally too expensive to search for them in practice.

5.5 Implementation

We have modified our MIndiGolog execution planner from Chapter 3 to perform offline execution planning and generate a joint execution rather than a raw situation term. For details on obtaining the full source code see Appendix B; for the full axiomatisation of observability in our example domain see Appendix C.

As mentioned in Section 5.2, Figure 5.2 shows the output of our planner when run on the $MakeSalad$ program from Chapter 3. Since all actions in this execution have a single outcome, the outcome events have been suppressed for brevity.

In the cooking agents domain, actions are independent if they deal with different objects. As seen in Figure 5.2, the use of a partial order structure facilitates independent execution between the agents, with each processing a different ingredient and only synchronising on the availability of the required resources. This execution provides the maximum potential for concurrency given the resource constraints of the domain, and is clearly a significant improvement over totally ordered sequences

of actions as produced by the earlier MIndiGolog planner.

However, the simple $MakeSalad$ program does not demonstrate a key feature of joint executions: branching. Consider instead the program $MakeSalad2$ shown in Figure 5.6. In this case the agents are unsure whether there are any eggs available, so the sensing action *checkFor* is required. If there are eggs then they should make an egg salad, otherwise they should make the standard vegetable salad. Note that since lettuce appears in both dishes, they are permitted to begin processing that ingredient before checking for the eggs.

$$
\begin{array}{c}
\textbf{proc}\, MakeSalad2(dest) \\
[\ \pi(agt, ChopTypeInto(agt, Lettuce, dest)) \,||\, \\
ChopEggOrVeg(dest)\ \ ]\ ; \\
\pi(agt, [acquire(agt, dest)\,; \\
beginTask(agt, mix(dest, 1))\,; \\
endTask(agt, mix(dest, 1))\,; \\
release(agt, dest)])\ \textbf{end} \\
\\
\\
\textbf{proc}\, ChopEggOrVeg(dest) \\
\pi(agt,\ checkFor(agt, Egg))\,; \\
\textbf{if} \\
\exists e :\ IsType(e, Egg) \wedge \neg Used(e) \\
\textbf{then} \\
[\pi(agt, ChopTypeInto(agt, Egg, dest)) \,||\, \\
\pi(agt, ChopTypeInto(agt, Cheese, dest))] \\
\textbf{else} \\
[\pi(agt, ChopTypeInto(agt, Carrot, dest)) \,||\, \\
\pi(agt, ChopTypeInto(agt, Tomato, dest))]\ ; \\
\textbf{endif end}
\end{array}
$$

Figure 5.6: A Golog program for making Egg or Veg Salad

The joint execution found by our implementation for $MakeSalad2$ is shown in Figure 5.7. The event nodes in this diagram are colour-coded into three groups: white nodes can occur independently of the sensing results from *checkFor*; light-grey nodes can only occur if *checkFor* returns false; dark-grey nodes can only occur if *checkFor* returns true.

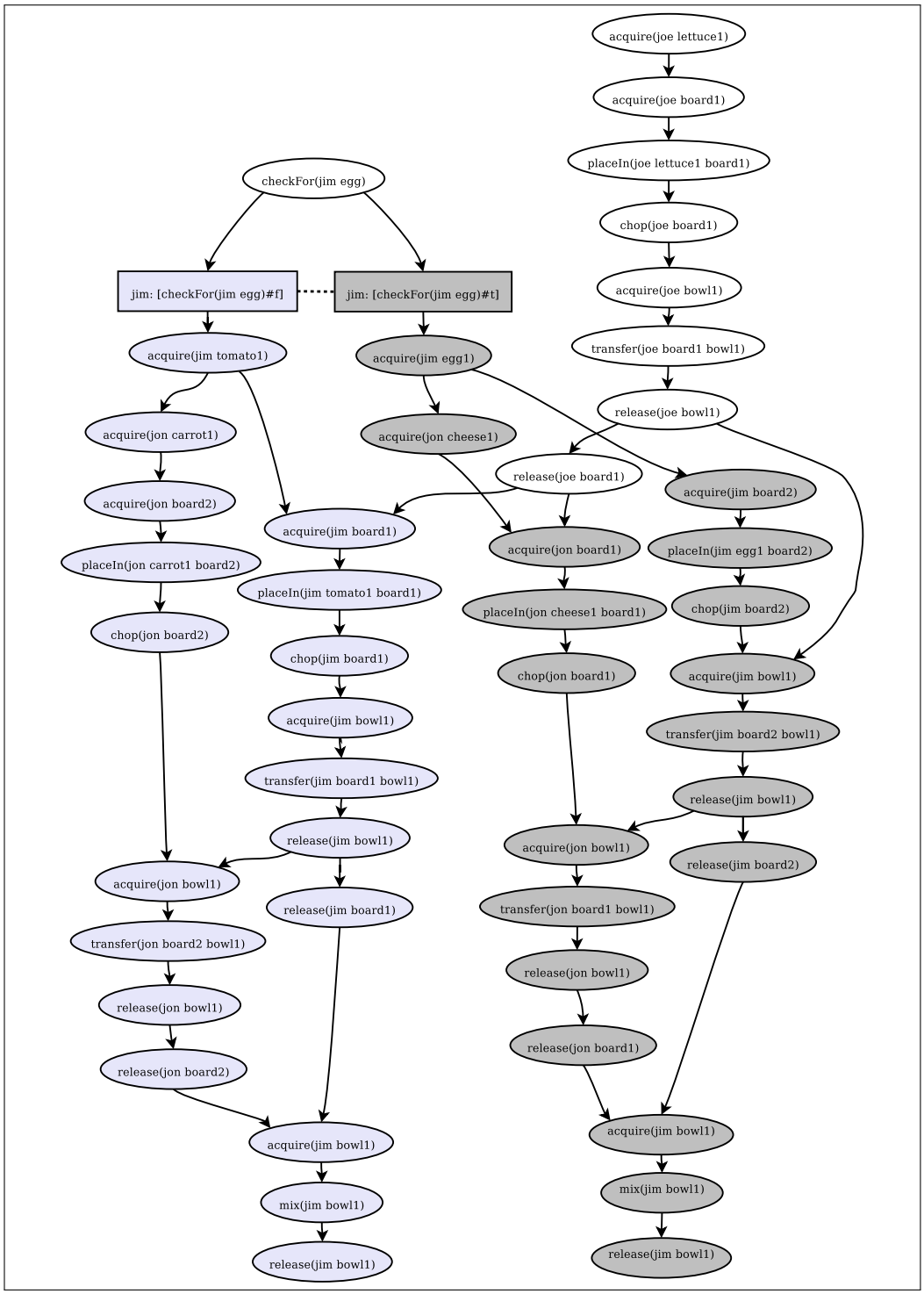

Figure 5.7: Joint Execution for the *MakeSalad*2 program

We see from this joint execution that *Joe* can indeed prepare the lettuce without needing to know whether eggs are available. *Jim* is assigned to check for the eggs, and acquires either an egg or a tomato depending on the outcome of his sensing action. Importantly, *Jon* has to wait until *Jim* acquires his ingredient before he knows whether to process the cheese, or the carrot. This is due to him being unable to directly observe the outcome of the *checkFor* action.

Again, there is a significant amount of independence between agents in this execution. They do not need to be able to observe the private processing actions such as *chop* and *mix*, and only need to synchronise their actions when they come to *release*/*acquire* the shared resources. By basing branching and synchronisation directly on the observations made by each agent, joint executions allow us to capture this kind of rich branching and partial-order structure while ensuring that the agents can still feasibly execute the plan based solely on their local information.

In the following sections we highlight some key aspects of our implementation.

5.5.1 Program Steps

The *Trans* predicate of MIndiGolog is modified to generate *steps* instead of constructing a new situation term. These are records that describe not only the next action to perform, but also meta-data about that action's role in the overall program. Step records have the following attributes.

- action: the action performed in that step, or *nil* if it is an internal program transition
- test: an additional fluent formula that must hold immediately before performing the step
- thread: a sequence of 'l' and 'r' characters indicating the concurrent thread in which the step is performed
- outcome: the outcome of performing the action.

Step records track the necessary information to determine whether the program allows any given pair of actions to be performed independently. A sequence of steps can be used as a history term in the obvious way, taking only the actions and outcomes.

The thread-naming scheme used here is similar to that of [33]. Each time *Trans* chooses to execute a step from the left-hand side of a concurrency operator it appends an "l" to the thread name, and each time it chooses the right-hand side it appends

an "r". If one step's thread name is a prefix of another step's thread name, then those two steps must be performed in the order they are generated; if not, they are steps from different threads and can potentially be performed concurrently.

The procedure implementing *Trans* takes a program and a history as input, returning a new step of execution along with the remainder of the program to be executed. The code below is representative of this procedure:

```
proc {Trans D H Dp Sp}
  case D of
      nil then fail
  []  test(Cond) then
        {Sitcalc.holds Cond H}
        Dp = nil
        Sp = {Step.init step(test:Cond)}
  []  choose(D1 D2) then
        choice {Trans D1 H Dp Sp}
        []  {Trans D2 H Dp Sp}
        end
  []  conc(D1 D2) then
        choice D1p S1p in
            {Trans D1 H D1p S1p}
            Dp = conc(D1p D2)
            Sp = {Step.addthred S1p l}
        []  D2p S2p in
            {Trans D2 H D2p S2p}
            Dp = conc(D1 D2p)
            Sp = {Step.addthred S2p r}
        end
  [] ... <additional cases ommitted> ...
  else  Dp = nil
        {Sitcalc.legal [D] H}
        Sp = {Step.init step(action:D)}
       end
  end
end
```

In particular, note that the evaluation of test conditions calls *Sitcalc.holds* passing it the input history. This procedure performs standard regression-based reasoning using a "just-in-time history" assumption to handle sensing results, in the same manner as standard IndiGolog [17]. The planner ensures that this is the canonical history of the leaf that is being planned for, so we can be sure that the test will hold in all possible histories of the leaf if it holds in the given history.

We say that two steps are *ordered* if any of the following holds: their action terms are not independent; one's thread is a prefix of the other; one's action falsifies the

test condition associated with the other. When building a joint execution, ordered steps are forced to be executed in the order they were generated by the planner, while unordered steps may be performed independently and potentially concurrently.

Note that we no longer require a separate clause to handle true concurrency in the $conc(D1\, D2)$ case, unlike the MIndiGolog implementation from Chapter 3. Rather, the potential for truly concurrent execution is captured in the partial ordering of the joint execution itself, which allows independent actions to be performed concurrently while forcing non-independent actions to occur in a particular order.

5.5.2 Building Joint Executions

Our implementation builds up joint executions by inserting one action at a time, in much the same way that the standard Golog planning loop builds up situation terms. The procedure *Insert* is called with the step object whose action is to be inserted, the leaf for which it is a new program step, and a function $MustPrec$ that will be used to determine the action's enablers.

```
proc {Insert JIn Lf S MustPrec JOut Outcomes}
  PossEns = {FindEnablingEvents JIn S.action Lf MustPrec}
  Ens = {FilterEnablers JIn PossEns}
in
  {InsertWithEnablers JIn Lf S Ens JOut Outcomes}
end

proc {InsertWithEnablers JIn Lf S Ens JOut Outcomes}
  Outs = {Sitcalc.outcomes S}
  AId|OIds = {IntMap.nextAvailLabels JIn S|Outs}
  J1 J2
in
  J1 = {IntMap.append JIn act(action: S.action
                              enablers: Ens
                              outcomes: OIds)}
  J2 = {InsertOutcomes AId J1 Outs OIds}
  JOut = {FixFeasibility J2 AId}
  Outcomes = for collect:C I in OIds do
               {C {BranchPush JOut I Lf}}
             end
end
```

The joint execution code enumerates all possible outcomes of the action and inserts corresponding outcome events. This extends the input leaf by the given action, and if the input leaf was legal, then one the new leaves so generated is also

guaranteed to be legal. The updated joint execution is returned along with the new outcome events.

The call to $FixFeasibility$ ensures that the joint execution remains feasible, by inserting additional action events if it discovers branches that have identical views for the performing agent. Currently this is done using a brute-force search through all possible histories, but it is able to eliminate many branches quickly by first checking whether they will always contain an incompatible observation.

When determining the enablers for a new action, the joint execution code has potentially many choices, and generates choice points accordingly. It processes all existing events on the leaf in turn, first checking if they are *orderable* according to the restrictions on feasible joint executions. If they are orderable, the function $MustPrec$ is called to determine whether they must be *ordered* according to the semantics of the program. If they are orderable, but need not be ordered, a choice point is generated.

```
proc {FindEnablingEvents J Act Ns MustPrec Ens}
  case Ns of N|Nt then
    RemNs = {BranchPop J Ns _} in
    if {Orderable J N Act} then
      if {MustPrec N} then
        % Orderable, and must preceed, so it's an enabler.
        Ens = N|_
        {FindEnablingEvents J Act Nt MustPrec Ens.2}
      else
        % Orderable +  not must prec == choicepoint
        choice {FindEnablingEvents J Act RemNs MustPrec Ens}
        []     Ens = N|_
               {FindEnablingEvents J Act Nt MustPrec Ens.2}
        end
      end
    else
      % Not orderable, so {MustPrec} must return false
      {MustPrec N} = false
      {FindEnablingEvents J Act RemNs MustPrec Ens}
    end
  else Ens = nil end
end
```

5.5.3 Planning Loop

The main execution planning loop operates by extending a joint execution one leaf at a time. At each iteration, the current state of the plan is represented by the joint

execution built so far, along with a list of program#history#leaf tuples tracking each leaf in the joint execution. The history here is the list of program steps performed on that leaf, and also gives the canonical leaf history, while the program represents what remains to execute on that leaf. The planning loop can only terminate when every leaf has a program that is final in its canonical history.

The top-level procedure *Plan* takes a program as input, and calls *MakePlan* with an empty joint execution and a single, empty leaf:

```
proc {Plan D J}
  {MakePlan {JointExec.init} [D#now#nil] J}
end
```

MakePlan is a recursive procedure implementing the planning loop. Note that it cannot discard leaves or process them in isolation, since extending one leaf with an action may cause actions to be added to other leaves in order to maintain the feasibility restrictions.

```
proc {MakePlan JIn Leaves JOut}
  LCls LRest
in
  {FindOpenLeaf JIn Leaves LCls LRest}
  case LRest of (D#H#N)|Ls then Dp Hp S J2 OutNs OutLs in
     {FindTrans1 D H Ls Dp Hp S}
     OutNs = {JointExec.insert JIn N S {MkPrecFunc S Hp} J2}
     OutLs = for collect:C N2 in OutNs do
                  {C Dp#ex({JointExec.getout J2 N2 S} Hp)#N2}
             end
     {MakePlan J2 {List.append LCls {List.append OutLs Ls}} JOut}
  else
    JOut = JIn
  end
end
```

Each iteration of the planning loop proceeds as follows. First, it searches for an *open leaf,* one for which a terminating execution of the program has not yet been found. If no open leaves are found, planning can terminate. Otherwise, the procedure *FindTrans*1 is called to find a new step of execution for that leaf. The action is inserted into the joint execution, which returns a list of new leaves, one for each possible outcome of the action. Each is added to the list of leaves to be processed, and the loop repeats.

The procedure to find an open leaf must also deal with any new events that were inserted into the joint execution to maintain its feasibility invariants. The procedure *HandleExistingEvents* rolls the leaf forward to account for these new events, or fails if an event was added that does not form part of a legal program execution.

```
proc {FindOpenLeaf J Leaves LCls LRest}
  case Leaves of (D1#H1#N1)|Ls then D H N NewLs in
      (D#H#N)|NewLs = {HandleExistingEvents J D1#H1#N1}
      if {MIndiGolog.isFinal D H} then
        LCls = (D#H#N)|_
        {FindOpenLeaf J {List.append NewLs Ls} LCls.2 LRest}
      else
        LClosed = nil LRest = (D#H#N)|{List.append NewLs Ls}
      end
  else LCls = nil LRest = nil end
end
```

Of particular interest is the procedure $FindTrans1$, which uses the encapsulated search functionality of Mozart to yield possible next steps according to an estimate of their potential for concurrency. The procedure *LP.yieldOrdered* yields the solutions of the given search context, sorted using the procedure *CompareSteps*. This procedure gives preference to steps that can be performed concurrently with as many existing actions as possible.

```
proc {FindTrans1 D H Ls Dp Rp S}
  Searcher SearchProc
in
  proc {SearchProc Q} Dp Hp S in
    {MIndiGolog.trans1 D H Dp Hp S}
    Q = Dp#Hp#S
  end
  Searcher = {New Search.object script(SearchProc)}
  Dp#Rp#S = {LP.yieldOrdered Searcher CompareTrans1}
end
```

This use of encapsulated search allows our implementation to find highly concurrent executions, such as the one shown in Figure 5.7.

5.6 Discussion

In this chapter we have defined a *joint execution* as a special kind of prime event structure. We contend that such structures are highly suitable for planning the actions to be performed by a team of agents in service of some shared task, such as executing a shared Golog program.

On one hand, joint executions are restricted enough to be practical for such use. By limiting ourselves to *reasonable* joint executions, each leaf can be easily converted into a single history term for the purposes of reasoning, and can be extended one action at a time. This allows us to re-use much of the standard IndiGolog reasoning machinery. By ensuring that the joint execution is *feasible*, the agents are guaranteed to be able to perform it in a purely reactive fashion, with each agent acting based only on its local information.

Joint executions are also significantly more flexible than previous approaches. They allow independent actions to be performed without synchronisation, in any order. The agents need never know precisely what actions have been executed, as long as their local observations are sufficient to determine the next action to perform. Synchronisation is automatically achieved when required by explicitly reasoning about what actions each agent can observe, rather than requiring the public observability of all actions.

To demonstrate the utility of these structures, we have implemented a new version of our MIndiGolog interpreter that produces joint executions as its output, and shown that the resulting executions can enable significant independence among agents when cooperatively executing the plan.

An alternate approach to coordinating concurrent execution in Golog-like languages is the TeamGolog language developed in [26], where agents explicitly synchronise through communication and a shared state. By contrast, our approach constructs synchronisation implicitly by reasoning about the actions that can be observed by each agent. This has the advantage of requiring no changes to the form or semantics of the agents' control program, but the disadvantage that joint execution construction may fail if too many actions are unobservable. It would be interesting to combine these approaches by automatically incorporating explicit communication when implicit synchronisation is not possible.

There is, of course, an extensive body of work on partial-order planning in the context of goal-based planning. Unsurprisingly, the joint execution structure we develop here has deep similarities to the structures used in conditional partial-order planners such as [75]. It is, however, intentionally specific to the situation calculus.

We make no use of many concepts common in partial-order goal-based planning (causal links, threats, conflicts, etc) because we do not deal explicitly with goals, but with steps generated by an underlying transition semantics. Our approach can be considered roughly equivalent to *deordering* of a totally-ordered plan as described in [3]; we plan as if actions are performed in the specific order identified by the canonical leaf history, but allow actions to be performed out-of-order if they are independent.

Our use of a restrictive plan representation that branches directly on the sensing results returned by actions has strong parallels with the "robot programs" of [61, 64], but is significantly less expressive. In particular, joint executions do not allow looping constructs and thus lack the universality of general robot programs. It would be interesting to incorporate loops in our structures, but how to do so is far from clear in the face of partial observability. Indeed, producing iterative plans is still an active area of research even in the single-agent case [60].

By explicitly formalising the local perspective of each agent, we have given an account of planning with coordination and feasibility guarantees without needing to perform explicit epistemic reasoning. On one hand, this means we can implement a practical planning system without concern for the computational difficulties involved in epistemic reasoning. But this has also limited us to purely offline planning, since the correctness of the algorithm depends crucially on all agents having the same knowledge of the domain.

As discussed in Section 5.3, extending the use of joint executions for online execution in asynchronous domains poses a significant challenge, and seems to require explicit reasoning about knowledge and common knowledge. The remainder of this thesis is devoted to developing the foundations of such a reasoning system, by extending the standard account of epistemic reasoning in the situation calculus to handle asynchronous domains.

Chapter 6

Property Persistence

This chapter develops a new inductive reasoning technique for the situation calculus that can handle certain types of universally-quantified query. As discussed in Chapter 4, for an agent in an asynchronous domain to reason about the world based on its local information, it needs to pose queries that universally quantify over situation terms. Unfortunately such queries cannot be handled using the regression operator, and have thus far been beyond the reach of automated reasoning systems for the situation calculus.

We study a restricted subset of universally-quantified queries that we refer to as *property persistence queries*, introducing an approach to reasoning about them that is similar in spirit to the standard regression operator: transform the query into a form more amenable to automated reasoning. A new meta-operator $\mathcal{P}_{\mathcal{D}}$ is defined such that ϕ persists in s if and only if $\mathcal{P}_{\mathcal{D}}(\phi)$ holds in s. We term the formula generated by this operator the *persistence condition* of ϕ.

The persistence condition is shown to be a fixpoint of applications of the regression operator, which can be calculated using an iterative approximation algorithm. The resulting formula can then be used in combination with standard regression-based reasoning techniques, allowing the inductive component of the reasoning to be "factored out" and approached using a special-purpose reasoning algorithm. The technique is always sound, and is complete in several interesting cases.

Chapter 4 identified a universally-quantified query with which an agent can reason about its own world based on its local view. This query is *not* in a form that that can be handled directly using the persistence condition. However, Chapter 7 will demonstrate how to combine the techniques developed in this chapter with a new formalism for epistemic reasoning, allowing an agent to reason effectively about its own knowledge using a combination of regression and property persistence.

The chapter proceeds as follows: after some more detailed background material on inductive reasoning in the situation calculus in Section 6.1, we formally define the class of property persistence queries in Section 6.2, along with several examples of practical queries that are of this form. Section 6.3 defines the persistence condition operator and demonstrates that it is equivalent to the result of a meta-level fixpoint calculation. Section 6.4 presents a simple iterative algorithm for calculating the persistence condition, and discusses its correctness, completeness, and effectiveness. We conclude with some general discussion in Section 6.5.

6.1 Background

While there is a rich and diverse literature base for the situation calculus, there appears to have been little work on reasoning about universally quantified queries. The work of Reiter [89] shows how to handle such queries manually using an appropriate instantiation of the second-order induction axiom, but makes no mention of automating this reasoning.

Other work considering queries that universally quantify over situations focuses exclusively on verifying state constraints. These are uniform formulae that must hold in every possible situation, a highly specialised form of the more general persistence queries we define in this chapter. The work of Lin and Reiter [66] shows that the induction axiom can be "compiled away" when verifying a state constraint, by means of the following equivalence:

$$\mathcal{D} \models \phi[S_0] \rightarrow (\forall s : S_0 \leq s \rightarrow \phi[s])$$

$$\textit{iff}$$

$$\mathcal{D}_{una} \models \forall s, a : \phi[s] \wedge \mathcal{R}_{\mathcal{D}}(Poss(a, s)) \rightarrow \mathcal{R}_{\mathcal{D}}(\phi[do(a, s)])$$

The set $\mathcal{D}_{una}$ here performs the same role as our background axioms $\mathcal{D}_{bg}$ but contains only the unique names axioms for actions. Verification of a state constraint can thus be reduced to reasoning about a universally quantified uniform formula using only the static background theory, a comparatively straightforward reasoning task which we call *static domain reasoning*. Verification of state constraints was also approached by Bertossi et al. [11], who develop an automatic constraint verification system using an induction theorem prover.

However, there are many issues that are not addressed by work specific to state constraints. What if we are interested in the future of some arbitrary situation σ, rather than only S_0? What if want to restrict future actions according to an arbitrary action description predicate? Can we integrate a method for handling

universally-quantified queries with existing regression techniques? Our treatment of property persistence can provide a concrete basis for these considerations, and is hence significantly more general than this existing work.

Another field that deals with induction over situations is the verification of ConGolog programs. De Giacomo et al. [18] show how to formulate various safety, liveness, and starvation properties of a ConGolog program as fixpoint queries in second-order logic. A preliminary model-checker capable of verifying these properties is described in [42]. Claßen and Lakemeyer [14] develop a logic of ConGolog programs in $\mathcal{ES}$, a variant of the situation calculus based on modal logic. They demonstrate that properties of a program can be verified using an iterative fixpoint computation similar to the one we propose in this chapter.

As we shall see, property persistence queries are equivalent to a particular kind of safety property of a ConGolog program, so our work is in some ways less general than that described above. This means, however, that we can be more specific in our algorithm and approach. These ConGolog verifiers are designed to operate in isolation, while we seek a method of handling universally-quantified queries that can integrate directly with the existing meta-theoretical reasoning machinery of the situation calculus, in particular with the regression operator.

Finally, let us introduce an important property of situations first formally identified by Savelli [96]: that universal quantification over situation terms is equivalent to an infinite conjunction over the *levels* of the tree of situations:

$$\mathcal{D} \models \forall s : \psi(s)$$

$$\text{iff}$$

$$\mathcal{D} \models \bigwedge_{n \in \mathbb{N}} \forall a_1, \ldots, a_n : \psi(do([a_1, \ldots, a_n], S_0))$$

This is a direct consequence of the induction axiom for situations, which restricts situations to be constructed by performing some countable number of actions in the initial situation. While we do not use this result directly in this chapter, it captures an important intuition about situation terms that is fundamental to the operation of our approach.

6.2 Property Persistence Queries

Let us now formally define the kinds of query that will be approached in this chapter. Given some property ϕ and situation σ, a *property persistence query* asks whether

ϕ will hold in all situations in the future of σ:

$$\mathcal{D} \models \forall s : \sigma \sqsubseteq s \rightarrow \phi[s]$$

More generally, one may wish to limit the futures under consideration to those brought about by actions satisfying a certain action description predicate α, which is easily accomplished using the $\leq_\alpha$ macro. We thus have the following definition of a persistence query:

Definition 18 (Property Persistence Query)**.** *Let ϕ be a uniform formula, α an action description predicate, and σ a situation term. Then a property persistence query is a query of the form:*

$$\mathcal{D} \models \forall s : \sigma \leq_\alpha s \rightarrow \phi[s]$$

In words, a persistence query states that "ϕ holds in σ, and assuming all subsequent actions satisfy α, ϕ will continue to hold". For succinctness we will henceforth describe this as "ϕ persists under α". Queries of this form are involved in many useful reasoning tasks, of which the following are a small selection:

Goal Impossibility: Given a goal G, establish that there is no legal situation in which that goal is achieved:

$$\mathcal{D} \models \forall s : S_0 \leq_{Legal} s \rightarrow \neg G(s)$$

Goal Futility: Given a goal G and situation σ, establish that the goal cannot be achieved in any legal future of σ:

$$\mathcal{D} \models \forall s : \sigma \leq_{Legal} s \rightarrow \neg G(s)$$

Note how this differs from goal impossibility: while the agent may have initially been able to achieve its goal, the actions that have subsequently been performed have rendered the goal unachievable. Agents would be well advised to avoid such situations.

Checking State Constraints: Given a state constraint SC, show that the constraint holds in every legal situation:

$$\mathcal{D} \models \forall s : S_0 \leq_{Legal} s \rightarrow SC(s)$$

This can be seen as a variant of goal impossibility, by showing that the constraint can never be violated.

Need for Cooperation: Given an agent agt, goal G and situation σ, establish that no sequence of actions performed by that agent can achieve the goal. Suppose we define $MyAction$ to identify the agent's own actions:

$$MyAction(a, s) \stackrel{\text{def}}{=} actor(a) = agt$$

Then the appropriate query is:

$$\mathcal{D} \models \forall s : \sigma \leq_{MyAction} s \rightarrow \neg G(s)$$

If this is the case, the agent will need to seek cooperation from another agent in order to achieve its goal.

Knowledge with Hidden Actions: An agent reasoning about its own knowledge in asynchronous domains must account for arbitrarily-long sequences of hidden actions. To establish that it knows ϕ, it must establish that ϕ cannot become false through a sequence of hidden actions:

$$\mathcal{D} \models \forall s : \sigma \leq_{Hidden} s \rightarrow \phi[s]$$

This last case is our main motivation for the developments in this chapter, and we will explore the use of property persistence in this context in detail in Chapter 7. The other examples are designed to show that persistence queries are quite a general form of query, and the techniques developed in this chapter thus have application beyond our specific use of them in the remainder of this thesis.

Unfortunately, persistence queries do not meet the criteria for regressable formulae found in Definition 5, since they quantify over situation terms. Such queries therefore cannot be handled using the standard regression operator. Indeed, since universal quantification over situation terms requires the use of a second order induction axiom, current systems needing to answer such queries must resort to second-order theorem proving. This is hardly an attractive prospect for effective automated reasoning.

6.3 The Persistence Condition

To implement practical systems that can perform persistence queries, we clearly need to transform the query into a form suitable for effective automated reasoning. Our approach is to transform a property persistence query at σ into the evaluation of a uniform formula at σ. This transformed query can then be handled effectively using the standard regression operator.

To achieve this we need some transformation of a property ϕ and action description predicate α into a uniform formula $\mathcal{P}_{\mathcal{D}}(\phi, \alpha)$ that is true at precisely the situations in which ϕ persists under α. We call such a formula the *persistence condition* of ϕ under α.

Definition 19 (Persistence Condition)**.** *The persistence condition of ϕ under α, denoted $\mathcal{P}_{\mathcal{D}}(\phi, \alpha)$, is a uniform formula that is equivalent to the persistence of ϕ under α with respect to a basic action theory $\mathcal{D}$ without the initial situation axioms. Formally:*

$$\mathcal{D} - \mathcal{D}_{S_0} \models \forall s : \left(\mathcal{P}_{\mathcal{D}}(\phi, \alpha)[s] \equiv \forall s' : s \leq_\alpha s' \rightarrow \phi[s']\right)$$

Defining $\mathcal{P}_{\mathcal{D}}$ to be independent of the initial world state allows an agent to calculate it regardless of what (if anything) is known about the actual state of the world – after all, an agent may not know all the details of $\mathcal{D}_{S_0}$, and we still want it to be able to use this technique.

This definition alone clearly does not make the task of answering a persistence query any easier, since it gives no indication of how the persistence condition might be calculated in practice. Indeed, we have not yet even shown whether such a formula actually exists. In order to establish these results, we first need to define the weaker notion of a formula *persisting to depth n* in a situation.

Since we wish to establish our technique as a general reasoning mechanism for the situation calculus, we drop the assumption that concurrent actions are in use for the duration of this chapter. Note that nothing in our definitions precludes the use of various situation calculus extensions as described in Section 2.1.4.

Definition 20 (Persistence to Depth 1)**.** *A uniform formula ϕ persists to depth 1 under α in situation s when the formula $\mathcal{P}^1_{\mathcal{D}}(\phi, \alpha)[s]$ holds, as defined by:*

$$\mathcal{P}^1_{\mathcal{D}}(\phi, \alpha) \stackrel{def}{=} \phi^{-1} \wedge \forall a : \mathcal{R}_{\mathcal{D}}(\alpha[a, s])^{-1} \rightarrow \mathcal{R}_{\mathcal{D}}(\phi[do(a, s)])^{-1}$$

Note that $\mathcal{P}^1_{\mathcal{D}}$ is a literal encoding of the requirement "ϕ holds in s and in all its direct successors", using the standard regression operator $\mathcal{R}_{\mathcal{D}}$ and the situation-suppression operator ϕ^{-1} to produce a situation-suppressed uniform formula. With-

out the use of regression, the definition would appear as follows:

$$\mathcal{P}^1_{\mathcal{D}}(\phi, \alpha)[s] \equiv \phi[s] \wedge \forall a : \alpha[a, s] \rightarrow \phi[do(a, s)]$$

Since α is an action description predicate and ϕ is a uniform formula, the expressions $\mathcal{R}_{\mathcal{D}}(\alpha[a, s])^{-1}$ and $\mathcal{R}(\phi[do(a, s)])^{-1}$ are always defined and always produce uniform formulae. Successive applications of $\mathcal{P}^1_{\mathcal{D}}$ can then assert persistence to greater depths:

Definition 21 (Persistence to Depth N)**.** *For any $n \geq 0$, a uniform formula ϕ persists to depth n under α in situation s when the formula $\mathcal{P}^n_{\mathcal{D}}(\phi, \alpha)[s]$ holds, as defined by:*

$$\mathcal{P}^0_{\mathcal{D}}(\phi, \alpha) \stackrel{def}{=} \phi$$
$$\mathcal{P}^n_{\mathcal{D}}(\phi, \alpha) \stackrel{def}{=} \mathcal{P}^1_{\mathcal{D}}(\mathcal{P}^{n-1}_{D}(\phi, \alpha), \alpha)$$

The following theorem confirms that $\mathcal{P}^n_{\mathcal{D}}$ operates according to this intuition – that for any sequence of actions of length $i = 0$ to $i = n$, if each action satisfies α in the situation it is executed in, then ϕ will hold after executing those actions.

Theorem 5. *For any $n \in \mathbb{N}$, $\mathcal{P}^n_{\mathcal{D}}(\phi, \alpha)$ holds in σ iff ϕ holds in σ and in all successors of σ reached by performing at most n actions satisfying α:*

$$\mathcal{D} \models \mathcal{P}^n_{\mathcal{D}}(\phi, \alpha)[\sigma] \equiv$$
$$\bigwedge_{i \leq n} \forall a_1, \ldots, a_i : \left(\bigwedge_{j \leq i} \alpha[a_j, do([a_1, \ldots, a_{j-1}], \sigma)] \rightarrow \phi[do([a_1, \ldots, a_i], \sigma)] \right)$$

Proof Sketch. By induction on the natural numbers. For $n = 0$ we have $\phi[\sigma] \equiv \phi[\sigma]$ by definition. For the inductive case, we expand the definition of $\mathcal{P}^n_{\mathcal{D}}(\phi, \alpha)[\sigma]$ to get the following for the LHS:

$$\mathcal{P}^{n-1}_{\mathcal{D}}(\phi, \alpha)[\sigma] \wedge \forall a : \mathcal{R}_{\mathcal{D}}(\alpha[a, \sigma]) \rightarrow \mathcal{R}_{\mathcal{D}}(\mathcal{P}^{n-1}_{\mathcal{D}}(\phi, \alpha)[do(a, \sigma)])$$

Substituting for $\mathcal{P}^{n-1}_{\mathcal{D}}$ using the inductive hypothesis gives us a conjunction ranging over $i \leq n - 1$, with universally quantified variables $a_1, \ldots, a_i$, and we must establish the $i = n$ case. Pushing this conjunction inside the scope of the $\forall a$ quantifier, we find we can rename $a \Rightarrow a_1$, $a_1 \Rightarrow a_2$ etc to get the required expression. For a detailed proof see Appendix A. □

The $\mathcal{P}^n_{\mathcal{D}}$ operator thus allows us to express the persistence of a formula to any

given depth using a simple syntactic translation based on regression. Intuitively, one would expect $\mathcal{P}_{\mathcal{D}}(\phi, \alpha)$ to be some sort of fixpoint of $\mathcal{P}^1_{\mathcal{D}}(\phi, \alpha)$, since $\mathcal{P}_{\mathcal{D}}(\phi, \alpha)$ must imply persistence up to any depth. Such a fixpoint could then be calculated using standard iterative approximation techniques. The remainder of this section is devoted to verifying this intuition.

We begin by adapting two existing results involving induction from the situation calculus literature, so that they operate with our generalised $\leq_\alpha$ notation and can be based at situations other than S_0.

Proposition 4. *For any action description predicate α, the foundational axioms of the situation calculus entail the following induction principle:*

$$\forall W, s:\ W(s) \wedge \left[\forall a, s':\ \alpha[a, s'] \wedge s \leq_\alpha s' \wedge W(s') \rightarrow W(do(a, s'))\right] \\ \rightarrow \forall s':\ s \leq_\alpha s' \rightarrow W(s')$$

Proof. A trivial adaptation of Theorem 1 in [89]. □

Proposition 5. *For any basic action theory $\mathcal{D}$, uniform formula ϕ and action description predicate α:*

$$\mathcal{D} - \mathcal{D}_{S_0} \models \forall s:\ \phi[s] \rightarrow \left(\forall s':\ s \leq_\alpha s' \rightarrow \phi[s']\right)$$
$$\text{iff}$$
$$\mathcal{D}_{bg} \models \forall s, a:\ \phi[s] \wedge \mathcal{R}_{\mathcal{D}}(\alpha[a, s]) \rightarrow \mathcal{R}_{\mathcal{D}}(\phi[do(a, s)])$$

Proof. A straightforward generalisation of the model-construction proof of Lemma 5 in [66], utilising Proposition 4. □

Proposition 5 will be key in our algorithm for calculating the persistence condition. It allows one to establish the result "if ϕ holds in s, then ϕ persists in s" by using static domain reasoning, a comparatively straightforward reasoning task.

We next formalise some basic relationships between $\mathcal{P}_{\mathcal{D}}$ and $\mathcal{P}^n_{\mathcal{D}}$.

Lemma 2. *Given a basic action theory $\mathcal{D}$, uniform formula ϕ and action description predicate α, then for any n:*

$$\mathcal{D} - \mathcal{D}_{S_0} \models \forall s:\ \left(\forall s':\ s \leq_\alpha s' \rightarrow \phi[s']\right) \equiv \left(\forall s':\ s \leq_\alpha s' \rightarrow \mathcal{P}^n_D(\phi, \alpha)[s']\right)$$

That is, ϕ persists under α iff $\mathcal{P}^n_{\mathcal{D}}[\phi, \alpha]$ persists under α.

Proof. Since $\mathcal{P}^n_{\mathcal{D}}[\phi, \alpha]$ implies ϕ by definition, the *if* direction is trivial. For the *only-if* direction we proceed by induction on n.

For the base case, assume that ϕ persists but $\mathcal{P}^1_{\mathcal{D}}(\phi,\alpha)$ does not, then we must have some s' with $s \leq_\alpha s'$ and $\neg\mathcal{P}^1_{\mathcal{D}}(\phi,\alpha)[s']$. By the definition of $\mathcal{P}^1_{\mathcal{D}}$, this means that:

$$\neg\left(\phi[s'] \wedge \forall a : \alpha[a,s'] \rightarrow \phi[do(a,s')]\right)$$

Since ϕ persists it must hold at s', so there must be some a such that $\alpha[a,s']$ and $\neg\phi[do(a,s')]$. But $s \leq_\alpha do(a,s')$ and so $\phi[do(a,s')]$ must hold by our assumption that ϕ persists, and we have a contradiction.

For the inductive case, assume that $\mathcal{P}^{n-1}_{\mathcal{D}}(\phi,\alpha)$ persists but $\mathcal{P}^n_{\mathcal{D}}(\phi,\alpha)$ does not. By definition we have $\mathcal{P}^n_{\mathcal{D}}(\phi,\alpha) = \mathcal{P}^1_{\mathcal{D}}(\mathcal{P}^{n-1}_{\mathcal{D}}(\phi,\alpha),\phi)$, and we repeat the base case proof using $\phi' = \mathcal{P}^{n-1}_{\mathcal{D}}(\phi,\alpha)$ in place of ϕ to obtain a contradiction. □

Lemma 3. *Given a basic action theory $\mathcal{D}$, uniform formula ϕ and action description predicate α, then for any n:*

$$\mathcal{D} - \mathcal{D}_{S_0} \models \forall s : (\mathcal{P}_{\mathcal{D}}(\phi,\alpha)[s] \rightarrow \mathcal{P}^n_{\mathcal{D}}(\phi,\alpha)[s])$$

Proof. $\mathcal{P}_{\mathcal{D}}(\phi,\alpha)$ implies the persistence of ϕ by definition. If ϕ persists at s, then by Lemma 2 we have that $\mathcal{P}^n_{\mathcal{D}}(\phi,\alpha)$ persists at s . Since the persistence of $\mathcal{P}^n_{\mathcal{D}}(\phi,\alpha)$ at s implies that $\mathcal{P}^n_{\mathcal{D}}(\phi,\alpha)$ holds at s by definition, we have the lemma as desired. □

We are now equipped to prove the major theorem of this chapter: that if $\mathcal{P}^n_{\mathcal{D}}(\phi,\alpha)$ implies $\mathcal{P}^{n+1}_{\mathcal{D}}(\phi,\alpha)$, then $\mathcal{P}^n_{\mathcal{D}}(\phi,\alpha)$ is the persistence condition for ϕ under α.

Theorem 6. *Given a basic action theory $\mathcal{D}$, uniform formula ϕ and action description predicate α, then for any n:*

$$\mathcal{D}_{bg} \models \forall s : \mathcal{P}^n_{\mathcal{D}}(\phi,\alpha)[s] \rightarrow \mathcal{P}^{n+1}_{\mathcal{D}}(\phi,\alpha)[s] \tag{6.1}$$

iff

$$\mathcal{D} - \mathcal{D}_{s_0} \models \forall s : \mathcal{P}^n_{\mathcal{D}}(\phi,\alpha)[s] \equiv \mathcal{P}_{\mathcal{D}}(\phi,\alpha)[s] \tag{6.2}$$

Proof. For the *if* direction, we begin by expanding equation (6.1) using the definition of $\mathcal{P}^1_{\mathcal{D}}$ to get the equivalent form:

$$\mathcal{D}_{bg} \models \forall s : \mathcal{P}^n_{\mathcal{D}}(\phi,\alpha)[s] \rightarrow \mathcal{P}^1_{\mathcal{D}}(\mathcal{P}^n_{\mathcal{D}}(\phi,\alpha),\alpha)[s]$$
$$\mathcal{D}_{bg} \models \forall s : \mathcal{P}^n_{\mathcal{D}}(\phi,\alpha)[s] \rightarrow (\mathcal{P}^n_{\mathcal{D}}(\phi,\alpha)[s] \wedge \forall a : \mathcal{R}_{\mathcal{D}}(\alpha[a,s]) \rightarrow \mathcal{R}_{\mathcal{D}}(\phi[do(a,s)]))$$
$$\mathcal{D}_{bg} \models \forall s,a : \mathcal{P}^n_{\mathcal{D}}(\phi,\alpha)[s] \wedge \forall a : \mathcal{R}_{\mathcal{D}}(\alpha[a,s]) \rightarrow \mathcal{R}_{\mathcal{D}}(\phi[do(a,s)])$$

By Proposition 5, equation (6.1) thus lets us conclude that $\mathcal{P}^n_{\mathcal{D}}(\phi,\alpha)$ persists under α. By Lemma 2 this is equivalent to the persistence of ϕ under α, which is equivalent

to $\mathcal{P}_{\mathcal{D}}(\phi, \alpha)$ by definition, giving:

$$\mathcal{D} - \mathcal{D}_{s_0} \models \forall s : \mathcal{P}^n_{\mathcal{D}}(\phi, \alpha)[s] \rightarrow \mathcal{P}_{\mathcal{D}}(\phi, \alpha)[s]$$

By Lemma 3 this implication is an equivalence, yielding equation (6.2) as required.

The *only if* direction is a straightforward reversal of this reasoning: $\mathcal{P}_{\mathcal{D}}(\phi, \alpha)$ implies the persistence of ϕ, which implies the persistence of $\mathcal{P}^n_{\mathcal{D}}(\phi, \alpha)$, which yields equation (6.1) by Proposition 5. □

Since $\mathcal{D}_{bg} \models \mathcal{P}^{n+1}_{\mathcal{D}}(\phi, \alpha) \rightarrow \mathcal{P}^n_{\mathcal{D}}(\phi, \alpha)$ by definition, equation (6.1) identifies $\mathcal{P}^n_{\mathcal{D}}(\phi, \alpha)$ as a fixpoint of the $\mathcal{P}^1_{\mathcal{D}}$ operator, as our initial intuition suggested. In fact, we can use the constructive proof of Tarski's fixpoint theorem [15] to establish that the persistence condition always exists for a given ϕ and α.

Theorem 7. *Given a uniform formula ϕ and action description predicate α, the persistence condition $\mathcal{P}_{\mathcal{D}}(\phi, \alpha)$ always exists, and is unique up to equivalence under the static background theory $\mathcal{D}_{bg}$.*

Proof. Let L be the subset of the Lindenbaum algebra of the static background theory $\mathcal{D}_{bg}$ containing only sentences uniform in s. L is thus a boolean lattice in which each element is a set of sentences uniform in s that are equivalent under $\mathcal{D}_{bg}$. L is a complete lattice with minimal element the equivalence class of $\bot$ and maximal element the equivalence class of $\top$. Fixing α, $\mathcal{P}^1_{\mathcal{D}}$ is a function whose domain and range are the elements of L.

By definition, we have that $\mathcal{P}^1_{\mathcal{D}}(\phi, \alpha) \rightarrow \phi$, and $\mathcal{P}^1_{\mathcal{D}}$ is thus a *monotone decreasing* function over L. By the constructive proof of Tarski's fixpoint theorem, $\mathcal{P}^1_{\mathcal{D}}$ must have a unique greatest fixpoint less than the equivalence class of ϕ, which can be determined by transfinite iteration of the application of $\mathcal{P}^1_{\mathcal{D}}$. By Theorem 6, this fixpoint is the equivalence class of $\mathcal{P}_{\mathcal{D}}(\phi, \alpha)$ under $\mathcal{D}_{bg}$. □

This theorem legitimates the use of the persistence condition for reasoning about property persistence queries – for any persistence query at situation σ, there is a unique (up to equivalence) corresponding query that is uniform in σ and is thus amenable to standard effective reasoning techniques of the situation calculus.

Of course, it remains to actually calculate the persistence condition for a given ϕ and α. The definition of $\mathcal{P}_{\mathcal{D}}(\phi, \alpha)$ as a fixpoint suggests that it can be calculated by iterative approximation, which we discuss in the next section.

Algorithm 6 Calculate $\mathcal{P}_\mathcal{D}(\phi, \alpha)$

```
pn ⇐ φ
pn1 ⇐ P¹_D(pn, α)
while D_bg ⊭ ∀s : pn[s] → pn1[s] do
  pn ⇐ pn1
  pn1 ⇐ P¹_D[pn, α]
end while
return pn
```

6.4 Calculating $\mathcal{P}_\mathcal{D}$

Since we can easily calculate $\mathcal{P}^n_\mathcal{D}(\phi, \alpha)$ for any n, we have a straightforward algorithm for determining $\mathcal{P}_\mathcal{D}(\phi, \alpha)$: search for an n such that

$$\mathcal{D}_{bg} \models \forall s : \left(\mathcal{P}^n_\mathcal{D}(\phi, \alpha)[s] \rightarrow \mathcal{P}^{n+1}_\mathcal{D}(\phi, \alpha)[s]\right)$$

Since we expect $\mathcal{P}^n_\mathcal{D}(\phi, \alpha)$ to be simpler than $\mathcal{P}^{n+1}_\mathcal{D}(\phi, \alpha)$, we should look for the smallest such n. Algorithm 6 presents an iterative procedure for doing just that.

Note that the calculation of $\mathcal{P}^1_\mathcal{D}(\phi, \alpha)$ is a straightforward syntactic transformation, so we do not present an algorithm for it.

6.4.1 Correctness

If Algorithm 6 terminates, it terminates returning a value of pn for which equation (6.1) holds. By Theorem 6 this value of pn is equivalent to the persistence condition for ϕ under α. The algorithm therefore correctly calculates the persistence condition.

In particular, note that equation (6.1) holds when $\mathcal{P}^n_\mathcal{D}(\phi, \alpha)$ is unsatisfiable for any situation, as it appears in the antecedent of an implication. The algorithm thus correctly returns an unsatisfiable condition (equivalent to $\bot$) when ϕ can never persist under α.

6.4.2 Completeness

Since Theorem 6 is an equivalence, the persistence condition is always the fixpoint of $\mathcal{P}^1_\mathcal{D}$. From Theorem 7 this fixpoint always exists and can be calculated by transfinite iteration. Therefore, the only source of incompleteness in our algorithm will be failure to terminate. Algorithm 6 may fail to terminate for two reasons: the loop condition may never be satisfied, or the first-order logical inference in the loop condition may be undecidable and fail to terminate.

The later case indicates that the background theory $\mathcal{D}_{bg}$ is undecidable. While

this is a concern, it affects more than just our algorithm – any system implemented around such an action theory will be incomplete. With respect to this source of incompleteness, our algorithm is no more incomplete than any larger reasoning system it would form a part of.

The former case is of more direct consequence to our work. Unfortunately, there is no guarantee in general that the fixpoint can be reached via *finite* iteration, which is required for termination of Algorithm 6.

Indeed, it is straightforward to construct a fluent for which the algorithm never terminates: consider a fluent $F(x, s)$ that is affected by a single action that makes it false whenever $F(suc(x), s)$ is false. Letting α be vacuously true, the sequence of iterations produced by our algorithm would be:

$$\begin{gathered}
\mathcal{P}^1_{\mathcal{D}}(F(x,s)) \equiv F(x,s) \wedge F(suc(x),s) \\
\mathcal{P}^2_{\mathcal{D}}(F(x,s)) \equiv F(x,s) \wedge F(suc(x),s) \wedge F(suc(suc(x)),s) \\
\vdots \\
\mathcal{P}^n_{\mathcal{D}}(F(x,s)) \equiv \bigwedge_{i=0}^{i=n} F(suc^i(x),s)
\end{gathered}$$

The persistence condition in this case is clearly:

$$\mathcal{P}_{\mathcal{D}}(F(x,s)) \equiv \forall y : x \leq y \rightarrow F(y,s)$$

While this is equivalent to the infinite conjunction produced as the limit of iteration in our algorithm, it will not be found after any finite number of steps.

As discussed in the proof of Theorem 7, $\mathcal{P}^1_{\mathcal{D}}$ operates over the boolean lattice of equivalence classes of formulae uniform in s, and the theory of fixpoints requires that this lattice be *well-founded* to guarantee termination of an iterative approximation algorithm such as Algorithm 6. We must therefore identify restricted kinds of basic action theory for which this well-foundedness can be guaranteed.

The most obvious case is theories in which the action and object sorts are finite. In such theories the lattice of equivalence classes of formulae uniform in s is finite, and any finite lattice is well-founded. These theories also have the advantage that the static domain reasoning performed by Algorithm 6 can be done using propositional logic, meaning it is decidable and so providing a strong termination guarantee.

Alternately, suppose all successor state axioms and action description predicates have the following restricted form, where the terms in $\bar{y}$ are a subset of the terms in $\bar{x}$ and Φ_F, Π_{ADP} mention no terms other than $\bar{x}$, a and s:

$$F(\bar{x}, do(a, s)) \equiv \bigvee_{i=1}^{n} a = a_i(\bar{y}_i) \wedge \Phi_F(\bar{x}, a, s)$$
$$ADP(\bar{x}, a, s) \equiv \bigvee_{i=1}^{n} a = a_i(\bar{y}) \wedge \Pi_{ADP}(\bar{x}, a, s)$$

Under such theories, applications of $\mathcal{P}^1_\mathcal{D}$ will introduce no new terms into the query, apart from finitely many action terms a_i. The range of $\mathcal{P}^1_\mathcal{D}$ applied to ϕ is then a finite subset of the lattice of equivalence classes of formulae uniform in s, again guaranteed well-foundedness and terminating calculation of $\mathcal{P}_\mathcal{D}$.

Of course this is a very strong restriction on the structure of the theory, as the successor state axioms are not able to contain any quantifiers. It does demonstrate, however, that certain syntactical restrictions on $\mathcal{D}$ are able to guarantee terminating calculation of $\mathcal{P}_\mathcal{D}$. It seems there should be a more general "syntactic well-foundedness" restriction that can be applied to these axioms, but we have not successfully formulated one at this stage.

In a similar vein, suppose that the theory of action is *context free* [67]. In such theories successor state axioms have the following form:

$$F(\bar{x}, do(a, s)) \equiv \Phi^+_F(\bar{x}, a) \vee \big(F(\bar{x}, s) \wedge \neg\Phi^-_F(\bar{x}, a)\big)$$

The effects of an action are thus independent of the situation it is performed in. Lin and Levesque [64] demonstrate that such theories have a finite state space, again ensuring our algorithm operates over a finite lattice and hence guaranteeing termination. Context free domains are surprisingly expressive; for example, domains described in the style of STRIPS operators are context free.

From a slightly different perspective, suppose that ϕ can never persist under α, so that $\mathcal{P}_\mathcal{D}(\phi, \alpha) \equiv \bot$. Further suppose that $\mathcal{D}$ has the *compactness* property as in standard first-order logic. Then the "quantum levels" of Savelli [96] guarantee that there is a fixed, finite number of actions within which $\neg\phi$ can always be achieved. In this case Algorithm 6 will determine $\mathcal{P}_\mathcal{D}(\phi, \alpha) \equiv \bot$ within finitely many iterations.

It would also be interesting to determine whether known decidable variants of the situation calculus (such as [38]) are able to guarantee termination of the fixpoint construction, or whether more sophisticated fixpoint algorithms can be applied instead of simple iterative approximation. Investigating such algorithms would be a promising avenue for future research.

The important point here is not that we can guarantee completeness in general,

but that we have precisely characterised the inductive reasoning necessary to answer property persistence queries, and shown that it can always be replaced by the evaluation of a uniform formula at the situation in question.

6.4.3 Effectiveness

Our algorithm replaces a single reasoning task based on the full action theory $\mathcal{D}$ with a series of reasoning tasks based on the static background theory $\mathcal{D}_{bg}$. Is this a worthwhile trade-off in practice? The following points weigh strongly in favour of our approach.

First and foremost, we avoid the need for the second-order induction axiom. All the reasoning tasks can be performed using standard first-order reasoning, for which there are high-quality automated provers. Second, the calculation of $\mathcal{P}_{\mathcal{D}}$ performs only *static doing reasoning*, which as discussed in Chapter 2 is a comparatively straightforward task which can be made decidable under certain conditions. Third, $\mathcal{P}_{\mathcal{D}}(\phi, \alpha)[s]$ is in a form amenable to regression, a standard tool for effective reasoning in the situation calculus. Fourth, the persistence condition for a given ϕ and α can be cached and re-used for a series of related queries about different situations, a significant gain in amortised efficiency. Finally, in realistic domains we expect many properties to fail to persist beyond a few situations into the future, meaning that our algorithm will require few iterations in a large number of cases.

Of course, we also inherit the potential disadvantage of the regression operator: the length of $\mathcal{P}_{\mathcal{D}}(\phi, \alpha)$ may be exponential in the length of ϕ. As with regression, our experience has been that this is rarely a problem in practice, and is more than compensated for by the reduced complexity of the resulting reasoning task.

6.4.4 Applications

The persistence condition is readily applicable to the example persistence query problems given in Section 6.2. All of the transformed queries can then be answered using standard regression.

Goal Impossibility: Given a goal G, establish that there is no legal situation in which that goal is satisfied:

$$\mathcal{D} \models \mathcal{P}_{\mathcal{D}}(\neg G, Legal)[S_0]$$

The persistence condition of $\neg G$ with respect to action legality allows goal impossibility to be checked easily.

Goal Futility: Given a goal G and situation σ, establish that the goal cannot be satisfied in any legal future situation from σ:

$$\mathcal{D} \models \mathcal{P}_{\mathcal{D}}(\neg G, Legal)[\sigma]$$

Precisely the same formula is required for checking goal impossibility and goal futility. This highlights the advantage of re-using the persistence condition at multiple situations. Our approach makes it feasible for an agent to check for goal futility each time it considers performing an action, and avoid situations that would make its goals unachievable.

Checking State Constraints: Given a state constraint SC, show that the constraint holds in every legal situation:

$$\mathcal{D} \models \mathcal{P}_{\mathcal{D}}(SC, Legal)[S_0]$$

However, since we want a state constraint to *always* persist, it must satisfy the following equivalence:

$$\mathcal{D}_{bg} \models \phi \equiv \mathcal{P}_{\mathcal{D}}(\phi, Legal)$$

If this equivalence does not hold then $\mathcal{P}_{\mathcal{D}}(\phi, Legal)$ indicates the additional conditions that are necessary to ensure that ϕ persists, which may be used to adjust the action theory to enforce the constraint. This particular application has strong parallels to the approach to state constraints developed by Lin and Reiter [66].

Need for Cooperation: Given an agent agt, goal G and situation σ, establish that no sequence of actions performed by that agent can achieve the goal:

$$\mathcal{D} \models \mathcal{P}_{\mathcal{D}}(\neg G, MyAction)[\sigma]$$

Knowledge with Hidden Actions: In Chapter 7 we will develop a regression rule for knowledge that uses the persistence condition to account for arbitrarily-long sequences of hidden actions. While we defer the details to that chapter, the general form of the rule is:

$$\mathcal{R}_{\mathcal{D}}(\mathbf{Knows}(\phi, do(a, s))) \stackrel{\text{def}}{=} \mathbf{Knows}(\mathcal{R}_{\mathcal{D}}(\mathcal{P}_{\mathcal{D}}(\phi, Hidden), a), s)$$

6.5 Discussion

In this chapter we have developed an algorithm that transforms property persistence queries, a very general and useful class of situation calculus query, to a form that is amenable to standard techniques for effective reasoning in the situation calculus. The algorithm replaces a second-order induction axiom with a meta-level fixpoint calculation based on iterative application of the standard regression operator. It is shown to be correct, and also complete in some interesting cases.

Our approach generalises previous work on universally-quantified queries in several important ways. It can consider sequences of actions satisfying a range of conditions, not just the standard ordering over action possibility, enabling us to treat problems such as need for cooperation and knowledge with hidden actions. It can establish that properties persist in the future of an arbitrary situation, not necessarily the initial situation, enabling us to answer the question of goal futility. The results of calculating the persistence condition can be cached, allowing for example the goal futility question to be efficiently posed on a large number of situations once the persistence condition has been calculated.

Most importantly for the remainder of this thesis, we have *factored out* the inductive reasoning required to answer these queries. Work on increasing the effectiveness of this inductive reasoning, and on guaranteeing a terminating calculation in stronger classes of action theory, can now proceed independently from the development of formalisms that utilise persistence queries. We will henceforth use $\mathcal{P}_{\mathcal{D}}$ as a kind of "black box" operator to formulate regression rules within our framework, dropping the explicit $\mathcal{D}$ subscript as we do for the regression operator.

As noted in Section 6.1, our use of fixpoints in this chapter has much in common with the study of properties of ConGolog programs by [14, 18]. Indeed, a property persistence query is equivalent to a safety query stating that the property ϕ never becomes false during execution of the following program:

$$\delta_{P\alpha} \stackrel{\text{def}}{=} (\pi(a,\, \alpha[a]?\,;\, a))^*$$

Formally:

$$\mathcal{D} \models \forall s : \sigma \leq_\alpha s \rightarrow \phi[s]$$

iff

$$\mathcal{D} \cup \mathcal{D}_{golog} \models \forall s, \delta : Trans^*(\delta_{P\alpha}, \sigma, \delta, s) \rightarrow \phi[s]$$

Since we intend to use persistence queries as part of a larger reasoning apparatus, rather than as a stand-alone query, we cannot directly leverage the existing work on verifying ConGolog programs. However, given the similarity between the approaches, we are confident that advances in reasoning effectively about ConGolog programs will also advance our ability to effectively answer persistence queries.

This chapter has thus significantly increased the scope of queries that can be posed when building systems upon the situation calculus. In the coming chapters, the persistence condition operator will allow us to factor out certain inductive aspects of reasoning, treating them as separate, well-defined components of the overall reasoning machinery.

Chapter 7

Knowledge with Hidden Actions

This chapter develops a new theory of knowledge in the situation calculus that directly leverages the agent-local perspective developed in Chapter 4.

Existing accounts of epistemic reasoning in the situation calculus require that whenever an action occurs, all agents *know* that an action has occurred. This demands a level of synchronicity that is unreasonable in many multi-agent domains. In asynchronous domains, each agent's knowledge must instead account for arbitrarily-long sequences of *hidden actions*. This requires a second-order induction axiom which precludes the use of regression for effective automated reasoning. It also make it difficult for agents to reason about their own knowledge, as they may not have enough information to formulate an appropriate query.

To overcome this limitation we combine two of the contributions developed in preceding chapters - the explicit representation of an agent's local perspective from Chapter 4, and the persistence condition meta-operator from Chapter 6 - to formulate an account of knowledge in the situation calculus that can faithfully represent the hidden actions inherent in asynchronous domains while maintaining a regression rule for effective automated reasoning.

We begin by developing an axiomatisation of knowledge based explicitly on each agent's local view. This axiomatisation is shown to respect our intuitions about how knowledge should behave, and to preserve important properties of the agent's epistemic state through the occurrence of actions. Moreover, our formulation is *elaboration tolerant*, automatically preserving these properties in the face of more complex information-producing actions, such as guarded sensing actions, that can easily invalidate these properties if not axiomatised in the appropriate way.

To formulate a regression rule for knowledge, we appeal to the persistence condition operator to factor out the inductive reasoning required for dealing with hidden

actions. We propose a new regression rule that is sound and complete with respect to our axiomatisation of knowledge, and demonstrate how it can be used by a situated agent to reason about its own knowledge in a straightforward manner.

The end result is a significantly more general and robust theory of knowledge in the situation calculus that still permits an effective reasoning procedure.

The chapter proceeds as follows: after introducing more detailed background material on epistemic reasoning in the situation calculus in Section 7.1, we develop the axioms for our new observation-based account of knowledge in Section 7.2. Section 7.3 explores the properties of our formalism with reference to the standard account of knowledge. Section 7.4 develops a regression rule for our formalism using the persistence condition operator, while Section 7.5 gives a brief example of its use for reasoning about a partially-observable domain. Section 7.6 explores some avenues for combining our formalism with other recent developments in the situation calculus literature, while Section 7.7 concludes with a general discussion and summary.

7.1 Background

Recall from Section 2.3 that the dynamics of knowledge in the situation calculus are specified using an additional set of axioms $\mathcal{D}_K$, which define the behaviour of a special knowledge-fluent K. In the standard account of knowledge, based on the work of Scherl and Levesque [98] and incorporating concurrent actions and multiple agents [97, 106], the axioms in $\mathcal{D}_K$ are the following:

$$Init(s) \rightarrow \big(K(agt, s', s) \equiv K_0(agt, s', s)\big) \tag{7.1}$$

$$\begin{aligned} K(agt, s'', do(c, s)) \equiv \exists s' : s'' = do(c, s') \wedge K(agt, s', s) \\ \wedge Legal(c, s') \wedge \forall a \in c : \big(actor(a) = agt \rightarrow SR(a, s) = SR(a, s')\big) \end{aligned} \tag{7.2}$$

Equation (7.1) ensures that the agents begin with their knowledge as specified by the initial knowledge fluent K_0. In fact, the work of [97, 98, 106] does not actually use an explicit K_0 fluent and instead specifies initial knowledge directly using K. The introduction of K_0 in this case is purely cosmetic, but it will make comparisons with our new formalism easier.

Equation (7.2) takes the form of a standard successor state axiom for the K fluent. It ensures that s'' is considered a possible alternative to $do(c, s)$ when s'' is the result of doing the same actions c in a situation s' that is considered a possible alternative to s. It must furthermore have been possible to perform those actions in

s', and the sensing results must match for each action that was carried out by the agent in question. Thus an agent's knowledge after the occurrence of an action is completely determined by the combination of its knowledge before the action, and the sensing results from the action.

Definition 22. *We will denote by $\mathcal{D}_K^{std}$ the axioms of the standard account of knowledge due to [97, 98], as detailed in equations (7.1,7.2) above.*

While powerful, this knowledge-representation formalism has an important limitation: it is fundamentally *synchronous.* Each agent is assumed to have full knowledge of all actions that have occurred – in other words, all actions are assumed to be public. While suitable for some domains, there are clearly many multi-agent domains where achieving total awareness of actions would be infeasible. A major contribution of this chapter is a more flexible formalism for knowledge that can be applied to a much wider range of domains.

7.1.1 Reasoning about Knowledge

A key contribution of Scherl and Levesque [98] was showing how to apply the regression operator to formulae containing the **Knows** macro, allowing it to be treated syntactically as if it were a primitive fluent. This means that epistemic queries can be approached using standard reasoning techniques of the situation calculus. Although we have changed the notation somewhat to account for concurrent actions and to foreshadow the techniques we will develop in Section 7.4, their definition operates as follows. First, define the *results* of a concurrent action to be the set of *action#result* pairs for all primitive actions performed by the agent in question:

$$\mathbf{res}(agt, c, s) \stackrel{\text{def}}{=} \{a\#SR(a, s) \mid a \in c \wedge actor(a) = agt\}$$

This definition is then used to formulate a regression rule as follows:

$$\begin{aligned}\mathcal{R}(\mathbf{Knows}(agt, \phi, do(c, s)) \stackrel{\text{def}}{=} \quad & \exists y : y = \mathbf{res}(agt, c, s) \\ & \wedge \mathbf{Knows}(agt, [Legal(c) \wedge \mathbf{res}(agt, c) = y] \rightarrow \mathcal{R}(\phi[do(c, s)]), s) \quad (7.3)\end{aligned}$$

This works by collecting the sensing results from each action performed by the agent into the set y, then ensuring matching sensing results in every situation considered possible. It expresses the knowledge of the agent after a concurrent action in terms of what it knew before the action, along with the information returned by the action. This technique relies heavily on the fact that all actions are public, since it requires every agent's knowledge to be updated in response to every action.

As with the non-epistemic case, repeated applications of $\mathcal{R}$ can transform a knowledge query into one that is uniform in the initial situation. While it would be valid to then expand the **Knows** macros and handle the query using first-order logic, in practice the reasoning procedure would leave **Knows** intact and use a specialised prover based on modal logic.

7.1.2 Accessibility Properties

A fundamental aspect of epistemic reasoning is identifying certain properties of the K relation (in modal logic terminology, the agent's *accessibility* relation) that guarantee certain desired axioms of knowledge. The most commonly-used accessibility restrictions are:

Reflexivity: $K(agt, s, s)$

Transitivity: $K(agt, s_2, s_1) \wedge K(agt, s_3, s_2) \rightarrow K(agt, s_3, s_1)$

Euclidean: $K(agt, s_2, s_1) \wedge K(agt, s_3, s_1) \rightarrow K(agt, s_3, s_2)$

Symmetry: $K(agt, s_2, s_1) \rightarrow K(agt, s_1, s_2)$

The first three of these properties are directly equivalent to asserting three important axioms about the knowledge operator:

Correct Knowledge: $\mathbf{Knows}(agt, \phi) \rightarrow \phi$

Positive Introspection: $\mathbf{Knows}(agt, \phi) \rightarrow \mathbf{Knows}(agt, \mathbf{Knows}(agt, \phi))$

Negative Introspection: $\neg\mathbf{Knows}(agt, \phi) \rightarrow \mathbf{Knows}(agt, \neg\mathbf{Knows}(agt, \phi))$

In particular, the restriction of K to be reflexive ensures that the logic is one of *knowledge* rather than *belief*; the agents cannot know statements that are not true in the real world.

One of the major theorems of [98] is that if one or more of these properties hold for the K relation in the initial situation, then they will hold for the K relation in all future situations. This means that the above axioms of knowledge are preserved through the occurrence of actions, an important confirmation that the semantics of knowledge have been axiomatised correctly; the occurrence of an action should not, for example, remove an agent's ability to introspect its own knowledge.

7.1.3 Extending Knowledge Theories

As we discussed in Section 4.1, different kinds of information-generating actions are typically modelled by directly modifying the successor state axiom for K [54, 77, 106–108]. The axiom grows from the form presented in equation (7.2) to look something like the following:

$$\begin{aligned} K(agt, s'', do(a, s)) \equiv \exists s' : s'' = do(a, s) \wedge K(agt, s', s) \wedge \\ a = A_1(\bar{x}_1) \rightarrow \Phi_1(\bar{x}_1, s, s') \\ \dots \\ a = A_n(\bar{x}_n) \rightarrow \Phi_n(\bar{x}_1, s, s') \end{aligned}$$

Here each formula Φ_i encodes the particular semantics of action A_i, typically restricting s' to satisfy some formula if and only if it is satisfied at s. Unfortunately, these extensions do not in general maintain the important theorems of the standard account of knowledge [98]; there is no guarantee that the modified axioms will preserve accessibility properties of K, or will permit a regression rule for **Knows**.

For example, Petrick [77] extends the approach of [98] to include *guarded* sensing actions. These actions cause the agent to learn that some formula ϕ holds, but only if an additional guard formula ψ also holds in the world. They are included in the axiom for K as follows:

$$a = sense_i \rightarrow \left[\psi_i(s) \rightarrow \phi_i(s) \equiv \phi_i(s')\right]$$

It is straightforward to demonstrate that the modified successor state axiom is no longer guaranteed to preserve the accessibility properties identified in Section 7.1.2, although it is possible to syntactically restrict ψ to regain this important result [77].

As another example, consider the alternate successor state axiom for K proposed by Lespérance et al. [54], in one of the few existing works in the situation calculus that does not assume a synchronous domain:

$$\begin{aligned} K(agt, s'', do(a, s)) \equiv \exists s' : K(agt, s', s) \\ \wedge (actor(a) \neq agt \rightarrow s' \leq_{actor(a) \neq agt} s'')) \\ \wedge (actor(a) = agt \rightarrow \exists s^* : \left[s' \leq_{actor(a) \neq agt} s^* \wedge \right. \\ \left. s'' = do(a, s^*) \wedge Poss(a, s^*) \wedge sr(a, s) = sr(a, s^*))\right] \end{aligned}$$

In order to account for the actions of other agents being completely hidden, this axiom must universally quantify over situation terms. It is therefore incompati-

ble with the standard regression rule for knowledge, and [54] offers no reasoning procedure other than general second-order theorem proving.

The standard approach to formalising the local information available to each agent, by directly modifying the successor state axiom for knowledge, is clearly not *elaboration tolerant* – it is easy to invalidate important properties of the formalism by changing the successor state axiom for K. As we shall demonstrate in this chapter, by basing our approach on an explicit, separately-axiomatised account of the local perspective of each agent, our formulation robustly maintains both accessibility property preservation and a regression rule regardless of the specific semantics of information-producing actions.

7.2 Knowledge and Observation

In this section we develop the axioms for our new formulation of knowledge, which is based on the explicit account of each agent's local perspective that we developed in Chapter 4. We begin from one of the basic tenets of epistemic reasoning, as described by [39]. An agent's knowledge at any particular time must depend solely on its local history: the knowledge that it started out with combined with the observations it has made since then .

Given an explicit account of the observations made by each agent, the required semantics of the K relation are clear: $K(agt, s', s)$ must hold whenever s' is legal, both s and s' would result in the same view for the agent, and s and s' are rooted at K_0-related initial situations:

$$K(agt, s', s) \equiv Legal(s') \wedge View(agt, s') = View(agt, s) \wedge K_0(root(s'), root(s)) \quad (7.4)$$

In essence, this is a direct encoding into the situation calculus of the definitions of knowledge from the classic epistemic reasoning literature [25, 39, 74].

While a wonderfully succinct definition of how knowledge should behave, this formulation cannot be used directly in a basic action theory. The dynamics of fluent change must be specified by a successor state axiom, so we must formulate a successor state axiom for the K fluent which enforces the above equivalence.

For notational convenience, let us first introduce an action description predicate *LbU* (for "legal but unobservable") indicating that the actions c are legally performed in s, but no observations will be made by agt if they occur:

$$LbU(agt, c, s) \equiv Legal(c, s) \wedge Obs(agt, c, s) = \{\} \quad (7.5)$$

By stating that $s \leq_{LbU(agt)} s'$, we assert that agt would make no observations were the world to move from situation s to s'. This means that the agent's view in both situations would be identical, so if it considers s possible then it must also consider s' possible. Following this intuition, we propose the following successor state axiom to capture the desired dynamics of the knowledge fluent:

$$\begin{aligned} K(agt, s'', do(c, s)) \equiv & \left[Obs(agt, c, s) = \{\} \rightarrow K(agt, s'', s) \right] \\ & \wedge \left[Obs(agt, c, s) \neq \{\} \rightarrow \exists c', s' : Obs(agt, c', s') = Obs(agt, c, s) \right. \\ & \qquad \left. \wedge Legal(c', s') \wedge K(agt, s', s) \wedge do(c', s') \leq_{LbU(agt)} s'' \right] \end{aligned} \tag{7.6}$$

If c was totally unobservable, the agent's state of knowledge does not change. Otherwise, it considers possible any legal successor to a possible alternate situation s' that can be brought about by an action c' yielding identical observations. It also considers possible any future of such a situation in which is would make no further observations. To reiterate: unlike the standard successor state axiom from equation (7.2), our new formalism requires agents to consider any possible future situation in which they would make no further observations, which is necessary in order to correctly specify knowledge in asynchronous domains.

It remains to specify K in the initial situation. The relation K_0 defines knowledge before *any* actions have occurred, but the agents must consider the possibility that some hidden actions have occurred. In other words, we must include situations where $root(s) \leq_{LbU(agt)} s$ in the K-relation for initial situations. We therefore propose the following axiom:

$$Init(s) \rightarrow \left[K(agt, s'', s) \equiv \exists s' : K_0(agt, s', s) \wedge s' \leq_{LbU(agt)} s'' \right] \tag{7.7}$$

Definition 23. *We will denote by $\mathcal{D}_K^{obs}$ the axioms for our new observation-based semantics for knowledge, as detailed in equations (7.6,7.7) above.*

These axioms suffice to ensure that knowledge behaves as we require: two situations will be related by $K(agt, s', s)$ if and only if they result in identical views for that agent, s' is legal, and their root situations were initially related.

Theorem 8. *For any agent agt and situations s and s'':*

$$\begin{aligned} \mathcal{D} \cup \mathcal{D}_K^{obs} \models K(agt, s'', s) \equiv \\ Legal(s'') \wedge View(agt, s'') = View(agt, s) \wedge K_0(root(s''), root(s)) \end{aligned}$$

Proof Sketch. For the *if* direction we establish each of the three conjuncts individ-

ually. The *root* case is trivial since equation (7.6) always expresses $K(s'', do(c, s))$ in terms of $K(s', s)$, while equation (7.7) relates K for initial situations back to K_0. The *Legal* case relies on the fact that *LbU* implies *Legal*, while the $View$ case relies on the fact that $s \leq_{LbU} s' \rightarrow View(s) = View(s')$. For the *only-if* direction we show how to construct an s' satisfying the $\exists s'$ parts of equations (7.6,7.7). For a detailed proof see Appendix A. □

Using this new formulation, an agent's knowledge is completely decoupled from the global notion of actions, instead depending only on the local information that it has observed. Of course, this must be combined with a specific axiomatisation of how the *Obs* function behaves. Any of the axiomatisations described in Chapter 4 can be used, and our account of knowledge will be directly applicable.

As a demonstration of the correctness of their axioms, Scherl and Levesque [98] prove five properties of their formalism: that knowledge-producing actions have only knowledge-producing effects; that unknown fluents remain unknown by default; that knowledge incorporates the results of sensing actions; that known fluents remain known by default; and that agents have knowledge of the effects of their actions.

However, the intuition behind these properties depends heavily on the assumption of public actions and on the separation of actions into two classes: knowledge-producing actions that only return sensing information, and ordinary actions that only affect the state of the world. In asynchronous multi-agent domains, these restrictions cannot be meaningfully applied.

For example, it is entirely possible that a knowledge-producing action and an ordinary action are performed concurrently by two different agents, so the results of a sensing action might immediately be made invalid. Moreover, suppose that an agent performs an action to make a formula ϕ true, but there is a series of hidden actions that could subsequently make ϕ false. The agent cannot meaningfully claim to know ϕ, since it could become false without updating the local view of that agent.

The proofs used in [98] all hinge on showing that the situations K-related to $do(a, s)$ are precisely the "correct" ones, where correctness if formulated in terms of the preconditions and effects of a. We claim that in our formulation, the "correct" situations to be related to $do(c, s)$ are precisely those that are legal and have the same view, and the validity of Theorem 8 provides sufficient justification for the correctness of our knowledge axioms.

7.3 Properties of Knowledge

With the basic axioms in place, let us study some properties of our formalism in greater detail. We begin by comparing it to the standard account of knowledge due to Scherl and Levesque [98]. Its basic assumption that "all agents are aware of all actions" is captured in our observation-based formulation using equations (4.2,4.3) from Chapter 4, which we repeat here for convenience:

$$a \in Obs(agt, c, s) \equiv a \in c$$
$$a\#r \in Obs(agt, c, s) \equiv a \in c \wedge actor(a) = agt \wedge SR(a, s) = r$$

That is, an agent observes all actions that occur, and additionally observes the sensing results of all actions that it performs. If these definitions are used, our new account of knowledge will behave identically to the standard account:

Theorem 9. *Suppose $\mathcal{D}_{ad}$ contains equations (4.2,4.3) as definitions of the Obs function, then for any legal situation terms σ and σ':*

$$\mathcal{D} \cup \mathcal{D}_K^{std} \models K(agt, \sigma', \sigma) \text{ iff } \mathcal{D} \cup \mathcal{D}_K^{obs} \models K(agt, \sigma', \sigma)$$

Proof. Equations (4.2,4.3) mean $Obs(agt, c, s)$ cannot be empty for $c \neq \{\}$, so $s = s'$ iff $s \leq_{LbU(agt)} s'$. Since we restrict our attention to legal situations, we can substitute $\perp$ for $Obs(agt, c, s) = \{\}$ and $\top$ for $Obs(agt, c, s) \neq \{\}$ into equations (7.6,7.7) to obtain the following:

$$K(agt, s'', do(c, s)) \equiv \big[\perp \rightarrow K(agt, s'', s)\big] \wedge \big[\top \rightarrow \exists c', s' : Obs(agt, c', s') = Obs(agt, c, s) \wedge Legal(c', s') \wedge K(agt, s', s) \wedge do(c', s') = s''\big]$$

$$Init(s) \rightarrow \equiv \big[K(agt, s'', s) \equiv \exists s' : K_0(agt, s', s) \wedge s' = s''\big]$$

Which further simplify to:

$$K(agt, s'', do(c, s)) \equiv \exists c', s' : Obs(agt, c', s') = Obs(agt, c, s) \wedge Legal(c', s') \wedge K(agt, s', s) \wedge do(c', s') = s'' \quad (7.8)$$

$$Init(s) \rightarrow \equiv \big[K(agt, s'', s) \equiv K_0(agt, s'', s)\big] \quad (7.9)$$

Using equations (4.2,4.3), it is straightforward to show that:

$$Obs(agt, c', s') = Obs(agt, c, s) \equiv$$
$$c = c' \wedge \forall a \in c : actor(a) = agt \rightarrow \left[SR(a, s) = SR(a, s')\right]$$

Equations (7.8,7.9) are therefore equivalent to equations (7.1,7.2) from $\mathcal{D}_K^{std}$, meaning that K behaves the same under both theories. □

Having established that our account subsumes the standard "public actions" account of knowledge, we can also show that it maintains many of its desirable properties in the general case. One of the fundamental results in [98] is that if the initial knowledge relation K_0 is reflexive, symmetric, transitive or Euclidean, then the K relation has these properties for any situation. In our formalism, such preservation of accessibility properties follows immediately from Theorem 8 and the reflexive, symmetric, transitive and Euclidean nature of the equality operator.

Theorem 10. *If the K_0 relation is restricted to be reflexive, transitive, symmetric or Euclidean, then the K relation defined by $\mathcal{D}_K^{obs}$ will satisfy the same restrictions at every legal situation.*

Proof. Each follows directly from Theorem 8 and the properties of equality. We will take the transitive case as an example; other cases are virtually identical.

Suppose that K_0 is transitive, and we have legal situations s_1, s_2, s_3 such that $K(agt, s_2, s_1)$ and $K(agt, s_3, s_2)$. Then by Theorem 8 we have the following:

$$K_0(root(s_2), root(s_1))$$
$$K_0(root(s_3), root(s_2))$$
$$View(agt, s_1) = View(agt, s_2)$$
$$View(agt, s_2) = View(agt, s_3)$$

From the transitivity of K_0 we can conclude that $K_0(root(s_3), root(s_1))$. From the transitivity of equality we can conclude that $View(agt, s_1) = View(agt, s_3)$. Since s_3 is restricted to be legal, we have enough to satisfy the RHS of the equivalence in Theorem 8, so $K(agt, s_3, s_1)$ and K is therefore transitive. □

That these properties hold regardless of the axiomatisation of *Obs* is a compelling argument in favour of our approach. As discussed in Section 7.1.3, certain kinds of sensing action can easily invalidate these properties if not axiomatised carefully. It is therefore worth considering such cases in more detail.

The problematic sensing actions identified in [77] are *guarded* sensing actions, which update $K(agt, s', s)$ according to the following axiom:

$$K(agt, s'', do(a, s)) \equiv \exists s' : s'' = do(a, s) \wedge K(agt, s', s) \wedge \ldots$$
$$a = sense_{\phi,\psi} \rightarrow \left[\psi(s) \rightarrow \phi(s) \equiv \phi(s')\right]$$

The problem with this approach is that although the agent will learn ϕ if the guard ψ is true, it cannot conclude that the guard was false by virtue of not learning ϕ. Since the agent's local perspective is only modelled implicitly, it has no way of detecting that the action failed to produce its sensing result. This means symmetry of the K relation may not be preserved.

To ensure that symmetry is preserved through action, it is necessary to axiomatise these sensing actions in such a way that the status of the guard formula itself also becomes known. While this can be achieved by syntactically restricting the formulae, as in [77], our approach of explicitly representing the observations made by each agent avoids the problem automatically.

Consider how guarded sensing actions can be axiomatised using explicit observations, as discussed in Section 4.3.3 and repeated below for convenience:

$$sense_{\phi,\psi}\#T \in Obs(agt, c, s) \equiv sense_{\phi,\psi} \in c \wedge actor(sense_\phi) = agt \wedge \psi(s) \wedge \phi(s)$$
$$sense_{\phi,\psi}\#F \in Obs(agt, c, s) \equiv sense_{\phi,\psi} \in c \wedge actor(sense_\phi) = agt \wedge \psi(s) \wedge \neg\phi(s)$$
$$sense_{\phi,\psi} \in Obs(agt, c, s) \equiv sense_{\phi,\psi} \in c \wedge actor(sense_\phi) = agt \wedge \neg\psi(s)$$

An agent using our formalism can therefore conclude, by virtue of not receiving a sensing result from $sense_{\phi,\psi}$, that the guard condition must not hold. This is sufficient to maintain symmetry of the knowledge accessibility relation as guaranteed by Theorem 10.

Our formalism is thus a proper generalisation of the standard account of knowledge in the situation calculus. It is also an *elaboration tolerant* generalisation, maintaining important properties of the axiomatisation as more complex models of sensing and observability are introduced. To demonstrate the power gained by such generalisation, Section 7.5 shows how to use our formalism to model a domain in which agents can only observe actions performed in the same room as them.

As we will demonstrate in the next section, we can also formulate a regression rule that is applicable regardless of the particular observability axioms in use. To overcome the problems that prevented the development of such a rule in [54], we use the persistence condition operator to reason about arbitrarily-long sequences of hidden actions.

7.4 Reasoning about Knowledge

The final aspect of our new account of knowledge is to extend the techniques for effective reasoning in the situation calculus to handle the modified formalism – that is, to develop a regression rule for **Knows**.

The appearance of $\leq_{LbU(agt)}$ in equation (7.6) means that our new successor state axiom universally quantifies over situations, so the regression technique developed in [98] cannot be used directly. We must appeal to the persistence condition meta-operator introduced in Chapter 6 to handle the inductive component of this reasoning, by transforming the quantification into a uniform formula so that standard regression techniques can be applied.

We propose the following as the regression rule for **Knows** under our formalism:

$$\begin{aligned}\mathcal{R}(\mathbf{Knows}(agt, \phi, do(c, s)) \stackrel{\text{def}}{=} \exists o : & \; Obs(agt, c, s) = o \\ & \wedge \; [o = \{\} \rightarrow \mathbf{Knows}(agt, \phi, s)] \\ & \wedge \; \big[o \neq \{\} \rightarrow \mathbf{Knows}(agt, \forall c' : Obs(agt, c') = o \\ & \qquad \wedge Legal(c') \rightarrow \mathcal{R}(\mathcal{P}(\phi, LbU(agt)), c'), s)\big] \quad (7.10)\end{aligned}$$

Note the similarity to the standard regression rule for knowledge in equation (7.3). New in our version are: the replacement of the **res** macro with an explicit, flexible definition of what the agent has observed; explicit handling of the case when the agent makes no observations; and use of the persistence condition to account for arbitrarily-long sequences of hidden actions.

As required for a regression rule, equation (7.10) reduces a knowledge query at $do(c, s)$ to a knowledge query at s. It is also intuitively appealing: to know that ϕ holds, the agent must know that in all situations that agree with its observations, ϕ cannot become false without it making some further observation – this is the meaning of $\mathcal{P}(\phi, LbU(agt))$ in the above, to express the agent's knowledge that "if ϕ were to become false, I would notice".

We must also specify a regression rule for **Knows** in the initial situation, as equation (7.7) also uses the $\leq_{LbU(agt)}$ ordering. This clause produces an expression in $\mathbf{Knows_0}$ at S_0, meaning that it can be handled by epistemic reasoning about the initial situation only:

$$\mathcal{R}(\mathbf{Knows}(agt, \phi, S_0)) \stackrel{\text{def}}{=} \mathbf{Knows_0}(agt, \mathcal{R}(\mathcal{P}(\phi, LbU(agt))[S_0])^{-1}, S_0) \quad (7.11)$$

The use of $\mathcal{P}$ here is similar to its use in the previous regression rule. The use of $\mathcal{R}(\phi[S_0])^{-1}$ is required to transform nested knowledge formulae into nested *initial*

knowledge formulae. For example:

$$\mathbf{Knows}(A, \mathbf{Knows}(B, \phi), S_0) \Rightarrow \mathbf{Knows_0}(A, \mathbf{Knows_0}(B, \phi), S_0)$$

When the enclosed formula ϕ does not contain nested knowledge macros, regressing it at S_0 and then suppressing the situation term will leave it unchanged.

Theorem 11. *Given a basic action theory $\mathcal{D}$ and a uniform formula ϕ:*

$$\mathcal{D} \cup \mathcal{D}_K^{obs} \models \mathbf{Knows}(agt, \phi, s) \equiv \mathcal{R}(\mathbf{Knows}(agt, \phi, s))$$

Proof Sketch. In the $do(c, s)$ case, we proceed by expanding the definition for **Knows** using our new successor state axiom for K, collecting sub-formulae that match the form of the **Knows** macro, and using regression and the persistence condition to render the resulting expressions uniform in s. In the base case, we apply the persistence condition to an expansion of **Knows** at S_0 to produce the desired result. For a detailed proof see Appendix A. □

These regression rules thus enable us to handle knowledge queries in our formalism using a standard regression-based approach, appealing to the persistence condition operator to perform the necessary inductive reasoning in an effective manner.

While this reasoning method is suitable for modelling and simulation purposes, it would be unreasonable for a situated agent to ask "do I know ϕ in the current situation?" using the situation calculus query $\mathcal{D} \models \mathbf{Knows}(agt, \phi, \sigma)$. As discussed in Chapter 4, an agent cannot be expected to have the full current situation σ. It will however have its current view v and can construct a query about its own knowledge as follows:

$$\mathcal{D} \cup \mathcal{D}_K \models \forall s : View(agt, s) = v \wedge root(s) = S_0 \wedge Legal(s) \rightarrow \mathbf{Knows}(agt, \phi, s)$$

Such a query universally quantifies over situations and so cannot be handled using regression. It is also not in a form amenable to the persistence condition operator, so the agent has no means of effectively answering such a query.

However, we should expect from Theorem 8 that the quantification over situations is unnecessary in this case – after all, any situation with the same view for that agent should result in it having the same knowledge. Let us explicitly define

knowledge with respect to a view as follows:

$$\mathbf{Knows}(agt, \phi, v) \stackrel{\text{def}}{=} \forall s : View(agt, s) = v \wedge root(s) = S_0 \wedge Legal(s) \rightarrow \mathbf{Knows}(agt, \phi, s)$$

We can then modify the regression rules in equations (7.10,7.11) to work directly on formulae of this form. The resulting rules are actually simpler than for regression over situations, as there are no empty observations in a view. The result is:

$$\mathcal{R}(\mathbf{Knows}(agt, \phi, o \cdot v)) \stackrel{\text{def}}{=} \mathbf{Knows}(agt, \forall c : Obs(agt, c) = o \wedge Legal(c) \rightarrow \mathcal{R}(\mathcal{P}(\phi, LbU(agt)), c), v) \quad (7.12)$$

$$\mathcal{R}(\mathbf{Knows}(agt, \phi, \epsilon)) \stackrel{\text{def}}{=} \mathbf{Knows_0}(agt, \mathcal{R}(\mathcal{P}(\phi, LbU(agt))[S_0])^{-1}, S_0) \quad (7.13)$$

Using regression in this way, an agent can reduce the query $\mathbf{Knows}(agt, \phi, v)$ to an equivalent query about its knowledge in the initial situation.

Theorem 12. *Given a basic action theory $\mathcal{D}$ and a uniform formula ϕ:*

$$\mathcal{D} \cup \mathcal{D}_K^{obs} \models \mathbf{Knows}(agt, \phi, v) \equiv \mathcal{R}(\mathbf{Knows}(agt, \phi, v))$$

Proof Sketch. The proof hinges on a simple corollary of Theorem 8: that situations with the same root and same view entail the same knowledge:

$$\mathcal{D} \cup \mathcal{D}_K^{obs} \models \forall s, s', s'' : root(s) = root(s') \wedge View(s) = View(s') \wedge K(agt, s'', s) \rightarrow K(agt, s'', s')$$

We can then proceed by induction over views. For the ϵ and $o \cdot v$ cases we split on whether there exists a situation having that view. If no such situation exists, we show that the regression rules (7.12, 7.13) generate a formula that is vacuously true, as an invalid view causes anything to be known. If such a solution does exist, we select an arbitrary witness and demonstrate that rules (7.12, 7.13) generate an equivalent formula to rules (7.10,7.11) using that witness. By the above corollary, this is enough to establish equivalence for any such situation. For a detailed proof see Appendix A. □

Our formalism thus allows agents to reason effectively about their own knowledge using only their local information, even in asynchronous domains where they do not know how many actions have been performed.

It is worth re-iterating that our regression rules are no longer straightforward

syntactic transformations – rather, they involve a fixpoint calculation to generate $\mathcal{P}(\phi, LbU(agt))$. Can this really be considered an effective reasoning technique? The previous work on the persistence condition meta-operator discussed the advantages of this approach in detail. The primary advantage is that this form of reasoning can be performed at all, as the alternative is general second-order theorem proving.

Of course, the ultimate proof is in the implementation. We have implemented a preliminary version of our technique and used it to verify the examples found in the following section. For details on obtaining the code, see Appendix B.

We close this section with a formal statement of a simple but important point: the persistence condition is not required when reasoning in synchronous domains. It is straightforward to show that $\mathcal{P}(\phi, LbU(agt))$ in synchronous domains is always equivalent to ϕ. The regression rules in equations (7.10,7.11) then reduce to purely syntactic manipulations.

We thus do not introduce unnecessary complications for domains in which effective reasoning procedures already exist, while extending the reach of our formalism into richer domains where some inductive reasoning is required.

Theorem 13. *Let $\mathcal{D}_{sync}$ be a synchronous basic action theory, then:*

$$\mathcal{D}_{sync} \models \forall s, agt : \phi[s] \equiv \mathcal{P}(\phi, LbU(agt))[s]$$

Proof. By definition, we have:

$$\mathcal{D}_{sync} \models \forall agt, c, s : Legal(c, s) \rightarrow Obs(agt, c, s) \neq \{\}$$

Recall from equation (7.5) that:

$$LbU(agt, c, s) \equiv Legal(c, s) \wedge Obs(agt, c, s) = \{\}$$

So clearly:

$$\mathcal{D}_{sync} \models \forall agt, c, s : LbU(agt, c, s) \equiv \bot$$

The definition of $\mathcal{P}^1(\phi, LbU(agt))$ will then produce:

$$\mathcal{P}^1(\phi, LbU(agt)) \equiv \phi \wedge \forall c : \bot \rightarrow \mathcal{R}(\phi, c) \equiv \phi$$

The calculation of $\mathcal{P}$ thus terminates immediately at the first iteration, giving $\mathcal{P}(\phi, LbU(agt))$ equal to $\mathcal{P}^1(\phi, LbU(agt))$, which is equivalent to ϕ as desired. □

7.5 An Illustrative Example

We now give a brief demonstration of our formalism in action, using it to model the "party invitation" domain outlined in Chapter 1. We adopt an explicit axiomatisation of partial observability based on the $CanObs/CanSense$ predicates introduced in Section 4.4.1.

The fluents of interest in this domain are the location of the party (the function loc) and whether each agent is in the room (the predicate $InRoom$). The action $read$ reads the invitation and returns the location of the party, while the non-sensing actions $enter$ and $leave$ cause the agents to move in/out of the room. The $read$ action is only observed by agents who are in the room. This domain can be summarised by the following axioms:

$$loc(S_0) = C$$
$$loc(do(c, s)) = l \equiv loc(s) = l$$

$$InRoom(Alice, S_0) \equiv InRoom(Bob, S_0) \equiv \top$$
$$InRoom(agt, do(c, s)) \equiv enter(agt) \in c \vee InRoom(agt, s) \wedge leave(agt) \notin c$$
$$Poss(enter(agt), s) \equiv \neg InRoom(agt, s)$$
$$Poss(leave(agt), s) \equiv InRoom(agt, s)$$

$$Poss(read(agt), s) \equiv InRoom(agt, s)$$
$$SR(read(agt), s) = r \equiv r = loc(s)$$

$$\forall agt, l : \neg \mathbf{Knows_0}(agt, loc = l, S_0)$$
$$\forall agt_1, agt_2, l : \mathbf{Knows_0}(agt_1, \neg \mathbf{Knows_0}(agt_2, loc = l), S_0)$$
$$\forall agt : \mathbf{Knows_0}(agt, InRoom(Alice) \wedge InRoom(Bob), S_0)$$

$$CanObs(agt, leave(agt'), s) \equiv CanObs(agt, enter(agt'), s) \equiv \top$$
$$CanSense(agt, leave(agt'), s) \equiv CanSense(agt, enter(agt'), s) \equiv \bot$$
$$CanObs(agt, read(agt'), s) \equiv InRoom(agt', s)$$
$$CanSense(agt, read(agt'), s) \equiv agt = agt'$$

The following are examples of knowledge queries that can be posed in our formal-

ism, a brief explanation of their outcome, and a demonstration of how they can be answered using our new regression rules. Each has been verified by the preliminary implementation of our reasoning engine.

Example 1. *Initially, Alice doesn't know where the party is:*

$$\mathcal{D} \cup \mathcal{D}_K^{obs} \models \neg\exists l : \mathbf{Knows}(A, loc = l, S_0)$$

It is given that $\neg\exists l : \mathbf{Knows_0}(A, loc = l, S_0)$, and the only way for her to learn such information is by performing a *read* action. Since she would always observe such an action, she cannot have learnt the party's location as a result of hidden actions, and the example is entailed. Formally:

$$\begin{aligned} \mathcal{P}(loc = l, LbU(A)) &\Rightarrow loc = l \\ \mathcal{R}(\neg\exists l : \mathbf{Knows}(A, loc = l, S_0)) &\Rightarrow \neg\exists l : \mathbf{Knows_0}(A, \mathcal{R}((loc = l)[S_0])^{-1}, S_0) \end{aligned}$$

Note that the calculation of this persistence condition is trivial since the location of the party cannot change. The nested regression of $loc = l$ at S_0 leaves the formula unchanged. The query thus regresses to:

$$\neg\exists l : \mathbf{Knows_0}(A, loc = l, S_0)$$

This is entailed by the domain.

Example 2. *After reading the invitation, Bob will know where the party is:*

$$\mathcal{D} \cup \mathcal{D}_K^{obs} \models \mathbf{Knows}(B, loc = C, do(\{read(B)\}, S_0))$$

The sensing results of the *read* action inform Bob of the location of the party. Since this location cannot change after any sequence of hidden actions, he can be sure of the party's location. Formally, using the fact that $Obs(B, \{read(B)\}, s) = \{read(B)\#loc(s)\}$:

$$\begin{aligned} \mathcal{R}(\mathbf{Knows}(B, loc = C, do(\{read(B)\}, S_0))) \Rightarrow \\ \exists o : o = Obs(B, \{read(B)\}, S_0) \wedge \\ \mathbf{Knows}(B, \forall c' : Legal(c') \wedge Obs(B, c') = o \rightarrow \mathcal{R}(\mathcal{P}(loc = C, LbU(B)), c'), S_0) \end{aligned}$$

Since $\mathcal{R}(\mathcal{P}(loc = C, LbU(Bob)), c')$ reduces to $loc = C$, and $loc(S_0) = C$ is given,

this simplifies to:

$$\mathbf{Knows}(B, \forall c' : Legal(c') \wedge Obs(B, c') = \{read(B)\#C\} \rightarrow loc = C, S_0)$$

Since the only possible value of c' that satisfies the antecedent is $\{read(B)\}$, we can insert the definitions of *Legal* and *Obs* to obtain:

$$\mathbf{Knows}(B, InRoom(B) \wedge loc = C \rightarrow loc = C, S_0)$$

This tautology is clearly entailed by the domain.

Example 3. *Initially, Bob knows that Alice doesn't know where the party is:*

$$\mathcal{D} \cup \mathcal{D}_K^{obs} \models \mathbf{Knows}(B, \neg\exists l : \mathbf{Knows}(A, loc = l), S_0)$$

Alice could learn the location of the party by performing the *read* action, but since Bob is in the room he would observe this action taking place. Since he has not observed it, he can conclude that Alice does not know the location of the party. Formally:

$$\mathcal{R}(\mathbf{Knows}(B, \neg\exists l : \mathbf{Knows}(A, loc = l), S_0)) \Rightarrow$$
$$\mathbf{Knows_0}(B, \mathcal{R}(\mathcal{P}(\neg\exists l : \mathbf{Knows}(A, loc = l), LbU(B))[S_0])^{-1}, S_0)$$

$$\mathcal{P}(\neg\exists l : \mathbf{Knows}(A, loc = l), LbU(Bob)) \Rightarrow$$
$$\neg\exists l : \mathbf{Knows}(A, loc = l) \wedge (InRoom(B) \vee \neg InRoom(A))$$

From the previous examples we know that:

$$\mathcal{R}(\neg\exists l : \mathbf{Knows}(A, loc = l, S_0)) \Rightarrow \neg\exists l : \mathbf{Knows_0}(A, loc = l, S_0)$$

So the entire query regresses to:

$$\mathbf{Knows_0}(B, \neg\exists l : \mathbf{Knows_0}(A, loc = l) \wedge (InRoom(B) \vee \neg InRoom(A)), S_0)$$

This is entailed by the domain.

Example 4. *After leaving the room, Bob won't know that Alice doesn't know the location of the party:*

$$\mathcal{D} \cup \mathcal{D}_K^{obs} \models \neg\mathbf{Knows}(B, \neg\exists l : \mathbf{Knows}(A, loc = l), do(\{leave(B)\}, S_0))$$

Once Bob leaves the room, he would be unable to observe Alice reading the invitation. He must therefore consider it possible that she has read it, and may know the location of the party. Formally, we can use $Obs(B, \{leave(B)\}) = \{leave(B)\}$ to regress the outer expression as follows:

$$\mathcal{R}(\neg\mathbf{Knows}(B, \phi, do(\{leave(B)\}, S_0))) \Rightarrow$$
$$\neg\mathbf{Knows}(B, InRoom(B) \rightarrow \mathcal{R}(\mathcal{P}(\phi, LbU(B)), \{leave(B)\}), S_0)$$

For the inner expression, we have from the previous example:

$$\mathcal{P}(\neg\exists l : \mathbf{Knows}(A, loc = l), LbU(B)) \Rightarrow$$
$$\neg\exists l : \mathbf{Knows}(A, loc = l) \wedge (InRoom(B) \vee \neg InRoom(A))$$

This expression is key: for Bob to know $\neg\exists l : \mathbf{Knows}(A, loc = l)$, he must also know either that he is in the room (and will thus observe the $read(A)$ action if it occurs) or that Alice is not in the room (so the $read(A)$ action will not be possible). Otherwise, Alice could learn the location of the party without him making any further observations.

When we regress over the action $\{leave(B)\}$ then $InRoom(B)$ is made false:

$$\mathcal{R}(\mathcal{P}(\neg\exists l : \mathbf{Knows}(A, loc = l), LbU(B)), \{leave(B)\}) \Rightarrow$$
$$\neg\exists l : \mathbf{Knows}(A, loc = l) \wedge (\bot \vee \neg InRoom(A))$$

And the entire expression can be simplified to:

$$\neg\mathbf{Knows}(B, InRoom(B) \rightarrow \neg\exists l : \mathbf{Knows}(A, loc = l) \wedge \neg InRoom(A), S_0)$$

Since Alice is known to be in the room, this will be entailed by the domain.

7.6 Future Applications

In this section we briefly consider how our work could integrate with some other recent developments in the situation calculus literature. While this is far from a detailed treatment, it does demonstrate the potential for interesting future work utilising our formalism.

7.6.1 Approximate Epistemic Reasoning

From the examples in the previous section, it is clear that agents may need to perform significant amounts of reasoning to answer knowledge queries about arbitrary formulae. This is on top of the already significant task of performing possible-worlds reasoning in the initial situation [77]. An interesting approach to making reasoning about knowledge more tractable is the formalism of Demolombe and Pozos Parra [23], in which knowledge is limited to be about fluent literals only.

The basic idea is to introduce, for each fluent F in the domain, two additional fluents $K^+_{agt}F$ and $K^-_{agt}F$ to explicitly represent "agt knows F" and "agt knows $\neg F$" respectively. By formulating ordinary successor state axioms for these fluents, literal-level knowledge can be reasoned about using standard regression and does not require an explicit possible-worlds K-relation. However, this approach cannot represent indeterminate disjunctive knowledge such as "agt knows F or G".

The Demolombe and Pozos Parra approach has been formally related to the standard Scherl and Levesque approach by Petrick and Levesque [76]. They show there is an equivalence between the two approaches when an agent's knowledge is restricted to be *disjunctive*, so that the following holds:

$$\mathbf{Knows}(agt, \phi_1 \vee \phi_2, s) \rightarrow \mathbf{Knows}(agt, \phi_1, s) \vee \mathbf{Knows}(agt, \phi_2, s)$$

In [77] this equivalence is extended to cover existential quantification by restricting knowledge to also satisfy the following:

$$\mathbf{Knows}(agt, \exists x : \phi(x), s) \rightarrow \exists x : \mathbf{Knows}(agt, \phi(x), s)$$

These disjunctive properties of knowledge are *not* entailed by a general possible-worlds style theory in the tradition of [98], although there are restrictions that can be placed on the theory in order to enforce them [77, 78].

While we do not consider maintenance of these disjunctive knowledge properties in any detail, we do note that they also permit a sound *approximation* of knowledge that can be reasoned about more tractably than the standard possible-worlds

account. Following the style of [76] we could provide the following definitions:

$$\mathbf{Knows}_A(agt, \phi_1 \wedge \phi_2, s) \stackrel{\text{def}}{=} \mathbf{Knows}_A(agt, \phi_1, s) \wedge \mathbf{Knows}_A(agt, \phi_2, s)$$
$$\mathbf{Knows}_A(agt, \neg(\phi_1 \wedge \phi_2), s) \stackrel{\text{def}}{=} \mathbf{Knows}_A(agt, \neg\phi_1, s) \vee \mathbf{Knows}_A(agt, \neg\phi_2, s)$$
$$\mathbf{Knows}_A(agt, \forall x : \phi(x), s) \stackrel{\text{def}}{=} \forall x : \mathbf{Knows}_A(agt, \phi(x), s)$$
$$\mathbf{Knows}_A(agt, \neg\forall x : \phi(x), s) \stackrel{\text{def}}{=} \exists x : \mathbf{Knows}_A(agt, \neg\phi(x), s)$$
$$\mathbf{Knows}_A(agt, \neg\neg\phi, s) \stackrel{\text{def}}{=} \mathbf{Knows}_A(agt, \phi, s)$$
$$\mathbf{Knows}_A(agt, F, s) \stackrel{\text{def}}{=} \mathbf{Knows}(agt, F, s)$$
$$\mathbf{Knows}_A(agt, \neg F, s) \stackrel{\text{def}}{=} \mathbf{Knows}(agt, \neg F, s)$$

A knowledge query is split across the logical operators until we are left with only knowledge of fluent literals, which is then handled using the formalism presented in this chapter. If we assume a finite number of fluents, then we can use our regression rule for knowledge to *pre-calculate* an explicit successor state axiom for $\mathbf{Knows}_A(agt, F, s)$ and $\mathbf{Knows}_A(agt, \neg F, s)$, allowing them to be treated as primitive fluents and reasoned about at run-time using purely syntactic transformations.

Unlike the approach of [23] in which the knowledge literal fluents must be axiomatised separately from the actual fluents they describe, the approach suggested here would allow a successor state axiom for literal-level knowledge to be derived from the dynamics of the domain. All persistence condition calculations could be performed once, offline, and then used directly for approximate reasoning about the knowledge of an agent.

7.6.2 Knowledge under a Protocol

The knowledge formalism we have developed here is *permissive*, in that it assumes the world could potentially evolve via any legal sequence of actions. In the wider field of epistemic reasoning, it is common to constrain the world to evolve according to a given *protocol* [25, 39, 118]. One then speaks of an agent's knowledge under a particular protocol.

As discussed in [118], permissive formulations of knowledge can easily be extended to support *local* protocols, where the allowable next actions can be determined based on the current state of the world. Our use of *Legal* in the axioms for knowledge could easily be replaced with predicates axiomatising actions that are, for example, *Permissable* or *Motivated*. But recent work by Fritz et al. [33] also presents an intriguing possibility to extend our approach to more general protocols.

The most natural language for expressing a protocol in the situation calculus is

Golog, so one may wish to reason about an agent's knowledge assuming the world evolves as specified by the Golog program δ:

$$K_p(agt, \delta, s', s) \stackrel{\text{def}}{=} K(agt, s', s) \wedge \exists s'', \delta' : Init(s'') \wedge Trans^*(\delta, s'', \delta', s')$$

Such knowledge would be queried like so:

$$\mathcal{D} \cup \mathcal{D}_{golog} \cup \mathcal{D}_K^{obs} \models \mathbf{Knows}_p(agt, \delta, \phi, \sigma)$$

Indeed, it is this kind of knowledge that would be needed to integrate epistemic reasoning into our MIndiGolog execution planner from Chapters 3 and 5, as the agents in that case know that their teammates will act according to the shared control program.

Fritz et al. [33] have demonstrated that the details of a given ConGolog program δ can be *compiled into* a theory of action $\mathcal{D}$, producing a new theory $\mathcal{D}^\delta$ in which the only legal situations are those that form part of a legal execution of δ:

$$\mathcal{D} \cup \mathcal{D}_{golog} \models \exists \delta' : Trans^*(\delta, S_0, \delta', \sigma)$$
$$\text{iff}$$
$$\mathcal{D}^\delta \models Legal(\sigma)$$

So to investigate the knowledge of an agent under a protocol, we could use the compilation technique of [33] to get a query which can be handled using the formalism developed in this chapter:

$$\mathcal{D} \cup \mathcal{D}_{golog} \cup \mathcal{D}_K^{obs} \models \mathbf{Knows}_p(agt, \delta, \phi, \sigma)$$
$$\text{iff}$$
$$\mathcal{D}^\delta \cup \mathcal{D}_K^{obs} \models \mathbf{Knows}(agt, \phi, \sigma)$$

The details are not quite so straightforward, as the compilation procedure introduces some auxiliary actions and fluents that should be hidden from the agent's knowledge. However, it does offer an intriguing possibility for future work.

7.7 Discussion

In this chapter, we have used a principled axiomatisation of the observability of actions to explicitly define an agent's knowledge in terms of its local view. By reifying observations and views as terms in the logic, we are able to give a succinct definition of the dynamics of the knowledge fluent and prove that its behaviour

matches our intuitive expectations. We have demonstrated that our account of knowledge is expressive enough to capture the standard account of knowledge based on public actions, as well as more complex formulations where the observability of actions depends on the state of the world.

Moreover, it maintains its important theorems and the availability of a regression rule as more complex kinds of information-producing action are introduced, such as those discussed in Chapter 4. The importance of this guaranteed elaboration tolerance should not be underestimated.

As an example of the problems that can arise when trying to characterise knowledge using an implicit representation of an agent's local perspective, consider again one of the few existing formulations of knowledge in the situation calculus that allows for hidden actions, that of [54]. Their successor state axiom for the K fluent is repeated below:

$$\begin{aligned} K(agt, s'', do(a, s)) \equiv\ & \exists s' : K(agt, s', s) \\ & \wedge\ (actor(a) \neq agt \rightarrow s' \leq_{actor(a) \neq agt} s'')) \\ & \wedge\ (actor(a) = agt \rightarrow \exists s^* : \big[s' \leq_{actor(a) \neq agt} s^* \wedge \\ & \qquad s'' = do(a, s^*) \wedge Poss(a, s^*) \wedge sr(a, s) = sr(a, s^*))\big] \end{aligned}$$

While the axiom seems intuitively plausible, it has a subtle problem: an agent's knowledge can change in response to actions performed by others. Suppose that agt has just performed action a_1, so the world is in situation $do(a_1, s)$. Another agent then performs the action a_2, leaving the world in situation $do(a_2, do(a_1, s))$. Since it is not aware of the occurrence of a_2, the knowledge of agt should be unchanged between these two situations. This is not the case under the formulation of [54], which introduces arbitrarily-long sequences of hidden actions into the *past* of the possible situation s''. Based on our explicit formalisation of the agent's local view, our axiom includes hidden actions in the *future* of s'' and avoids this unintuitive behaviour.

Our insistence on allowing for "all possible future actions" may seem like it would leave the agents with too little knowledge. Indeed, it is easy to construct cases in which agents can never know the value of certain fluents. We argue that this restriction is necessary in asynchronous domains: if Definition 9 is accepted as the definition of a view, and equation (7.4) is accepted as the definition for how knowledge should behave, then the agent is required to consider any arbitrarily-long sequence of legal hidden actions.

Note, however, that the sequences of actions considered must not only be legal,

but unobservable as well. As shown in our example domain, if the agents have good observability of parts of the domain, they can acquire significant amounts of knowledge because there will be few hidden actions.

One approach to further taming these hidden future situations is to assume that agents always know the current time, or some bound on the current time. They can use this information in conjunction with their local view to determine what they know, given that the current time is τ:

$$\mathcal{D} \cup \mathcal{D}_K^{obs} \models \mathbf{Knows}(agt, (start < \tau) \rightarrow \phi, v)$$

Since this query is uniform, it can be approached directly using our regression rules. If all actions take some finite duration, τ will effectively bound the length of hidden action sequences that need to be considered. Not only will this permit the agents to gain knowledge of more fluents, but it will also guarantee the termination of the persistence condition calculation. In essence, this approach adds some synchronicity back into the domain in order to allow more effective reasoning.

Another option would be to abandon the requirement that the agents *know* particular fluents and formulate a logic of *belief*. Belief-based formalisms have been constructed for the situation calculus using a possible-worlds approach similar to that used for knowledge [109]. In such formalisms agents are allowed to be mistaken, and so do not need to guarantee their knowledge is correct by considering all possible ways that the world might evolve. One example of an alternative is the work of Shapiro and Pagnucco [105], who have shown how an agent can hypothesise the occurrence of hidden actions when it discovers that its beliefs are mistaken.

We do not consider belief-based systems in this thesis, but note that the use of a similar possible-worlds semantics means that our explicit notions of observations and views could also provide benefits for such formalisms, as well as for the formulation of other modalities (e.g goals as in [107]) in the situation calculus.

We have thus provided the first foundational account of epistemic reasoning in the situation calculus in asynchronous multi-agent domains, including a precise characterisation of the inductive reasoning required to handle knowledge using a regression rule in such domains. The effectiveness of automated reasoning in our formalism is highly dependent on the effectiveness of persistence condition calculation. Since this is a fixpoint calculation, it can be computationally expensive and even undecidable in complex domains. But by factoring out the necessary inductive reasoning into a separate operator, it can now be studied and improved in isolation.

Finally, and perhaps most importantly, we have shown that a simple modification

to our regression rules allows a situated agent to reason directly about its own knowledge using only its local view, rather than constructing a query that universally quantifies over all situations compatible with its view. Our new observation-based semantics thus provides a powerful account of knowledge suitable both for reasoning *about*, and for reasoning *in*, asynchronous multi-agent domains.

Chapter 8

Complex Epistemic Modalities

This chapter develops an explicit formal treatment of group-level epistemic modalities in the situation calculus. The primary motivation for this work is a formal treatment of common knowledge that permits effective reasoning with a regression rule, but as we shall see, this requires significant technical machinery capable of handling much more general epistemic modalities.

Attempts to reason about common knowledge are bound by a fundamental expressivity limitation: the regression of common knowledge cannot be expressed in terms of common knowledge alone [8]. To overcome this limitation we take our cue from recent promising work in dynamic epistemic logic, with the main idea due to van Benthem et al. [119]: increase the expressiveness of the epistemic language so it is strong enough to formulate a proper regression rule. They have developed the Logic of Communication and Change (henceforth "LCC") using propositional dynamic logic to express epistemic modalities, and have shown that it allows reasoning about common knowledge using techniques akin to regression. We follow a similar approach in this chapter and introduce complex epistemic modalities to the situation calculus.

While this chapter naturally parallels the development of LCC, there are also substantial differences. LCC is built on modal logic and so handles only propositional, synchronous domains. The richer ontology of the situation calculus means our formalism must support first-order preconditions and effects, quantifying-in and de-dicto/de-re, and arbitrary sets of concurrent actions. It must also incorporate our new technique for handling hidden actions while remaining compatible with other extensions to the situation calculus. By building on our rigorous observation-based semantics for individual knowledge, and using a macro-expansion approach to construct group-level modalities, our formalism is able to neatly fulfill all these

requirements.

The language of first-order dynamic logic is adopted to construct complex epistemic paths, with the macro $\mathbf{PKnows}(\pi, \phi, s)$ expressing knowledge using such a path. Since common knowledge is defined as the transitive closure of the union of the agents' base knowledge operators, it can be expressed as $\mathbf{PKnows}((A \cup B)^*, \phi, s)$. Regression is then modified to treat $\mathbf{PKnows}(\pi, \phi, do(c, s))$ as a primitive fluent, producing an equivalent formula $\mathbf{PKnows}(\mathcal{T}(\pi), \mathcal{R}(\phi), s)$. Here $\mathcal{T}$ is a new meta-level operator called the epistemic path regressor.

After some more detailed background material in Section 8.1, the chapter proceeds as follows. Section 8.2 defines a variant of first-order dynamic logic for use as an epistemic path language, which is encoded in the situation calculus using the macro $\mathbf{KDo_0}(\pi, s, s')$. This macro is analogous to the fluent $K(agt, s', s)$ from Chapter 7 but expresses more complex epistemic relationships between situations.

Section 8.3 develops a *synchronous* account of complex epistemic modalities. The macro $\mathbf{PKnows_0}(\pi, \phi, s)$ expresses knowledge using an epistemic path π under the assumption that there have been no hidden actions. We develop a regression rule for the synchronous case by encoding the effects of an action inside the epistemic path as well as the enclosed formula, transforming $\mathbf{PKnows_0}(\pi, \phi, do(c, s))$ into an equivalent expression $\forall c' : \mathbf{PKnows_0}(\mathcal{T}(\pi, c, c'), \mathcal{R}(\phi, c'), s)$.

Section 8.4 introduces hidden actions by using the empty action set to explicitly represent a hidden action. We simulate agents reasoning about arbitrarily-long sequences of hidden actions by inserting arbitrarily many empty action sets between each real action in a situation term. A regression rule is formulated for this infinitary construction using the persistence condition meta-operator in a similar way to the previous chapter. Section 8.5 then demonstrates the correctness of this construction by showing that in the case of a single agent, it precisely matches the definition of individual knowledge presented in Chapter 7.

The effectiveness of our new technique is demonstrated in Section 8.6 by reasoning about common knowledge in an asynchronous, partially observable domain, a first for the situation calculus. Section 8.7 compares the resulting formalism with LCC, highlighting both similarities and differences, while Section 8.8 discusses the implications of our more powerful epistemic language for answering the regressed knowledge query. Finally, Section 8.9 concludes with some general discussion.

The end result is a powerful account of complex group-level epistemic modalities constructed almost entirely in the meta-level reasoning machinery of the situation calculus, and a regression-based effective reasoning procedure capable of reasoning about common knowledge in asynchronous domains.

8.1 Background

Let us begin by recalling the definitions of group-level modalities from Section 2.3:

$$\mathbf{EKnows}(G, \phi, s) \stackrel{\text{def}}{=} \bigwedge_{agt \in G} \mathbf{Knows}(agt, \phi, s)$$
$$\mathbf{EKnows}^1(G, \phi, s) \stackrel{\text{def}}{=} \mathbf{EKnows}(G, \phi, s)$$
$$\mathbf{EKnows}^n(G, \phi, s) \stackrel{\text{def}}{=} \mathbf{EKnows}(G, \mathbf{EKnows}^{n-1}(G, \phi), s)$$
$$\mathbf{CKnows}(G, \phi, s) \stackrel{\text{def}}{=} \bigwedge_{n \in \mathbb{N}} \mathbf{EKnows}^n(G, \phi, s)$$

Finite group-level knowledge operators such as **EKnows** can be reasoned about using existing techniques by simply expanding out the definitions into individual-level knowledge operators, and applying the regression rules developed in the previous chapter. But since the definition of common knowledge is second-order or infinitary, it cannot be handled in this way.

Existing treatments of common knowledge in the situation calculus and related literature specify it as the transitive closure of **EKnows** using an explicit second-order axiom [16, 35]. While logically sound, this approach forgoes the use of regression as an effective reasoning technique. Indeed, reasoning in such formalisms requires a second-order theorem prover.

This difficulty in effectively handling common knowledge can be attributed to a famous expressivity result from Batlag et al. [8] in the related field of dynamic epistemic logic:

> Epistemic logic with actions and common knowledge is more expressive than epistemic logic with common knowledge alone

In our terminology: given a formula $\mathbf{CKnows}(G, \phi, do(c, s))$, it is impossible in general to find an equivalent formula $\mathbf{CKnows}(G, \psi, s)$. We therefore cannot formulate a regression rule for common knowledge in terms of **Knows** and **CKnows**.

Given the deep similarities between the situation calculus and dynamic epistemic logic [117], we can be confident that this expressivity limitation also holds in the situation calculus. Rather than attempting to formally establish it, we present a short summary of the intuitions behind the result and why it should be expected to hold, then proceed directly with a technique to circumvent it.

Consider again the standard successor state axiom for the K fluent, which has the simplified general form:

$$K(do(c', s'), do(c, s)) \equiv K(s', s) \wedge \Phi_K(c', s')$$

We could construct an analogous fluent E that captures the **EKnows** relation, with a successor state axiom of the general form:

$$E(do(c', s'), do(c, s)) \equiv E(s', s) \wedge \Phi_E(c', s')$$

Now consider constructing such a fluent for the **EKnows**^{2} relation. The general form for its successor state axiom must hypothesise an intermediate situation:

$$\begin{aligned} E^2(do(c', s'), do(c, s)) &\equiv \exists c'', s'' : E(do(c'', s''), do(c, s)) \wedge E(do(c', s'), do(c'', s'')) \\ &\Rightarrow \exists c'', s'' : E(s'', s) \wedge E(s', s'') \wedge \Phi_E(c', s') \wedge \Phi_E(c'', s'') \\ &\Rightarrow \exists c'', s'' : E^2(s', s) \wedge \Phi_E(c', s') \wedge \Phi_E(c'', s'') \end{aligned}$$

This axiom must perform tests not only at s and s', but also at the hypothesised intermediate situation s''. Extending this reasoning, a successor state axiom for common knowledge would be required make assertions not only about s and s', but all of the intermediate situations in the transitive closure. However, the macro **CKnows** can only make assertions about the final situation reached in the transitive closure, not about the situations on the path leading to it. It is thus not expressive enough to formulate a proper regression rule.

To overcome this expressiveness limitation, we follow the recent promising work of van Benthem et al. [119], who use two important new ideas to produce a regression rule for common knowledge in their logic LCC:

- Form more expressive epistemic modalities using the syntax of dynamic logic, interpreted over the epistemic frame of the agents.
- Apply regression within the modality as well as to the enclosed formula.

While the full details of LCC would take us too far afield in this thesis, let us demonstrate the basic idea. LCC uses explicit *update models* to model partially-observable actions. These are finite Kripke structures in which each world represents a possible action that was performed. To represent that ϕ holds after performing event e from update frame U, LCC uses the standard box modality of dynamic logic:

$$[U, e]\phi$$

To express the agent's knowledge, the language of Propositional Dynamic Logic (henceforth "PDL") is adopted with standard semantics [40], but interpreted over the epistemic frame of the agents instead of over actions. Individual-level knowledge

is expressed as in standard modal logic:

$$[Bob]\phi$$

But these base knowledge operators can be combined using the dynamic logic operators choice ($\cup$), sequence (;), test ($?\phi$) and iteration ($*$). Common knowledge between *Alice* and *Bob* is expressed in LCC as:

$$[(Alice \cup Bob)^*]\phi$$

This modality is the transitive closure of the union of the agents' individual knowledge operators, which is one of the common semantic definitions of common knowledge. To perform reasoning, "reduction rules" are used that are similar in spirit to the regression operator of the situation calculus. Let π be an arbitrary epistemic modality and U an update model with n possible actions, then the reduction rule for knowledge in LCC is:

$$[U, e_i][\pi]\phi \;\Rightarrow\; \bigwedge_{j=0}^{n-1} [T_{ij}^U(\pi)][U, e_j]\phi$$

This definition enumerates all n possible actions that could be mistaken for the real action e_i according to update model U, then uses a special *program transformer* T_{ij}^U to encode the information from U into the epistemic modality π. We shall discuss some of the details of T_{ij}^U later in this chapter.

Our work applies these ideas to perform group-level epistemic reasoning in the situation calculus, allowing common knowledge to be handled using regression. While the development naturally parallels that of LCC, the much richer ontology of the situation calculus means there are also substantial differences. In particular:

- LCC is *propositional*: actions do not take arguments, there are finitely many actions, and no quantification is required.
- LCC is *synchronous*: reasoning is performed by regressing one action at a time, without the "all possible futures" approach needed to handle hidden actions.

By contrast, our formalism must capture first-order preconditions and effects, quantifying-into and de-dicto/de-re, and arbitrary sets of concurrent actions, while accounting for arbitrarily-long sequences of hidden actions and remaining compatible with other extensions to the situation calculus.

As we shall see throughout the development, the extra expressiveness of the situation calculus also provides some advantages for our formalism. We do not need

to manipulate explicit update models, since we have reified action terms and a full axiomatisation of each agent's epistemic uncertainty. The definition of our regression rules will also turn out to be much simpler than the analogous rules in [119].

8.2 Epistemic Paths

We will be approaching our formalism in two steps: first defining complex epistemic modalities in synchronous domains, and then building support for hidden actions on top of that foundation. So to begin, we must define the synchronous knowledge of an individual agent in an arbitrary situation s. This is the agent's knowledge when it assumes that no hidden actions have occurred, and so it is not required to do any "all possible futures" style reasoning. We extend the fluent $K_0(agt, s', s)$ which is already used to represent synchronous knowledge in the initial situation. The axiom set $\mathcal{D}_K^{obs}$ gains the following successor state axiom for K_0:

$$\begin{aligned} K_0(agt, s'', do(c, s)) \equiv \exists s', c' :\ & K_0(agt, s', s) \wedge Obs(agt, c, s) = Obs(agt, c', s') \\ & \wedge \big(s'' = do(c', s') \wedge Legal(c', s') \ \vee \ s'' = s' \wedge c' = \{\}\big) \end{aligned} \quad (8.1)$$

Given synchronicity, this axiom is a simple modification of the standard successor state axiom for knowledge from equation (7.2). The only complication is that when $Obs(agt, c, s) = \{\}$, the agent considers it possible that no actions were actually performed, so $s' = s''$ and $c' = \{\}$. Thus the number of actions in s puts an upper bound on the number of actions that the agent thinks might actually have occurred.

While we hope this definition is intuitively plausible, we will not formally relate it to the definitions given in the previous chapter until Section 8.5. Beginning with the assumption of synchronicity allows us to focus first on increasing the expressiveness of the epistemic language. Once this has been achieved, we will generalise the formalism to asynchronous domains.

The next step is to adopt the language of dynamic logic to express complex *epistemic* modalities rather than modalities regarding action. To deal gracefully with the many first-order aspects of the situation calculus we use a variant of *first-order dynamic logic* (henceforth "FODL"), which we adapt with some simplifications from the dynamic term-modal logic of Kooi [49].

First, we must specify the syntax of our epistemic path language. We will use π to denote an arbitrary epistemic path expression.

Definition 24 (Epistemic Path)**.** *Let agt be an* AGENT *term, ϕ a uniform formula and x a variable name, then the epistemic path terms π are the smallest set matching the following structural rules:*

$$\pi ::= agt \mid ?\phi \mid \pi_1; \pi_2 \mid \pi_1 \cup \pi_2 \mid \pi^* \mid \exists x$$

The test (?), sequence (;), choice ($\cup$) and transitive closure (*) operators are standard in dynamic logic, although test formulae may now contain variables that must be interpreted. The operator $\exists x$ allows the value of a variable to change during path traversal, by non-deterministically re-binding x to some value. The variables used in paths must be distinct from all symbols in $\mathcal{L}_{sitcalc}$.

The semantics of this epistemic path language are defined at the meta-level as a series of macro expansions. Formulae of first-order dynamic logic are interpreted relative to both a "current world" and a "current variable binding" [49]. We will represent the variable binding as a first-order substitution μ; the notation $\mu(\phi)$ applies the substitution to the variables in ϕ and $\mu[x/z]$ sets the value of variable x to the term z. However, we do not want to introduce new terms or axioms to $\mathcal{L}_{sitcalc}$ in order to model these substitutions as concrete objects. Since any path π will have a finite number of variables, the substitutions used in its macro-expansion can be directly replaced with finite tuples of variables from $\mathcal{L}_{sitcalc}$. We will continue to use the notation μ in line with the standard semantics of FODL, which operate over pairs (μ, s).

Definition 25 (Epistemic Path Semantics)**.** *A situation s' is reachable from situation s via epistemic path π, denoted $\mathbf{KDo_0}(\pi, s, s')$, according to the following definitions:*

$$\mathbf{KDo_0}(\pi, s, s') \stackrel{def}{=} \exists \mu, \mu' : \mathbf{KDo_0}(\pi, \mu, s, \mu', s')$$

$$\mathbf{KDo_0}(agt, \mu, s, \mu', s') \stackrel{def}{=} \mu' = \mu \wedge K_0(agt, s', s)$$

$$\mathbf{KDo_0}(?\phi, \mu, s, \mu', s') \stackrel{def}{=} s' = s \wedge \mu' = \mu \wedge \mu(\phi)[s]$$

$$\mathbf{KDo_0}(\pi_1; \pi_2, \mu, s, \mu', s') \stackrel{def}{=} \exists \mu'', s'' : \mathbf{KDo_0}(\pi_1, \mu, s, \mu'', s'') \wedge \mathbf{KDo_0}(\pi_2, \mu'', s'', \mu', s')$$

$$\mathbf{KDo_0}(\pi_1 \cup \pi_2, \mu, s, \mu', s') \stackrel{def}{=} \mathbf{KDo_0}(\pi_1, \mu, s, \mu', s') \vee \mathbf{KDo_0}(\pi_2, \mu, s, \mu', s')$$

$$\mathbf{KDo_0}(\exists x, \mu, s, \mu', s') \stackrel{def}{=} \exists z : s' = s \wedge \mu' = \mu[x/z]$$

$$\forall P : [\mathbf{refl} \wedge \mathbf{tran} \wedge \mathbf{cont}] \rightarrow \left[P(\mu, s, \mu', s') \rightarrow \mathbf{KDo_0}(\pi^*, \mu, s, \mu', s')\right]$$

Where we use the following abbreviations (standing for "reflexive", "transitive"

and "contains" respectively) to specify π^* as the reflexive transitive closure of π:

$$\begin{aligned}
\mathbf{refl} &\stackrel{\text{def}}{=} P(\mu, s, \mu, s) \\
\mathbf{tran} &\stackrel{\text{def}}{=} \forall \mu_1, s_1, \mu_2, s_2 : P(\mu_1, s_1, \mu, s) \wedge \mathbf{KDo_0}(\pi, \mu_2, s_2, \mu_1, s_1) \rightarrow P(\mu_2, s_2, \mu, s) \\
\mathbf{cont} &\stackrel{\text{def}}{=} \forall \mu', s' : \mathbf{KDo_0}(\pi, \mu', s', \mu, s) \rightarrow P(\mu', s', \mu, s)
\end{aligned}$$

Let us re-iterate: paths are not terms in $\mathcal{L}_{sitcalc}$, but rather are handled by macro-expansion of $\mathbf{KDo_0}(\pi, s, s')$ into second-order sentences of the situation calculus. In particular, this means that the variables used in epistemic paths are not actual terms and cannot appear outside of a $\mathbf{KDo_0}$ macro.

For notational convenience, we also introduce an explicit assignment operator:

$$\mathbf{KDo_0}(x \Leftarrow \tau, \mu, s, \mu', s') \stackrel{\text{def}}{=} \mathbf{KDo_0}(\exists x\,;\, ?x = \tau, \mu, s, \mu', s')$$

This operator non-deterministically rebinds x to any value, then immediately asserts that it is equal to the specific value τ. Since this expands to a test formula, τ can potentially be a functional fluent that is interpreted at the current situation. It cannot, however, depend on the current value of x.

8.3 A Synchronous Epistemic Fluent

At this point it's worth reviewing again the purpose of this path language. Despite utilising the syntax of dynamic logic, it is *not* related to actions in any way. Rather it expresses complex *epistemic* paths, and is interpreted over the epistemic frame generated by the agents' knowledge relations. We will be introducing a new macro $\mathbf{PKnows}(\pi, \phi, s)$ (read this as "Path-Knows") to express knowledge using these epistemic paths. To make this clear, here is how some different kinds of knowledge would be expressed using the standard account of knowledge, and how we intend to express them using epistemic paths:

$$\begin{aligned}
\mathbf{Knows}(A, \phi, s) &\equiv \mathbf{PKnows}(A, \phi, s) \\
\mathbf{Knows}(A, \mathbf{Knows}(B, \phi), s) &\equiv \mathbf{PKnows}(A; B, \phi, s) \\
\mathbf{Knows}(A, \phi, s) \wedge \mathbf{Knows}(B, \phi, s) &\equiv \mathbf{PKnows}(A \cup B, \phi, s) \\
\mathbf{EKnows}(G, \phi, s) &\equiv \mathbf{PKnows}(\bigcup_{a \in G} a, \phi, s) \\
\mathbf{CKnows}(G, \phi, s) &\equiv \mathbf{PKnows}((\bigcup_{a \in G} a)^*, \phi, s) \qquad (8.2)
\end{aligned}$$

In this section, we develop a synchronous version $\mathbf{PKnows_0}(\pi, \phi, s)$ of our path-

knowledge operator, since the semantics of $\mathbf{KDo_0}$ reference the synchronous knowledge fluent K_0 defined earlier. The definition of $\mathbf{PKnows_0}$ is a straightforward analogue of the individual-level **Knows** macro:

$$\mathbf{PKnows_0}(\pi, \phi, s) \stackrel{\text{def}}{=} \forall s' : \mathbf{KDo_0}(\pi, s, s') \rightarrow \phi[s']$$

By virtue of $\mathbf{KDo_0}$ this macro expands to a complicated second-order formula in the base language of the situation calculus. As with the case of the basic **Knows** macro, we need to treat $\mathbf{PKnows_0}$ syntactically as a primitive fluent. This means we need a regression rule for such expressions. It is here that we incorporate the second key idea from LCC – use of a syntactic transform to encode the effects of actions within epistemic paths as well as in primitive formulae. Mirroring LCC, we introduce the meta-operator $\mathcal{T}_\mathcal{D}$ for this purpose. As with regression and the persistence condition, we assume a fixed action theory and write $\mathcal{T}$ rather than $\mathcal{T}_\mathcal{D}$.

Let us consider the required operation of $\mathcal{T}$ by analogy with the standard regression operator $\mathcal{R}$. One can think of regression as a "pre-encoding" of the effects of an action: ϕ will hold in $do(c, s)$ if and only if $\mathcal{R}(\phi, c)$ holds in s. The path regressor $\mathcal{T}$ needs to lift this idea to epistemic paths as follows: there is a π-path from $do(c, s)$ to $do(c', s')$ if and only if there is a $\mathcal{T}(\pi, c, c')$-path from s to s'.

In order to accomplish this task of pre-encoding the effects of actions, the path regressor will need to make various assertions about the action that is to be performed in each situation traversed by the regressed path. It uses a fresh variable to keep track of this "current action". The basic operation of $\mathcal{T}$ is as follows:

- Introduce a fresh variable x to hold the action that was performed in the current situation;
- at the beginning of the path, bind x to the known action c;
- at the end of the path, assert that x is the known action c'; and
- when the path moves to a new situation, select a new action using $\exists x$.

This is accomplished with an auxiliary operator $\mathcal{T}_a(\pi, x)$, which translates π under the assumption that x contains the action to be performed in the current situation.

Definition 26 (Epistemic Path Regressor)**.** *The epistemic path regressor $\mathcal{T}(\pi, c, c')$ operates according to the definitions below, where x and z are fresh path variables not appearing in π:*

$$\mathcal{T}(\pi, c, c') \stackrel{def}{=} x \Leftarrow c\,;\, \mathcal{T}_a(\pi, x)\,;\, ?x = c'$$

$$\mathcal{T}_a(agt, x) \stackrel{def}{=} z \Leftarrow Obs(agt, x)\,;\, agt\,;\, \exists x\,;\, ?Legal(x) \vee x = \{\}\,;\, ?Obs(agt, x) = z$$
$$\mathcal{T}_a(?\phi, x) \stackrel{def}{=} ?\mathcal{R}(\phi, x)$$
$$\mathcal{T}_a(\exists y, x) \stackrel{def}{=} \exists y$$
$$\mathcal{T}_a(\pi_1; \pi_2, x) \stackrel{def}{=} \mathcal{T}_a(\pi_1, x)\,;\, \mathcal{T}_a(\pi_2, x)$$
$$\mathcal{T}_a(\pi_1 \cup \pi_2, x) \stackrel{def}{=} \mathcal{T}_a(\pi_1, x) \cup \mathcal{T}_a(\pi_2, x)$$
$$\mathcal{T}_a(\pi^*, x) \stackrel{def}{=} \mathcal{T}_a(\pi, x)^*$$

Most of these rules are straightforward, but note how the clause for an individual agent term encodes the successor-state axiom for K_0 from equation (8.1). In order to capture the requirement that $Obs(agt, c, s) = Obs(agt, c', s')$ it uses a new variable z which is bound to the agent's observations in the current situation. It then moves to a new situation by making an agt path step, and selects a new value for the current action x; this corresponds to the $\exists c', s'$ in equation (8.1). Finally, it asserts that the observations in the new situation match those recorded in z.

Also note that c and c' are proper situation calculus variables, not path variables, and are being introduced into the path from outside the scope of the $\mathbf{KDo_0}$ and $\mathbf{PKnows_0}$ macros. The path regressor in essence encodes the conditions under which an occurrence of action c could be mistaken for an occurrence of c'.

The following theorem states that these definitions behave has desired, respecting the semantics of epistemic paths:

Theorem 14. *For any epistemic path π:*

$$\mathcal{D} \cup \mathcal{D}_K^{obs} \models \mathbf{KDo_0}(\pi, do(c, s), s'') \equiv$$
$$\exists c', s' : \left(c' \neq \{\} \wedge s'' = do(c', s') \vee c' = \{\} \wedge s'' = s'\right) \wedge \mathbf{KDo_0}(\mathcal{T}(\pi, c, c'), s, s')$$

Proof Sketch. The proof proceeds by cases, covering each path operator in turn. The base cases agt, $?\phi$ and $\exists y$ follow from Definition 25 and the successor state axiom for K_0 in equation (8.1). The inductive cases are straightforward as $\mathcal{T}_a$ is simply pushed inside each operator. For a detailed proof see Appendix A. □

Note that the situations reachable by $\mathbf{KDo_0}(\pi, do(c, s), s'')$ are not necessarily successors of the situations reachable by $\mathbf{KDo_0}(\mathcal{T}(\pi, c, c'), s, s')$ – if one of the agents mentioned in π does not make any observations, c' is permitted to be empty and the situation s' itself is still reachable from $do(c, s)$. This mirrors the handling of empty observations by the successor state axiom for K_0 and will be critical when we extend the formalism to asynchronous domains.

Given that $\mathcal{T}$ correctly regresses our epistemic path language, we are free to use it to define the regression of a complex epistemic modality. We define the regression of a $\mathbf{PKnows_0}$ expression as follows:

$$\mathcal{R}(\mathbf{PKnows_0}(\pi, \phi, do(c, s))) \stackrel{\text{def}}{=} \forall c' : \mathbf{PKnows_0}(\mathcal{T}(\pi, c, c'), \mathcal{R}(\phi, c'), s)$$

Note that this rule must universally quantify over action terms c' in order to account for different actions producing the same observations. Such quantification is also found in the rule for individual knowledge from equation (7.10), although here it has been taken outside the scope of the knowledge macro.

Theorem 15. *For any epistemic path π, uniform formula ϕ and action c:*

$$\mathcal{D} \cup \mathcal{D}_K^{obs} \models \mathbf{PKnows_0}(\pi, \phi, do(c, s)) \equiv \forall c' : \mathbf{PKnows_0}(\mathcal{T}(\pi, c, c'), \mathcal{R}(\phi, c'), s)$$

Proof Sketch. The mechanics of this proof mirror that of Theorem 11: we expand the $\mathbf{PKnows_0}$ macro, apply Theorem 14 as a successor state axiom for $\mathbf{KDo_0}$, rearrange to eliminate existential quantifiers, then collect terms back into forms that match $\mathbf{PKnows_0}$. For a detailed proof see Appendix A. □

8.4 Introducing Hidden Actions

We now have a powerful account of group-level knowledge for *synchronous* domains, but it remains to generalise this to *asynchronous* domains by incorporating support for arbitrarily-long sequences of hidden actions. We continue to operate at the meta-level, developing support for hidden actions directly in the rules governing the regression operator.

The idea is to use the empty action set to explicitly represent the occurrence of a single hidden action. We simulate agents reasoning about hypothetical futures in which they make no more observations by inserting these empty actions between each regular action in a situation term.

Definition 27. *Let $\mathcal{E}^n(s)$ be s with n empty actions inserted between each action:*

$$\begin{aligned}
\mathcal{E}^0(s) &\stackrel{def}{=} s \\
\mathcal{E}^1(S_0) &\stackrel{def}{=} do(\{\}, S_0) \\
\mathcal{E}^1(do(c, s)) &\stackrel{def}{=} do(\{\}, do(c, \mathcal{E}^1(s))) \\
\mathcal{E}^n(s) &\stackrel{def}{=} \mathcal{E}^1(\mathcal{E}^{n-1}(s))
\end{aligned}$$

The intuition here is that we want $\mathbf{PKnows}(\pi, \phi)$ to hold if $\mathbf{PKnows_0}(\pi, \phi)$ holds after allowing for any number of empty actions. Formally, we will define $\mathbf{PKnows}(\pi, \phi, s)$ to be the following infinite conjunction:

$$\mathbf{PKnows}(\pi, \phi, s) \stackrel{\text{def}}{=} \bigwedge_{n \in \mathbb{N}} \mathbf{PKnows_0}(\pi, \phi, \mathcal{E}^n(s))$$

To avoid an infinite set of sentences we could also use an equivalent second-order definition. The choice is immaterial, since we do not intend to actually expand this definition during reasoning. As before, we will formulate a regression rule allowing it to be treated as a primitive fluent. First, let us demonstrate that this definition is intuitively plausible with the following theorem:

Theorem 16. *For any epistemic path π:*

$$\mathcal{D} \cup \mathcal{D}_K^{obs} \models \mathbf{PKnows_0}(\pi, \phi, \mathcal{E}^1(s)) \rightarrow \mathbf{PKnows_0}(\pi, \phi, s)$$

Proof Sketch. By a case analysis on the definition of $\mathcal{T}$, we determine that that path $\mathcal{T}(\pi, \{\}, \{\})$ always contains the path π. Thus any situations reachable by π are also reachable by $\mathcal{T}(\pi, \{\}, \{\})$. Since $\mathcal{T}(\pi, \{\}, \{\})$ is always in the regression of $\mathbf{PKnows_0}(\pi, \phi, \mathcal{E}^1(s))$ and $\mathcal{R}(\phi, \{\}) = \phi$ always, we can conclude that $\mathbf{PKnows_0}(\pi, \phi, \mathcal{E}^1(s))$ always implies $\mathbf{PKnows_0}(\mathcal{T}(\pi, \{\}, \{\}), \phi, s)$, which implies $\mathbf{PKnows_0}(\pi, \phi, s)$ as required. For a detailed proof see Appendix A. □

The addition of empty actions into a $\mathbf{PKnows_0}$ expression is thus *monotone*. It cannot cause the agents to know things that were not already known, but may cause them to lose knowledge of fluents that can be falsified by a sequence of hidden actions. By arguments entirely analogous to those used in Chapter 6, the addition of empty actions must eventually reach a well-defined fixpoint, and it may be helpful to think of **PKnows** as a fixpoint definition with the following suggestive notation:

$$\mathbf{PKnows}(\pi, \phi, s) \stackrel{\text{def}}{=} \mathbf{PKnows_0}(\pi, \phi, \mathcal{E}^\infty(s))$$

It is this fixpoint nature that will allow us to construct a regression rule using the persistence condition operator. First, let us define an auxiliary meta-level operator $\mathcal{Z}$ that replaces instances of $\mathbf{PKnows_0}$ with **PKnows**:

$$\mathcal{Z}(\phi_1 \wedge \phi_2) \stackrel{\text{def}}{=} \mathcal{Z}(\phi_1) \wedge \mathcal{Z}(\phi_2)$$
$$\mathcal{Z}(\exists x : \phi(x)) \stackrel{\text{def}}{=} \exists x : \mathcal{Z}(\phi(x))$$
$$\mathcal{Z}(\neg\phi) \stackrel{\text{def}}{=} \neg\mathcal{Z}(\phi)$$
$$\mathcal{Z}(\mathbf{PKnows_0}(\pi, \phi, \sigma)) \stackrel{\text{def}}{=} \mathcal{Z}(\mathbf{PKnows}(\pi, \phi, \sigma))$$
$$\mathcal{Z}(\phi) \stackrel{\text{def}}{=} \phi\,, \text{ otherwise}$$

Next, we have a simple action description predicate that is only ever satisfied by the empty action set:

$$Empty(a, s) \equiv a = \{\}$$

We then propose the following regression rules for the macro **PKnows**:

$$\mathcal{R}(\mathbf{PKnows}(\pi, \phi, do(c, s))) \stackrel{\text{def}}{=} \mathcal{Z}(\mathcal{R}(\mathcal{P}(\mathbf{PKnows_0}(\pi, \phi), Empty), c)[s]) \quad (8.3)$$
$$\mathcal{R}(\mathbf{PKnows}(\pi, \phi, S_0)) \stackrel{\text{def}}{=} \mathcal{P}(\mathbf{PKnows_0}(\pi, \phi), Empty)[S_0] \quad (8.4)$$

Here we are using $\mathcal{P}$ with a very restrictive action description predicate to account for an arbitrary number of empty actions. By using $\mathcal{Z}$ to switch from **PKnows$_0$** back to **PKnows** once $\mathcal{P}$ and $\mathcal{R}$ have been applied, we effectively insert an arbitrary number of empty actions between each real action c in the situation term.

Note that, as in the case of individual-level knowledge, regressing **PKnows** at $do(c, s)$ produces an expression using **PKnows** at s, while regressing **PKnows** at S_0 produces an expression involving **PKnows$_0$** at S_0. Repeated applications of these rules will thus reduce a path-knowledge query at some future situation to a synchronous path-knowledge query in the initial situation.

Theorem 17. *Given a basic action theory $\mathcal{D}$ and a uniform formula ϕ:*

$$\mathcal{D} \cup \mathcal{D}_K^{obs} \models \mathbf{PKnows}(\pi, \phi, s) \equiv \mathcal{R}(\mathbf{PKnows}(\pi, \phi, s))$$

Proof Sketch. Proceed by induction on situation terms. For the base case we demonstrate that $\mathcal{P}(\mathbf{PKnows_0}(\pi, \phi), Empty)[S_0]$ is equivalent to the infinite conjunction $\bigwedge_{n\in\mathbb{N}} \mathbf{PKnows_0}(\pi, \phi, \mathcal{E}^n(S_0))$, from which the validity of (8.4) is immediate. To validate (8.3) in the inductive case of $\mathcal{E}^n(do(c, s))$, we demonstrate that the use of $\mathcal{P}$ accounts for the hidden actions inserted *after* c, the use of $\mathcal{R}$ accounts for c itself, and the use of $\mathcal{Z}$ lets the inductive hypothesis account for the hidden actions before c. For a detailed proof see Appendix A. □

These rules for **PKnows** operate in a very similar way to the regression rule for **Knows** from equation (7.10), using a fixpoint calculation to account for arbitrarily long sequences of hidden actions. In the following section we formalise the precise relationship between **Knows** and **PKnows**.

We also note that, as in the previous chapter, our regression rules for knowledge are no longer simple syntactic manipulations but require a meta-level fixpoint calculation to deal with hidden actions. In synchronous domains this fixpoint is unnecessary and **PKnows** is equivalent to $\mathbf{PKnows_0}$, which can be regressed using purely syntactic manipulation. We thus provide an effective reasoning procedure for common knowledge in synchronous domains that is comparable with techniques for handling individual-level knowledge, while also extending the reach of our formalism into richer domains where some inductive reasoning is required.

Theorem 18. *Let $\mathcal{D}_{sync}$ be a synchronous basic action theory, then:*

$$\mathcal{D}_{sync} \cup \mathcal{D}_K^{obs} \models \forall s : \mathbf{PKnows}(\pi, \phi, s) \equiv \mathbf{PKnows_0}(\pi, \phi, s)$$

Proof Sketch. It suffices to show that:

$$\mathcal{D}_{sync} \cup \mathcal{D}_K^{obs} \models \mathbf{PKnows_0}(\pi, \phi, s) \rightarrow \mathbf{PKnows_0}(\pi, \phi, \mathcal{E}^1(s))$$

Then by Theorem 16 we have that $\mathbf{PKnows_0}(\pi, \phi, s)$ is enough to establish $\mathbf{PKnows_0}(\pi, \phi, \mathcal{E}^n(s))$ for any n, which establishes the infinite conjunction in the definition of $\mathbf{PKnows}(\pi, \phi, s)$ as required. The regression of $\mathbf{PKnows_0}(\pi, \phi, \mathcal{E}^1(s))$ contains the modality $\mathcal{T}(\pi, \{\}, c')$, and we demonstrate that since the domain is synchronous it is not possible for the regressed path to exit successfully unless $c' = \{\}$. Showing that $\mathcal{T}(\pi, \{\}, \{\})$ is equivalent to π in synchronous domains completes the proof. For a detailed proof see Appendix A. □

8.5 The Link with Individual Knowledge

The last remaining link is the most important of all: showing that this new path-based account of knowledge actually captures the knowledge of the agents, according to the semantics of individual knowledge developed in Chapter 7. The following series of theorems establish this important link.

Lemma 4. *For any agt and ϕ:*

$$\mathcal{D} \cup \mathcal{D}_K^{obs} \models \mathbf{PKnows}(agt, \phi, S_0) \equiv \mathbf{PKnows_0}(agt, \mathcal{P}(\phi, LbU(agt)), S_0)$$

Proof Sketch. Begin by applying equation (8.4) to the LHS to get an expression in $\mathbf{PKnows_0}$. By stepping through the regression of $\mathbf{PKnows_0}(agt, \phi, do(\{\}, S_0))$ we show that for any value of n, $\mathcal{P}^n(\mathbf{PKnows_0}(agt, \phi,), Hidden)[S_0]$ is equivalent to $\mathbf{PKnows_0}(agt, \mathcal{P}^n(\phi, LbU(agt)), S_0)$. The fixpoint calculation for $\mathcal{P}$ thus terminates at the same value of n in both cases, giving the required equivalence. For a detailed proof see Appendix A. □

Lemma 5. *For any agt, ϕ, c and s:*

$$\begin{aligned}\mathbf{PKnows}(agt, \phi, do(c, s)) \equiv\ & \exists z : Obs(agt, c, s) = z \\ & \wedge\ [z = \{\} \rightarrow \mathbf{PKnows}(agt, \mathcal{P}(\phi, LbU(agt)), s)] \\ & \big[z \neq \{\} \rightarrow \mathbf{PKnows}(agt, \forall c' : \big(Legal(c') \wedge Obs(agt, c') = z\big) \\ & \qquad \rightarrow \mathcal{R}(\mathcal{P}(\phi, LbU(agt)), c), s)]\end{aligned}$$

Proof Sketch. We apply equation (8.4) to get an expression in $\mathbf{PKnows_0}$. Repeating the calculations from Lemma 4 gives us the required $\mathcal{P}$ expression, and regressing this over c gives us the required $\mathcal{R}$ expression. Using $\mathcal{Z}$ then allows us to transform from $\mathbf{PKnows_0}$ back to **PKnows** to obtain the desired result. For a detailed proof see Appendix A. □

Lemma 6. *For any agt, ϕ and s:*

$$\mathcal{D} \cup \mathcal{D}_K^{obs} \models \mathbf{Knows}(agt, \phi, s) \equiv \mathbf{Knows}(agt, \mathcal{P}(\phi, LbU(agt)), s)$$

Proof Sketch. By induction on situations and using the regression rules for knowledge, along with the following property of the persistence condition ("if ϕ persists, then $\mathcal{P}(\phi, \alpha)$ persists"):

$$\forall s' : \mathcal{P}(\phi, \alpha)[s] \wedge s \leq_\alpha s' \rightarrow \mathcal{P}(\phi, \alpha)[s']$$

We consider three cases: $s = S_0$, and $s = do(c, s)$ with c both observable and unobservable. Each case requires only a simple re-arrangement of the relevant regression rule. For a detailed proof see Appendix A. □

Theorem 19. *For any agt, ϕ and s:*

$$\mathcal{D} \cup \mathcal{D}_K^{obs} \models \mathbf{Knows}(agt, \phi, s) \equiv \mathbf{PKnows}(agt, \phi, s)$$

Proof. By induction on situation terms, using the regression rules for each expression. For the S_0 case, we require the following result which is true by the definition

of $\mathbf{KDo_0}$:

$$\mathbf{Knows_0}(agt, \phi, S_0) \equiv \mathbf{PKnows_0}(agt, \phi, S_0)$$

The regression rule for **Knows** from equation (7.11) then precisely matches the result of Lemma 4 and we have the required equivalence. For the $do(c, s)$ case, we can substitute the result of Lemma 6 into the regression rule for **Knows** from equation (7.10) to produce an expression precisely matching the result of Lemma 5. Using $\mathbf{Knows}(agt, \phi, s) \equiv \mathbf{PKnows}(agt, \phi, s)$ from the inductive hypothesis renders the two equivalent. □

Thus the expressions $\mathbf{Knows}(agt, \phi, s)$ and $\mathbf{PKnows}(agt, \phi, s)$ are equivalent under our formulation. This link is all that is required to validate the additional complex modalities shown in equation (8.2), repeated below for convenience.

Theorem 20. *The following identities hold under the theory of action $\mathcal{D} \cup \mathcal{D}_K^{obs}$:*

$$\begin{aligned}
\mathbf{Knows}(A, \phi, s) &\equiv \mathbf{PKnows}(A, \phi, s) \\
\mathbf{Knows}(A, \mathbf{Knows}(B, \phi), s) &\equiv \mathbf{PKnows}(A; B, \phi, s) \\
\mathbf{Knows}(A, \phi, s) \wedge \mathbf{Knows}(B, \phi, s) &\equiv \mathbf{PKnows}(A \cup B, \phi, s) \\
\mathbf{EKnows}(G, \phi, s) &\equiv \mathbf{PKnows}(\bigcup_{a \in G} a, \phi, s) \\
\mathbf{CKnows}(G, \phi, s) &\equiv \mathbf{PKnows}((\bigcup_{a \in G} a)^*, \phi, s)
\end{aligned}$$

Proof. Each follows from equivalence of $\mathbf{Knows}(agt, \phi, s)$ and $\mathbf{PKnows}(agt, \phi, s)$, using the semantics of first-order dynamic logic as defined by $\mathbf{KDo_0}$. For example, in the **CKnows** case we argue as follows: By definition, $\mathbf{PKnows}((\bigcup_{a \in G} a)^*, \phi, s)$ is the transitive closure of the union of $\mathbf{PKnows}(a, \phi, s)$ for $a \in G$. Also by definition, $\mathbf{CKnows}(G, \phi, s)$ is the transitive closure of the union of $\mathbf{Knows}(a, \phi, s)$ for $a \in G$. Since $\mathbf{PKnows}(a, \phi, s)$ and $\mathbf{Knows}(a, \phi, s)$ are the same relation, their transitive closures are also the same and the identity is entailed. □

Let us briefly return to the initial premise of this chapter: that the regression of common knowledge cannot be expressed in terms of common knowledge alone. Since Theorems 15 and 17 and equivalences, we have shown that the operation of $\mathcal{T}$ captures precisely the information needed to regress common knowledge. The path resulting from $\mathcal{T}((A \cup B)^*, c, c')$ will contain various test conditions and variable re-bindings *inside* the scope of the iteration. Such a path is clearly beyond the expressive power of the ordinary common knowledge operator, providing a satisfying

confirmation of our intuitions that the results of Batlag et al. [8] would apply to the situation calculus.

8.6 An Illustrative Example

With this technical machinery in place, we are now able to reason about the group-level epistemic modalities of a team of agents. To demonstrate we revisit our example domain from the previous chapter, in which Alice and Bob have received an invitation to a party. To keep the presentation simple we will assume that actions cannot be performed concurrently. We add the the following initial knowledge axiom to $\mathcal{D}_{S_0}$:

$$\mathbf{PKnows_0}((A \cup B)^*, InRoom(A) \wedge InRoom(B), S_0)$$

The preconditions and effects of actions in this domain are relatively simple, such that there is nothing that can be accomplished by a sequence of hidden actions that cannot be accomplished by a single hidden action. More formally, it is possible to show that in this domain:

$$\bigwedge_{n \in \mathbb{N}} \phi[\mathcal{E}^n(s)] \equiv \phi[\mathcal{E}^1(s)]$$

In other words, the fixpoint calculation required for reasoning about **PKnows** will always terminate after a single iteration. We will therefore use the following identity to simplify presentation of the example:

$$\begin{aligned}\mathcal{R}(\mathbf{PKnows}(\pi, \phi, do(c, s)) &= \mathcal{Z}(\mathcal{R}(\mathcal{P}(\mathbf{PKnows_0}(\pi, \phi), Hidden)[do(c, s)])) \\ &\equiv \mathcal{Z}(\mathcal{R}(\mathcal{R}(\mathbf{PKnows_0}(\pi, \phi, \mathcal{E}^1(do(c, s)))))) \\ &\equiv \forall c', c'' : \mathbf{PKnows}(\mathcal{T}(\mathcal{T}(\pi, \{\}, c'), c, c''), \mathcal{R}(\mathcal{R}(\phi, c'), c''), s)\end{aligned}$$

To keep the presentation compact, we abbreviate the fluent $InRoom$ to IR.

Example 5. *After Bob reads the invitation, it is common knowledge that he knows where the party is:*

$$\mathcal{D} \cup \mathcal{D}_K^{obs} \models \mathbf{PKnows}((A \cup B)^*, \exists x : \mathbf{Knows}(B, loc = x), do(read(B), S_0))$$

This example hinges on the fact that initially, it is common knowledge that both agents are in the room. It is thus common knowledge that the occurrence of $read(Bob)$ will be observed by both agents. Suppressing the inner expression for the

moment, regressing using the above-mentioned identity gives:

$$\begin{aligned}
&\mathcal{R}(\mathbf{PKnows}((A \cup B)^*, \phi, do(read(B), S_0))) \\
&\quad = \forall c, c' : \mathbf{PKnows}(\mathcal{T}(\mathcal{T}((A \cup B)^*, \{\}, c), read(B), c'), \mathcal{R}(\mathcal{R}(\phi, c), c'), S_0) \\
&\quad\quad = \forall c, c', c'' : \mathbf{PKnows_0}(\mathcal{T}(\mathcal{T}(\mathcal{T}((A \cup B)^*, \{\}, c), read(B), c'), \{\}, c''), \\
&\quad\quad\quad\quad \mathcal{R}(\mathcal{R}(\mathcal{R}(\phi, c), c'), c''), S_0)
\end{aligned}$$

We begin by evaluating $\mathcal{T}(\pi, \{\}, c)$:

$$\mathcal{T}((A \cup B)^*, \{\}, c) = \; x \Leftarrow \{\} \,;\, (\mathcal{T}_a(A, x) \cup \mathcal{T}_a(B, x))^* \,;\, ?x = c$$

With the cases for $\mathcal{T}_a(agt, x)$ both evaluating to:

$$\mathcal{T}_a(agt, x) = \; z \Leftarrow Obs(agt, x) \,;\, agt \,;\, \exists x \,;\, ?Legal(x) \vee x = \{\} \,;\, ?Obs(agt, x) = z$$

Since we know all possible observations that Alice could make, let us expand the path $\mathcal{T}_a(A, x)$ by enumerating the possible values for z. The first line in the result gives the case where $Obs(Alice, x) = \{\}$, and the last line is the case where $Obs(Alice, x) = read(Alice)\#r$. All other lines are where Alice simply observes the action x.

$$\begin{aligned}
&\mathcal{T}_a(A, x) = \\
&\quad ?\, x = \{\} \vee (x = read(B) \wedge \neg IR(A))) \,;\, A \,; \\
&\qquad\qquad \exists x \,;\, ?\, x = \{\} \vee (x = read(B) \wedge IR(B) \wedge \neg IR(A)) \\
&\qquad \cup\, ?x = enter(A) \,;\, A \,;\, \exists x \,;\, ?x = enter(A) \wedge IR(A) \\
&\qquad \cup\, ?x = enter(B) \,;\, A \,;\, \exists x \,;\, ?x = enter(B) \wedge IR(B) \\
&\qquad \cup\, ?x = leave(A) \,;\, A \,;\, \exists x \,;\, ?x = leave(A) \wedge \neg IR(A) \\
&\qquad \cup\, ?x = leave(B) \,;\, A \,;\, \exists x \,;\, ?x = leave(B) \wedge \neg IR(B) \\
&\quad \cup\, ?\, x = read(B) \wedge IR(A) \,;\, A \,;\, \exists x \,;\, ?\, x = read(B) \wedge IR(A) \wedge IR(B) \\
&\cup\, \exists r \,;\, ?\, x = read(A) \wedge loc(r) \,;\, A \,;\, \exists x \,;\, ?\, x = read(A) \wedge IR(A) \wedge loc(r)
\end{aligned}$$

Now consider starting with x set to $\{\}$ and following this path some number of times. On the first iteration, x may get rebound to either $\{\}$ or $read(B)$. Subsequent iterations may alternate between these values, but can never bind x to any other action. An analogous argument for $\mathcal{T}_a(B, x)$ shows that the path $\mathcal{T}((A \cup B)^*, \{\}, c)$ can be simplified to consider only the actions $\{\}$, $read(A)$ and $read(B)$, with any

other binding of x being unreachable:

$$\begin{aligned}
\mathcal{T}((A \cup B)^*, \{\}, c) = \quad & x \Leftarrow \{\} \,;\, (\\
& ?\, x = \{\} \vee (x = read(B) \wedge \neg IR(A)))\,;\, A\,; \\
& \qquad \exists x\,;\, ?\, x = \{\} \vee (x = read(B) \wedge IR(B) \wedge \neg IR(A)) \\
& \cup ?\, x = \{\} \vee (x = read(A) \wedge \neg IR(B)))\,;\, B\,; \\
& \qquad \exists x\,;\, ?\, x = \{\} \vee (x = read(A) \wedge IR(A) \wedge \neg IR(B)) \\
& \cup ?\, x = read(B) \wedge IR(A)\,;\, A\,;\, \exists x\,;\, ?\, x = read(B) \wedge IR(A) \wedge IR(B) \\
& \cup \exists r\,;\, ?\, x = read(A) \wedge loc(r)\,;\, A\,;\, \exists x\,;\, ?\, x = read(A) \wedge IR(A) \wedge loc(r) \\
& \cup ?\, x = read(A) \wedge IR(B)\,;\, B\,;\, \exists x\,;\, ?\, x = read(A) \wedge IR(B) \wedge IR(A) \\
& \cup \exists r\,;\, ?\, x = read(B) \wedge loc(r)\,;\, B\,;\, \exists x\,;\, ?\, x = read(B) \wedge IR(B) \wedge loc(r) \\
& \qquad)^*\,;\, ?x = c
\end{aligned}$$

This path matches with our intuitions about the observability of actions. Since the *enter* and *leave* actions are always public, they can never be mistaken for the empty action and are irrelevant in the path $\mathcal{T}(\pi, \{\}, c)$. Moreover, the binding of x may only switch from $\{\}$ to $read(B)$ if there is a possible world in which Alice is not in the room, reflecting the conditions under which that action would be hidden.

Next we evaluate the $\mathcal{T}(\pi, read(B), c')$ portion, which will produce:

$$\begin{aligned}
\mathcal{T}(\mathcal{T}((A \cup B)^*, \{\}, c), read(B), c') = \quad & x' \Leftarrow read(B)\,;\, x \Leftarrow \{\}\,; \\
& (\mathcal{T}_a(\mathcal{T}_a(A, x), x') \cup \mathcal{T}_a(\mathcal{T}_a(B, x), x'))^*\,;\, ?x = c\,;\, ?x' = c'
\end{aligned}$$

As before, the only possible values of x' are $\{\}$, $read(A)$ and $read(B)$. None of these actions affect the tests present in $\mathcal{T}((A \cup B)^*, \{\}, c)$, so we leave them unchanged. Expanding the applications of $\mathcal{T}_a$ gives the following inside the iteration operator:

$$\begin{aligned}
& ?x = \{\} \vee (x = read(B) \wedge \neg IR(A)))\,;\, \mathcal{T}_a(A, x')\,; \\
& \qquad \exists x\,;\, ?x = \{\} \vee (x = read(B) \wedge IR(B) \wedge \neg IR(A)) \\
& \cup ?x = \{\} \vee (x = read(A) \wedge \neg IR(B)))\,;\, \mathcal{T}_a(B, x')\,; \\
& \qquad \exists x\,;\, ?x = \{\} \vee (x = read(A) \wedge IR(A) \wedge \neg IR(B)) \\
& \cup ?\, x = read(B) \wedge IR(A)\,;\, \mathcal{T}_a(A, x')\,;\, \exists x\,;\, ?\, x = read(B) \wedge IR(A) \wedge IR(B) \\
& \cup \exists r\,;\, ?\, x = read(A) \wedge loc(r)\,;\, \mathcal{T}_a(A, x')\,;\, \exists x\,;\, ?\, x = read(A) \wedge IR(A) \wedge loc(r) \\
& \cup ?\, x = read(A) \wedge IR(B)\,;\, \mathcal{T}_a(B, x')\,;\, \exists x\,;\, ?\, x = read(A) \wedge IR(B) \wedge IR(A) \\
& \cup \exists r\,;\, ?\, x = read(B) \wedge loc(r)\,;\, \mathcal{T}_a(B, x')\,;\, \exists x\,;\, ?\, x = read(B) \wedge IR(B) \wedge loc(r)
\end{aligned}$$

We can complete this expansion using the previously calculated result for $\mathcal{T}_a(A, x)$, but will proceed leaving this implicit. Finally, we must apply $\mathcal{T}$ again for the $\mathcal{T}(\pi, \{\}, c'')$ portion. The process is identical and we will not repeat it here.

We next evaluate this path over the initial epistemic relation of the agents, about which we know:

$$\mathbf{PKnows_0}((A \cup B)^*, IR(A) \wedge IR(B), S_0)$$

Consider how this restricts the possible bindings of x, x' etc. For x to switch from its initial value of $\{\}$ to $read(A)$ (resp. $read(B)$) the path must traverse a test for $\neg IR(B)$ (resp. $\neg IR(A)$). Since we know that these tests will never succeed on the initial epistemic frame, we can conclude that x will always be bound to $\{\}$. Since the path insists that $x = c$ at termination, this is the only interesting value for c – any value of c other than $\{\}$ will result in no worlds being reachable by the regressed path and will thus have $\mathbf{PKnows_0}$ vacuously true. By a similar argument, we find that the only interesting values of c' and c'' are $read(B)$ and $\{\}$ respectively.

Turning now to the regression of the inner formula, we have:

$$\begin{aligned}
&\mathcal{R}(\mathcal{R}(\mathcal{R}(\exists x : \mathbf{Knows}(B, loc = x), c), c'), c'') \\
&= \mathcal{R}(\mathcal{R}(\mathcal{R}(\exists x : \mathbf{Knows}(B, loc = x), \{\}), read(B)), \{\}) \\
&= \mathcal{R}(\mathcal{R}(\exists x : \mathbf{Knows}(B, loc = x), read(B), \{\}) \\
&= true
\end{aligned}$$

Thus for $c = \{\}$, $c' = read(B)$, $c'' = \{\}$ the known formula is a tautology, while for any other values the regressed epistemic path has no reachable worlds. The example is therefore entailed by the domain.

Example 6. *After Bob reads the invitation, the location is not common knowledge*

$$\mathcal{D} \cup \mathcal{D}_K^{obs} \not\models \mathbf{PKnows}((A \cup B)^*, loc = C, do(read(B), S_0))$$

Regression of the epistemic path proceeds as with the previous example, but regression of the inner formula no longer produces a tautology. If we ignore the path components dealing with c and c'', which from the previous example we know to be redundant, regressing this expression produces the following:

$$\begin{aligned}&\mathbf{PKnows_0}(\ldots;(\\&\quad\cup\,?\,x' = read(B) \wedge IR(A)\,;\,A\,;\,\exists x'\,;\,?\,x' = read(B) \wedge IR(A) \wedge IR(B)\\&\cup\;\exists r\,;\,?\,x' = read(B) \wedge loc(r)\,;\,B\,;\,\exists x'\,;\,?\,x' = read(B) \wedge IR(B) \wedge loc(r)\\&\qquad\qquad\cup\ldots)^*;\ldots,loc = C, S_0)\end{aligned}$$

This expression basically asserts that if a situation in the initial epistemic frame can be reached by only traversing situations in which $IR(A)$ and $IR(B)$ holds, then that situation must entail that $loc = C$. In the previous example, the inner formula was a simple tautology and this was vacuously true.

In this case (and very roughly speaking) the regressed path indicates that $read(B)$ can potentially result in common knowledge of the party's location, but only if having common knowledge of the preconditions of $read(B)$ is enough to establish common knowledge that $loc = C$. Since no such information is given in the domain description, the example is not entailed.

Example 7. *After Alice also reads the invitation, the location is common knowledge:*

$$\mathcal{D} \cup \mathcal{D}_K^{obs} \models \mathbf{PKnows}((A \cup B)^*, loc = C, do(read(A), do(read(B), S_0)))$$

The additional action here can be handled in the same manner as the previous examples, but the paths quickly become too cumbersome to present on the page. We will not repeat the working, but simply state that this is indeed a valid consequence of our theory. Note, however, that it depends on having common knowledge that both agents are in the room.

8.7 Comparison with LCC

Having constructed our account of complex epistemic modalities in the situation calculus, let us now reflect on some of the similarities and differences between our approach and the Logic of Communication and Change of van Benthem et al. [119].

The most obvious difference is that our formalism supports - indeed, *must* support - first-order preconditions and effects and is not limited to finite domains. By contrast, LCC is propositional and actions must be explicitly specified in terms of a finite update model. This is for the most part simply reflective of the different traditions from which these formalisms emerge; of the underlying differences between

modal logics of action and the situation calculus [117]. It is likely that LCC could be expanded into a first-order modal formalism without significant modification. We note, however, that the macro-based approach of the situation calculus automatically settles the various semantic issues that can complicate first-order modal logics: quantifying into modalities; de-dicto vs de-re; rigid vs monotonic vs arbitrary domains; etc [32].

Another important difference is that in the situation calculus, actions are concrete terms in the logic that can be manipulated and quantified over. In LCC, the update models responsible for changing the world come from outside the world itself, and must be specified as an explicit structure whenever they are used. They cannot be quantified over. This difference is key to our handling of asynchronous domains, by allowing agents to consider arbitrarily-long sequences of hidden actions. While LCC is capable of representing hidden actions, the agents cannot account for them and they cause knowledge to degenerate into belief [119, section 5.2].

Our formalism also has the advantage that it can be used by a situated agent to reason about its own knowledge using its local view v, including reasoning about common knowledge using a query such as:

$$\mathcal{D} \cup \mathcal{D}_K^{obs} \models \mathbf{Knows}(agt, \mathbf{PKnows}((\bigcup_{a \in G} a)^*, \phi)), v)$$

This ability comes from having views reified as first-order terms. While LCC provides powerful mechanisms for reasoning *about* multi-agent knowledge, it does not appear to be suitable for reasoning *in* multi-agent domains.

For a more detailed comparison, consider the operation of the program transformer T_{ij}^U of LCC. First, we have the reduction rule for epistemic programs:

$$[U, e_i][\pi]\phi \Rightarrow \bigwedge_{j=0}^{n-1} [T_{ij}^U(\pi)][U, e_j]\phi$$

Since the update model U is finite, this rule can construct a finite conjunction of the possible actions $j = 0$ through $j = n - 1$, and separately consider the conditions under which action i would be mistaken for action j. Our regression rule lifts this to the first-order case by quantifying over all actions:

$$\mathcal{R}(\mathbf{PKnows_0}(\pi, \phi, do(c, s))) \stackrel{\text{def}}{=} \forall c' : \mathbf{PKnows_0}(\mathcal{T}(\pi, c, c'), \mathcal{R}(\phi, c'), s)$$

If the ACTION sort were finite, we could replace this quantifier with a conjunction to get essentially the same rule as LCC. Next, consider the definition of the T_{ij}^U

transformer from LCC:

$$T^U_{ij}(agt) \stackrel{\text{def}}{=} \begin{cases} ?pre(e_i); a & if\ e_i R(agt) e_j \\ ?\bot & otherwise \end{cases}$$

$$T^U_{ij}(?\phi) \stackrel{\text{def}}{=} \begin{cases} ?pre(e_i) \wedge [U, e_i]\phi & if\ i = j \\ ?\bot & otherwise \end{cases}$$

$$T^U_{ij}(\pi_1; \pi_2) \stackrel{\text{def}}{=} \bigcup_{k=0}^{n-1} (T^U_{ik}(\pi_1); T^U_{kj}(\pi_2))$$

$$T^U_{ij}(\pi_1 \cup \pi_2) \stackrel{\text{def}}{=} T^U_{ij}(\pi_1) \cup T^U_{ij}(\pi_1)$$

$$T^U_{ij}(\pi^*) \stackrel{\text{def}}{=} K^U_{ijn}(\pi)$$

Apart from the π^* case, which we will discuss separately, these definitions are broadly similar to the rules for our epistemic path regressor from Definition 26. Where our rules use a fresh variable to track the "current action" being performed in each situation, these rules use external indices i and j and enumerate each of the finitely-many actions. The case for a base agt modality encodes the possibility of that agent mistaking event e_i for event e_j, which is represented explicitly by the relation $R(agt)$ from the update model. The case for $\pi_1; \pi_2$ uses a finite disjunction to permit any intermediate event k.

The π^* case in LCC is handled by appealing to a variant of Kleene's construction on finite automata, whereby all paths through the automata are enumerated by progressively including more and more states [63, Theorem 2.5.1]. The auxiliary operator $K^U_{ijk}(\pi)$ represents all the paths from i to j through the update model U that can be generated by following π zero or more times, but without stopping at intermediate actions numbered greater than k:

$$K^U_{ij0}(\pi) \stackrel{\text{def}}{=} \begin{cases} ?\top \cup T^U_{ij} & if\ i = j \\ ?T^U_{ij} & otherwise \end{cases}$$

$$K^U_{ij(k+1)}(\pi) \stackrel{\text{def}}{=} \begin{cases} (K^U_{kkk}(\pi))^* & if\ i = j = k \\ (K^U_{kkk}(\pi)^*); K^U_{kjk}(\pi) & if\ i = j \neq k \\ K^U_{ikk}(\pi); (K^U_{kkk}(\pi))^* & if\ i \neq k = j \\ K^U_{ijk}(\pi) \cup (K^U_{ikk}(\pi); (K^U_{kkk}(\pi))^*; K^U_{kjk}(\pi)) & otherwise \end{cases}$$

The definition $T^U_{ij}(\pi^*) \stackrel{\text{def}}{=} K^U_{ijn}(\pi)$ thus permits paths using any of the n possible

intermediate events. The proof that T_{ij}^U works as required then requires several lemmas that establish the validity of this K_{ijk}^U construction.

Contrast this construction with our own rule for handling the π^* case:

$$\mathcal{T}_a(\pi^*, x) \stackrel{\text{def}}{=} \mathcal{T}_a(\pi, x)^*$$

Since we are using first-order dynamic logic, we do not need to explicitly enumerate all possible intermediate states reached by following π^*. We argue that our formulation, using a variable to keep track of the "current action" in a regressed epistemic path, is a more natural expression of the intended operation of the path regression operator. Our correctness proofs are correspondingly simpler. The more complicated Kleene-style operator of LCC is required in order to render epistemic path regression into a propositional formalism.

In synchronous domains with a finite state-space the situation calculus may not offer a gain in expressiveness, as these domains could be propositionalised and formulated in LCC. However, it can certainly provide a more succinct axiomatisation. Moving beyond such domains, our formalism allows hidden actions to be considered and also offers the potential to incorporate other rich domain features that have been developed for the situation calculus, such as continuous time and actions with duration.

8.8 Answering the Regressed Query

We are now in a position to reduce a complex epistemic query $\mathbf{PKnows}(\pi, \phi, \sigma)$ at some future situation to an epistemic query $\mathbf{PKnows_0}(\mathcal{T}^*(\pi), \mathcal{R}^*(\phi), S_0)$ at the initial situation. While this is a significant gain for effective automated reasoning, it still remains to calculate and answer the regressed query.

As with individual knowledge, we assume that queries will be handled by a special-purpose modal theorem prover rather than by expanding the macros. In order to calculate the persistence conditions in regression rules (8.3,8.4), we require s*tatic domain reasoning* about $\mathbf{PKnows_0}$ modalities. Likewise, to answer the regressed query, one must perform *initial situation reasoning* about $\mathbf{PKnows_0}$ modalities. While we have gained the ability to use proper regression rules, we have moved from the semi-decidable case of purely first-order reasoning to the undecidable case of reasoning in FODL, which is known to be Π_1^1-hard in general [49].

Facing the potential for undecidability is nothing new for the situation calculus, and we proceed in the tradition of previous work: by identifying domain restrictions that enable more effective reasoning.

As discussed in Sections 2.1.3 and 6.4.3, one of the best ways to gain decidability in the situation calculus is to assume that non-situation sorts such as ACTION and OBJECT are finitely enumerable. This allows the domain to be "propositionalised" and queries to be answered using propositional logic. Will the same restriction allow us to gain decidability for our complex epistemic modalities?

Since we do not use SITUATION terms in epistemic paths, it seems we should be able to translate $\mathbf{PKnows_0}$ modalities in such domains from FODL into PDL by enumerating the possible values for each path variable. This is not straightforward, however, since variable re-bindings can be performed inside the scope of an iteration. It is not obvious how to convert such a path into PDL.

Here we close the loop from our first-order formalism back to the propositional formalism of LCC – the Kleene-style construction used to regress π^* paths in LCC is precisely what is required to translate FODL into PDL and gain decidability.

Suppose the epistemic path π contains n path variables, then let the *state* of that path be a vector of length n giving the current value of each variable. Since variable domains are finite, we can finitely enumerate all possible states of the path. Let them be labelled 0 through m, and let ϕ^i represent the formula obtained from ϕ by replacing all path variables with their values in state i. Let $\phi^{\downarrow}$ represent the standard propositionalisation of first-order formula ϕ. A FODL modality $\mathbf{PKnows_0}(\pi, \phi)$ can be translated into a PDL modality $\mathbf{Prop}[\mathbf{PKnows_0}(\pi, \phi)]$ as follows. $\mathbf{Prop}_{ij}[\pi]$ represents the propositionalisation of path π with start state i and end state j:

$$\mathbf{Prop}[\mathbf{PKnows_0}(\pi, \phi)] \stackrel{\text{def}}{=} \bigwedge_{i=0}^{m} \bigwedge_{j=0}^{m} \mathbf{PKnows_0}(\mathbf{Prop}_{ij}[\pi], \phi^{\downarrow})$$

$$\mathbf{Prop}_{ij}[agt] \stackrel{\text{def}}{=} \begin{cases} agt & \text{if i} = \text{j} \\ ?\bot & \text{otherwise} \end{cases}$$

$$\mathbf{Prop}_{ij}[?\phi] \stackrel{\text{def}}{=} \begin{cases} ?\phi^{i\downarrow} & \text{if i} = \text{j} \\ ?\bot & \text{otherwise} \end{cases}$$

$$\mathbf{Prop}_{ij}[\exists x_n] \stackrel{\text{def}}{=} \begin{cases} ?\top & \text{if i, j differ only at n} \\ ?\bot & \text{otherwise} \end{cases}$$

$$\mathbf{Prop}_{ij}[\pi_1; \pi_2] \stackrel{\text{def}}{=} \bigcup_{k=0}^{m} (\mathbf{Prop}_{ik}[\pi_1]; \mathbf{Prop}_{kj}[\pi_2])$$

$$\mathbf{Prop}_{ij}[\pi_1 \cup \pi_2] \stackrel{\text{def}}{=} \mathbf{Prop}_{ij}[\pi_1] \cup \mathbf{Prop}_{ij}[\pi_2]$$
$$\mathbf{Prop}_{ij}[\pi^*] \stackrel{\text{def}}{=} \mathbf{PropK}_{ijm}(\pi)$$

$$\mathbf{PropK}_{ij0}[\pi] \stackrel{\text{def}}{=} \begin{cases} ?\top \cup \mathbf{Prop}_{ij}[\pi] & \text{if i} = \text{j} \\ \mathbf{Prop}_{ij}[\pi] & \text{otherwise} \end{cases}$$
$$\mathbf{PropK}_{ij(k+1)}[\pi] \stackrel{\text{def}}{=} \mathbf{PropK}_{ijk}[\pi] \cup (\mathbf{PropK}_{ikk}[\pi]; \mathbf{PropK}_{kkk}[\pi]^*; \mathbf{PropK}_{kjk}[\pi])$$

While this demonstrates that finite domains are decidable in principle, it should not be used as an implementation technique in practice, as it will produce an exponential blowup in the size of the query. We will return to this point in Section 8.9.

To answer the regressed query in the initial situation, we may be able to assume that initial knowledge is specified in a restricted form that further increases the effectiveness of reasoning. For example, it may be that the initial situation is completely known and uncertainty is introduced only due to partial observability of actions. In this case the initial epistemic frame contains the lone situation S_0, and the regressed path can be reduced to a series of tests and variable re-bindings.

More generally, we may have a specific possible-worlds model to represent the group's knowledge in the initial situation. In this case the regressed query can be answered using model-checking rather than satisfiability; for PDL this reduces the complexity from EXPTIME-complete to PTIME-complete, a significant win [53].

It may also be possible to identify restricted fragments of FODL that are sufficient to capture the regression of common knowledge in restricted domains. For example, van Benthem et al. [119] show that in domains with public actions the regression of common knowledge can be expressed using a relativised common-knowledge operator, rather than requiring the full expressivity of dynamic logic. Investigating whether this also holds for the situation calculus could be a launching point for future work in this direction.

We have produced a preliminary reasoning engine based on our technique for propositional domains, utilising a modified version of the PDL prover from the Tableaux Workbench suite [1], and have used it to verify some simple examples. However, due to the exponential blowup in path size using the propositionalisation scheme described above, we have so far not validated the examples from Section 8.6 with this implementation. As usual, details on obtaining the source code can be

found in Appendix B.

8.9 Discussion

In this chapter we have introduced a technique for representing and reasoning about complex epistemic modalities in the situation calculus. In order to formulate an effective reasoning procedure, we have had to move beyond just **Knows** and **CKnows** and introduce a powerful epistemic path language based on dynamic logic. Mirroring the development of knowledge for individual agents, group-level modalities are introduced as macros of the form $\mathbf{PKnows}(\pi, \phi, s)$ where π is a complex epistemic path. To avoid having to expand these macros during reasoning, we have modified the regression operator to treat them as primitive fluents.

Our development was inspired by and has clear parallels with the Logic of Communication and Change of van Benthem et al. [119]. We choose the situation calculus for its much richer ontology, e.g. preconditions and effects are first order, while actions take arguments and may be performed concurrently. On one hand, this forces us to use a more powerful dynamic logic for our epistemic language and run the risk of undecidability. On the other, it actually simplifies some aspects of our presentation. We do not need explicit update frames, and the definition of our path regressor does not require an auxiliary Kleene-style operator to handle iteration.

As with the previous chapter, hidden actions are handled using the persistence condition meta-operator to perform a fixpoint calculation. While the expressiveness of our epistemic path language means that answering a regressed knowledge query can be difficult in the general case, we have shown that for finite domains we maintain the ability to propositionalise the queries and produce a robustly decidable theory.

Demonstrating the utility of our approach, we have presented an example of reasoning about common knowledge in an asynchronous, partially observable domain. This powerful new ability is a first for the situation calculus.

Our synchronous path-knows operator $\mathbf{PKnows_0}$ is also significant new contribution in its own right. For synchronous domains this operator is provably equivalent to **PKnows** and allows common knowledge to be handled using a purely syntactic transformation, rather than requiring a fixpoint calculation to deal with hidden actions. This chapter thus provides powerful new reasoning techniques for group-level knowledge that are comparable in complexity to the standard account of epistemic reasoning in synchronous domains, while also extending its reach into asynchronous domains where some inductive reasoning is required.

As discussed in Section 8.8, reasoning in our new epistemic language can be very expensive in general. Although we have demonstrated that it can be reduced to PDL in finite domains, naive use of this reduction is not suitable for automated reasoning in any but the simplest domains. Like the reduction of first-order logic into propositional logic, it results in an exponential blowup in the size of the formula.

In standard first-order logic this is avoided using a *free-variable* prover which can avoid generating cases that will obviously not lead to a proof. Effective automated reasoning in our epistemic language would seem to require similar use of free variables to control this exponential blowup. We are aware of no existing systems that could easily support this reasoning.

We have the beginnings of a free-variable prover for our logic, using a combination of the first-order Shannon Graph techniques of [87] and the PDL tableaux rules described in [1], but there is substantial work still to be done towards this goal.

There is also an important open question left by our work: does FODL *precisely* characterise the expressivity required to regress common knowledge? This chapter has provided an upper bound on the required expressivity by formulating a sound and complete regression rule for FODL knowledge modalities. It is also clear that the standard **Knows** and **CKnows** modalities do not have sufficient expressivity. But is there a language between these two bounds that is rich enough to regress common knowledge, while having a more tractable reasoning procedure than full dynamic logic?

Our epistemic path regressor constructs paths with a very regular internal structure, so the possibility cannot be ruled out at this stage. For example, we note that when starting from the expression for common knowledge, $\mathcal{T}$ will not generate nested iteration operators. Formulae in such *star-normal form* are known to simplify proof search in PDL [1]. Identifying more restricted fragments that can simplify our epistemic language, even if they are only complete for restricted classes of action theory, is a promising avenue for future research.

On a related note, let us briefly justify why we construct a new dynamic logic semantics in the situation calculus rather than adapting the existing semantics of Golog for our purpose. The difficulty is that Golog has no notion of *state* – while the Golog operator $\pi(x, \delta(x))$ is similar to the FODL operator $\exists x$, its effect is localised to the contained program δ. By contrast, FODL allows variable assignments to affect the entire remaining program. While it is possible to simulate global state in Golog using mutually recursive procedure calls, the presentation and proofs are much cleaner using a special-purpose language as we have developed in this chapter.

Once again, let us note that we deal only with a logic of knowledge in this chapter, rather than a logic of belief. The discussion on belief-based formalisms at the end of Chapter 7 also applies to our group-level formalism. However, there is an intriguing opportunity for future research here – can an approach similar to the one developed in this chapter provide effective reasoning techniques for other group-level mental attitudes in the situation calculus, such a mutual belief or joint goals? We hope that this chapter can provide a foundation for exploring such ideas.

In Chapter 1 we noted a famous result of Halpern and Moses [39], which states that common knowledge cannot be obtained through asynchronous communication. Our complex epistemic modalities reflect this result in a more general way. Suppose we have an action a that is *never* observable by a particular agent:

$$Obs(agt, a) = o \equiv o = \{\}$$

Then according to the definition of K_0, the situations considered possible by agt after the occurrence of a are a *superset* of the situations considered possible by agt before the occurrence of a:

$$K_0(agt, s', s) \rightarrow K_0(agt, s', do(a, s))$$

Thus actions that are always hidden from an agent, such as asynchronous communication actions, cannot *remove* possible situations; they can only add additional uncertainty about the state of the world. For a proposition ϕ to be common knowledge after the occurrence of such an action, it must already have been common knowledge before that action. Always-hidden actions therefore do not increase the common knowledge of the agents, as demonstrated by the results of [39].

Our explicit formulation of Obs allows the epistemic path regressor $\mathcal{T}(\pi, c, c')$ to deal with more subtle cases, such as actions that are only observable under certain conditions. Intuitively, if the agents have common knowledge that an action c will be simultaneously observed, then the occurrence of c can increase the common knowledge of the agents. But if there is any path through the agent's epistemic frame which allows the occurrence of c to be hidden from an agent, then their epistemic frame after the occurrence of c will still contain those situations from before the occurrence of c, and their ability to obtain common knowledge will be limited.

Chapter 9

Conclusion

This thesis has laid the foundations for reasoning about asynchronous multi-agent domains in the situation calculus. As highlighted by our initial investigations and implementation of the multi-agent Golog variant MIndiGolog, the standard reasoning and planning machinery of the situation calculus often depends on an assumption of synchronicity. In many cases, this synchronicity is enforced by requiring all actions to be publicly observable.

We identified three main limitations of the situation calculus when trying to extend its reach into asynchronous domains. First, it generates fully-ordered sequences of actions as the output of the Golog execution planning process, which cannot be feasibly executed in the face of hidden actions. Second, its standard account of agent-level knowledge cannot effectively handle arbitrarily-long sequences of hidden actions. Finally, it lacks a formal account of reasoning about group-level epistemic modalities such as common knowledge, which are vital for managing coordination in multi-agent domains.

At the core of our approach to overcoming these limitations is a new, explicit representation of the local perspective of each agent. By formalising what each agent *observes* when a particular set of actions is performed, and its corresponding local *view* in each situation, we are able to approach reasoning and planning in a principled way without making any assumptions about the dynamics of the domain. In particular, we can explicitly define and represent asynchronous domains as those in which some action occurrences generate no observations; in other words, domains in which agents cannot determine how many actions have been performed.

Building on this foundation, we have developed four key extensions to the reasoning and planning machinery of the situation calculus that work to overcome its current limitations in asynchronous multi-agent domains.

9.1 Contributions

Our first key contribution defines a partially-ordered branching action structure to replace raw situation terms as the output of the Golog execution planning process. Called *joint executions*, they represent a set of many possible histories that could constitute a legal execution of the program. They allow independent actions to be performed independently, while ensuring that inter-agent synchronisation is always possible when required. By formulating these requirements explicitly in terms of the local view available to each agent, we identify joint executions that are feasible to perform in the world despite potential asynchronicity in the domain.

The utility of these structures was demonstrated by implementing an offline execution planner that produces joint executions as its output. By imposing some simple restrictions on the theory of action, the planner is able to reason about joint executions without having to explicitly consider the exponentially-many possible histories of such a partially-ordered structure. It can thus make use of the standard reasoning machinery of the situation calculus developed in existing Golog implementations.

Second, we have characterised a kind of inductive query that we call a *property persistence* query. These queries are restricted enough to be amenable to a special-purpose reasoning algorithm based on a meta-level fixpoint calculation. A simple iterative approximation algorithm was presented and shown to be complete for several interesting cases. More importantly, we have shown that such queries can always be replaced with a uniform formula called the *persistence condition,* in a way that integrates well with the standard regression operator. This allows certain second-order aspects of our formulation to be "factored out" and handled using special-purpose tools, while maintaining the use of regression as the primary reasoning technique.

The third major contribution is a powerful new account of individual-level epistemic reasoning, in which an agent's knowledge is expressed directly in terms of its local observations. In asynchronous domains agents are required to account for arbitrarily-long sequences of hidden actions and must therefore perform some inductive reasoning. By precisely characterising the inductive component of their reasoning in terms of a property persistence query, we factor it out of the reasoning process and provide a regression-based technique for answering knowledge queries.

Basing our formalism explicitly on an agent's observations provides two important benefits. It makes our formalism robust to theory elaboration, so that our

theorems and our regression rule apply unmodified as more complex knowledge-producing actions are added to the domain. Second, it means that a situated agent can directly use our regression rules to reason about its own knowledge using only its local information.

We have also demonstrated that if the theory of action is known to be synchronous, then our regression rule does not require inductive reasoning and it reduces to a simple syntactic transformation. Our account of individual-level knowledge thus provides a flexible new formalism that is comparable in reasoning complexity to the standard account of knowledge for synchronous domains, while at the same time extending gracefully to asynchronous domains where inductive reasoning is required.

Finally, we have introduced a powerful new language of complex epistemic modalities to the situation calculus. Based on an epistemic interpretation of dynamic logic, these modalities are expressive enough to formulate a regression rule for common knowledge while still permitting arbitrarily-long sequences of hidden actions to be handled using the persistence condition operator.

Our work provides the first formal account of effective reasoning about common knowledge in the situation calculus. If the domain is restricted to be synchronous, our regression rule does not require inductive reasoning and common knowledge can be reasoned using a purely syntactic manipulation, a powerful new ability in its own right. By basing our formalism on an explicit representation of each agent's local view, it can also extend gracefully to handle asynchronous domains in which some inductive reasoning is required.

These contributions provide a powerful fundamental framework for the situation calculus to represent and reason about asynchronous multi-agent domains.

9.2 Future Work

Throughout the thesis, we have also identified areas where further work is required to bring our new techniques together with practical systems built on the situation calculus. Work continues on developing a MIndiGolog execution planner capable of coordinating *online* execution of a shared program in asynchronous domains, building on the techniques we have developed here. The most promising avenues of future research are summarised below.

Chapter 8 developed a precise characterisation of the processes required to reason about common knowledge in the situation calculus, and demonstrated that it is decidable in principle in finite domains by conversion into PDL. Unfortunately exist-

ing PDL provers cannot be used for even very simple domains, due to an exponential blowup in problem size during this conversion. We have some initial intuitions on constructing a *free-variable* prover for our modalities to avoid performing this conversion in its entirety, but there is still significant implementation work to be done before the technique can be applied in practical domains.

There is also the possibility of identifying *fragments* of our epistemic language that have more tractable reasoning procedures, but are still powerful enough to capture the regression of common knowledge under certain domain restrictions. A promising starting point would be to extend the relativised common knowledge operator of van Benthem et al. [119] to the first-order case, and determine whether it suffices for regressing common knowledge when all actions are public, as it does in propositional formalisms.

While a possible-worlds formulation of knowledge as developed in this thesis provides an excellent theoretical foundation for epistemic reasoning, possible-worlds reasoning can be highly intractable in practice. In Section 7.6 we suggested a way to combine the formalism for tractable literal-level knowledge of [23] with our new observation-based semantics, which would be interesting to explore in more detail. It would also be interesting to extend the approach of [23] with a literal-level account of common knowledge, to enable approximate reasoning about group-level epistemic modalities. Such a formalism would need to decompose disjunctions inside epistemic paths, and it is far from clear whether this could be done in a principled manner.

Finally, agents in a cooperative team need not consider *all* possible actions of their teammates, since they know that the other agents will behave according to some protocol. We have identified a potential approach based on the work of [33] that would allow protocols to be expressed as Golog programs, but significant work will be required to produce a detailed theory based on these ideas.

References

[1] P. Abate, R. Gore, and F. Widmann. An On-the-fly Tableau-based Decision Procedure for PDL-Satisfiability. In *Proceedings of the Fifth Workshop on Methods for Modalities*, 2007.

[2] Pietro Abate and Rajeev Gor. System Description: The Tableau Work Bench. In *Proc. Fifth Workshop on Methods for Modalities*, 2007.

[3] C. Backstrom. Computational Aspects of Reordering Plans. *Journal of Artificial Intelligence Research*, 9:99–137, 1998.

[4] J. Baier and S. McIlraith. On Planning with Programs what Sense. In *Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning (KR'06)*, pages 492–502, 2006.

[5] Jorge Baier and Javier Pinto. Integrating True Concurrency into the Robot Programming Language GOLOG. In *Proceedings of the XIX International Conference of the Chilean Computer Science Society*, pages 179–186, 1999.

[6] C. Baral and T. Son. Extending ConGolog to allow partial ordering. In *Proceeings of the 6th International Workshop on Agent Theories, Architectures, and Languages*, pages 188–204, 2000.

[7] Bradley Bart, James P. Delgrande, and Oliver Schulte. Knowledge and Planning in an Action-Based Multi-agent Framework: A Case Study. In *Advances in Artificial Intelligence*, volume 2056 of *LNAI*, pages 121–130. Springer, 2001.

[8] Alexandru Batlag, Lawrence S. Moss, and Slawomir Solecki. The Logic of Public Announcements and Common Knowledge and Private Suspicions. In *Proceedings of the 7th conference on Theoretical Aspects of Rationality and Knowledge (TARK'98)*, pages 43–56, 1998.

[9] Kristof Van Belleghem, Marc Denecker, and Danny De Schreye. Combining Situation Calculus and Event Calculus. In *Proceedings of the 12th International Conference on Logic Programming*, pages 83–97, 1995.

[10] Kristof Van Belleghem, Marc Denecker, and Danny De Schreye. On the relation between situation calculus and event calculus. *Journal of Logic Programming*, 31:3–37, 1997.

[11] Leopoldo E. Bertossi, Javier Pinto, Pablo Saez, Deepak Kapur, and Mahadevan Subramaniam. Automating Proofs of Integrity Constraints in Situation Calculus. In *Proceedings of the 9th International Syposium on Methodologies for Intelligent Systems*, Lecture Notes in Artificial Intelligence, pages 212–222. Springer, 1996.

[12] Craig Boutilier, Raymond Reiter, Mikhail Soutchanski, and Sebastian Thrun. Decision-Theoretic, High-Level Agent Programming in the Situation Calculus. In *Proceedings of the 17th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence AAAI'00/IAAI'00*, pages 355–362, 2000.

[13] Ronald J. Brachman and Hector J. Levesque. *Knowledge Representation and Reasoning*. Morgan Kaufmann Publishers, 2004.

[14] Jens Claßen and Gerhard Lakemeyer. A Logic for Non-Terminating Golog Programs. In *Proceedings of the 11th International Conference on Principles of Knowledge Representation and Reasoning (KR'08)*, pages 589–599, 2008.

[15] Patrick Cousot and Radhia Cousot. Constructive Versions of Tarski's Fixed Point Theorems. *Pacific Journal of Mathematics*, 82(1):43–57, 1979.

[16] Ernest Davis and Leora Morgenstern. A First-order Theory of Communication and Multi-agent Plans. *Journal of Logic and Computation*, 15(5):701–749, October 2005.

[17] Giuseppe De Giacomo and Hector Levesque. An Incremental Interpreter for High-Level Programs with Sensing. In *Logical foundation for cognitive agents: contributions in honor of Ray Reiter*. Springer, 1999.

[18] Giuseppe De Giacomo, Evgenia. Ternovska, and Ray. Reiter. Non-terminating processes in the situation calculus. In *In Proceedings of the AAAI'97 Workshop on Robots, Softbots, Immobots: Theories of Action, Planning and Control*, 1997.

[19] Giuseppe De Giacomo, Raymond Reiter, and Mikhail Soutchanski. Execution Monitoring of High-Level Robot Programs. In *Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 453–465, 1998.

[20] Giuseppe De Giacomo, Luca Iocchi, Daniele Nardi, and Riccardo Rosati. A Theory and Implementation of Cognitive Mobile Robots. *Journal of Logic and Computation*, 9:759–785, 1999.

[21] Giuseppe De Giacomo, Yves Lespérance, and Hector Levesque. ConGolog, A Concurrent Programming Language Based on the Situation Calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.

[22] Keith Decker and Victor Lesser. Designing a Family of Coordination Algorithms. In *Proceedings of the 1st International Conference on Multi-Agent Systems (ICMAS'95)*, 1995.

[23] Robert Demolombe and Maria del Pilar Pozos Parra. A Simple and Tractable Extension of Situation Calculus to Epistemic Logic. In *Proceedings of the 12th International Symposium on Foundations of Intelligent Systems (ISMIS'00)*, pages 515–524, 2000.

[24] Silvio do Lago Pereira and Leliane Nunes de Barros. High-Level Robot Programming: An Abductive Approach Using Event Calculus. In *Proceedings of the 17th Brazilian Symposium on Artificial Intelligence*, Advances in Artificial Intelligence, 2004.

[25] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning about Knowledge*. The MIT Press, Cambridge, Massachesetts, 1995.

[26] Alessandro Farinelli, Alberto Finzi, and Thomas Lukasiewicz. Team Programming in Golog under Partial Observability. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 2097–2012, 2007.

[27] A. Ferrein, Ch. Fritz, and G. Lakemeyer. Using Golog for Deliberation and Team Coordination in Robotic Soccer. *Kunstliche Intelligenz*, I:24–43, 2005.

[28] Alberto Finzi and Thomas Lukasiewicz. Game-Theoretic Agent Programming in Golog. In *Proceedings of the 16th biennial European Conference on Artificial Intelligence (ECAI'03)*, 2003.

[29] Alberto Finzi and Thomas Lukasiewicz. Game-Theoretic Golog under Partial Observability. In *Proceedings of the 4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'05)*, pages 1301–1302, 2005.

[30] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2): 374–382, 1985.

[31] Melvin Fitting. *First-order logic and automated theorem proving (2nd ed.)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[32] Melvin Fitting and Richard L. Mendelsohn. *First-order Modal Logic*. Springer, 1998.

[33] Christian Fritz, Jorge A. Baier, and Sheila A. McIlraith. ConGolog, Sin Trans: Compiling ConGolog into Basic Action Theories for Planning and Beyond. In *Proceedings of the 11th International Conference on Principles of Knowledge Representation and Reasoning (KR'08)*, 2008.

[34] Alfredo Gabaldon. Programming Hierarchical Task Networks in the Situation Calculus. In *Proceedings of the 6th International Conference on AI Planning and Scheduling: Workshop on On-line Planning and Scheduling*, 2002.

[35] Hojjat Ghaderi, Hector Levesque, and Yves Lespérance. A Logical Theory of Coordination and Joint Ability. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI'07)*, pages 421–426, 2007.

[36] Henrik Grosskreutz and Gerhard Lakemeyer. cc-Golog: Towards More Realistic Logic-Based Robot Controllers. In *Proceedings of the 17th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence AAAI'00/IAAI'00*, pages 476–482, 2000.

[37] Barbara J. Grosz and Sarit Kraus. Collaborative Plans for Complex Group Action. *Artificial Intelligence*, 86(2):269–357, 1996.

[38] Yilian Gu and Mikhail Soutchanski. Decidable Reasoning in a Modified Situation Calculus. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 1891–1897, 2007.

[39] Joseph Y. Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37(3):549–587, 1990.

[40] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. Foundations of Computing. MIT Press, 2000.

[41] Sief Haridi and Nils Franzén. Tutorial of Oz, 1999. available at http://www.mozart-oz.org/.

[42] L. Kalantari and E. Ternovska. A Model Checker for Verifying ConGolog Programs. In *Proceedings of the 18th AAAI Conference on Artificial Intelligence (AAAI'02)*, 2002.

[43] John Kelly. *The Essence of Logic*. Prentice Hall, Inc., 1996.

[44] Ryan F. Kelly and Adrian R. Pearce. Towards High-Level Programming for Distributed Problem Solving. In *Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT'06)*, pages 490–497, 2006.

[45] Ryan F. Kelly and Adrian R. Pearce. Knowledge and Observations in the Situation Calculus. In *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'07)*, pages 841–843, 2007.

[46] Ryan F. Kelly and Adrian R. Pearce. Property Persistence in the Situation Calculus. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 1948–1953, 2007.

[47] Ryan F. Kelly and Adrian R. Pearce. Complex Epistemic Modalities in the Situation Calculus. In *Proceedings of the 11th International Conference on Principles of Knowledge Representation and Reasoning (KR'08)*, pages 611–620, 2008.

[48] S. Khan and Y. Lespérance. ECASL: A Model of Rational Agency for Communicating Agents. In *Proceedings of the 4th International Joint Conference on Autonomous Agents and Multiagent Systems*, 2005.

[49] Barteld Kooi. Dynamic Term-Modal Logic. In *Proceedings of the Workshop on Logic, Rationality and Interaction*, volume 8 of *Texts in Computing*, pages 173–185, Beijing, 2007. College Publications, London.

[50] R. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1):67–95, 1986. ISSN 0288-3635.

[51] Robert Kowalski and Fariba Sadri. Reconciling the Event Calculus with the Situation Calculus. *Journal of Logic Programming*, 31:39–58, 1997.

[52] Gerhard Lakemeyer. On sensing and off-line interpreting in Golog. In *Logical foundation for cognitive agents: contributions in honor of Ray Reiter*. Springer, 1999.

[53] Martin Lange. Model Checking Propositional Dynamic Logic with All Extras. *Journal of Applied Logic*, 4(1):39–49, 2005.

[54] Y. Lespérance, H. J. Levesque, and R. Reiter. A Situation Calculus Approach to Modeling and Programming Agents. In Rao A. and M. Wooldridge, editors, *Foundations and Theories of Rational Agency*, pages 275–299. Kluwer, 1999.

[55] Y. Lespérance, H. J. Levesque, F. Lin, and R. B. Scherl. Ability and Knowing How in the Situation Calculus. *Studia Logica*, 66(1):165–186, October 2000.

[56] Yves Lespérance. On the Epistemic Feasibility of Plans in Multiagent Systems Specifications. In *Proceedings of the 8th International Workshop on Agent Theories, Architectures, and Languages*, volume 2333 of *Lecture Notes in Artificial Intelligence*, pages 69–85, 2001.

[57] Yves Lespérance and Ho-Kong Ng. Integrating Planning into Reactive High-Level Robot Programs. In *In Proceedings of the Second International Cognitive Robotics Workshop*, pages 49–54, 2000.

[58] Yves Lespérance, Todd G. Kelley, John Mylopoulos, and Eric S. K. Yu. Modeling Dynamic Domains with ConGolog. In *Conference on Advanced Information Systems Engineering*, pages 365–380, 1999.

[59] H. Levesque, F. Pirri, , and R. Reiter. Foundations for the Situation Calculus. *Electronic Transactions on Artificial Intelligence,*, 2(3-4):159–178, 1998.

[60] Hector Levesque. Planning with Loops. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI'05)*, pages 509–515, 2005.

[61] Hector Levesque. What is Planning in the Presence of Sensing? In *Proc. of the 13th National Conference on Artificial Intelligence*, pages 1139–1146. AAAI, 1996.

[62] Hector J. Levesque, Ray Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A Logic Programming Language for Dynamic Domains. *Journal of Logic Programming*, 31(1-3):59–83, 1997.

[63] H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation.* Prentice Hall, Inc., 1981.

[64] F. Lin and H. Levesque. What robots can do: robot programs and effective achievability. *Artificial Intelligence*, 101:201–226, 1998.

[65] F. Lin and Y. Shoham. Concurrent actions in the Situation Calculus. In *Proceedings of the 10th National Conference on Artifical Intelligence (AAAI'92)*, 1992.

[66] Fangzhen Lin and Ray Reiter. State Constraints Revisited. *Journal of Logic and Computation*, 4(5):655–678, 1994.

[67] Fangzhen Lin and Ray Reiter. How to progress a database. *Artificial Intelligence*, 92:131–167, 1997.

[68] Y. Liu and H. J. Levesque. Tractable reasoning with incomplete first-order knowledge in dynamic systems with context-dependent actions. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI'05)*, 2005.

[69] Yves Martin. The Concurrent, Continuous FLUX. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI'03)*, 2003.

[70] John McCarthy and Patrick J. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969.

[71] Robert C. Moore. Reasoning about Knowledge and Action. Technical Note 191, SRI International, October 1980.

[72] Mogens Nielsen, Gordon D. Plotkin, and Glynn Winskel. Petri Nets, Event Structures and Domains. In *Proceedings of the International Symposium on Semantics of Concurrent Computation*, volume 70 of *Lecture Notes in Computer Science*, pages 266–284, 1979.

[73] Eric Pacuit. Some Comments on History Based Structures. *Journal of Applied Logic*, 5(4):613–624, 2007.

[74] Rohit Parikh and R. Ramanujam. Distributed Processes and the Logic of Knowledge. In *Proceedings of the Conference on Logic of Programs*, pages 256–268. Springer-Verlag, 1985.

[75] M. Peot and D. Smith. Conditional Nonlinear Planning. In *Proceedings of the 1st International Conference on AI Planning Systems*, pages 189–197, 1992.

[76] Ron Petrick and Hector Levesque. Knowledge equivalence in Combined Action Theories. In *Proceedings of the 8th International Conference on Principles of Knowledge Representation and Reasoning (KR'02)*, 2002.

[77] Ronald P. A. Petrick. *A Knowledge-level approach for effective acting, sensing, and planning.* PhD thesis, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada, 2006.

[78] Ronald P. A. Petrick. Cartesian Situations and Knowledge Decomposition in the Situation Calculus. In *Proceedings of the 11th International Conference on Principles of Knowledge Representation and Reasoning (KR'08)*, pages 629–639, 2008.

[79] Javier Pinto. Concurrency and Action Interaction. Unpublished work, submitted for publication: November, 2000. URL `http://www.scs.carleton.ca/~bertossi/javier/papers/pinto00.ps.gz`.

[80] Javier Pinto. Concurrent Actions and Interacting Effects. In *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 292–303, 1998.

[81] Javier Pinto. Using histories to model observations in theories of action. In *Selected Papers from the Workshop on Reasoning with Incomplete and Changing Information and on Inducing Complex Representations: Learning and Reasoning with Complex Representations*, volume 1359 of *Lecture Notes In Computer Science*, pages 221 – 233, 1998.

[82] Javier Pinto and Raymond Reiter. Reasoning About Time in the Situation Calculus. *Annals of Mathematics and Artificial Intelligence*, 14(2-4):251–268, 1995.

[83] Javier A. Pinto. *Temporal Reasoning in the Situation Calculus.* PhD thesis, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada, 1994.

[84] Fiora Pirri and Ray Reiter. Planning with natural actions in the situation calculus. In *Logic-Based Artificial Intelligence.* Kluwer Press, 2000.

[85] Fiora Pirri and Ray Reiter. Some contributions to the metatheory of the situation calculus. *Journal of the ACM*, 46(3):325–361, 1999.

[86] David A. Plaisted and Yunshan Zhu. Situation Calculus with Aspect. In *Proceedings of the IASTED International Conference on Artificial Intelligence and Soft Computing*, pages 17–20, 1997.

[87] Joachim Posegga and Peter H. Schmitt. Automated Deduction with Shannon Graphs. *Journal of Logic and Computation*, 5(6):697–729, 1995.

[88] Vaughan R. Pratt. Modeling Concurrency with Geometry. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*, pages 311–322, 1991.

[89] Ray Reiter. Proving Properties of States in the Situation Calculus. *Artificial Intelligence*, 64:337–351, 1993.

[90] Ray Reiter. Sequential, Temporal GOLOG. In *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 547–556, Trento, Italy, 1998.

[91] Ray Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. The MIT Press, 2001.

[92] Ray Reiter. The frame problem in situation the 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 Professional, Inc., 1991.

[93] Ray Reiter. Natural Actions, Concurrency and Continuous Time in the Situation Calculus. In *Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning (KR'96)*, pages 2–13, 1996.

[94] Peter Van Roy. Logic Programming in Oz with Mozart. In Danny De Schreye, editor, *International Conference on Logic Programming*, pages 38–51. The MIT Press, November 1999.

[95] Sebastian Sardina, Giuseppe De Giacomo, Yves Lespénce, and Hector Levesque. On the Semantics of Deliberation in IndiGolog – From Theory to Implementation. *Annals of Mathematics and Artificial Intelligence*, 41(2–4): 259–299, August 2004.

[96] Francesco Savelli. Existential assertions and quantum levels on the tree of the situation calculus. *Artificial Intelligence*, 170(6):643–652, 2006.

[97] Richard Scherl. Reasoning about the interaction of knowledge, time and concurrent actions in the situation calculus. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI'03)*, pages 1091–1098, 2003.

[98] Richard Scherl and Hector Levesque. Knowledge, Action, and the Frame Problem. *Artificial Intelligence*, 144:1–39, 2003.

[99] S. Schiffel and M Thielscher. Interpreting Golog Programs in Flux. In *Proceedings of the 7th International Symposium on Logical Formalizations of Commonsense Reasoning*, 2005.

[100] S. Schiffel and M. Thielscher. Reconciling Situation Calculus and Fluent Calculus. In *Proceedings of the 21st National Conference on Artificial Intelligence and the 18th Innovative Applications of Artificial Intelligence Conference (AAAI'06/IAAI'06)*, 2006.

[101] Christian Schulte. *Programming Constraint Services.* Doctoral dissertation, Universität des Saarlandes, Naturwissenschaftlich-Technische Fakultät I, Fachrichtung Informatik, Saarbrücken, Germany, 2000.

[102] Christian Schulte. Parallel Search Made Simple. Technical Report TRA9/00, School of Computing, National University of Singapore, 55 Science Drive 2, Singapore 117599, September 2000.

[103] Murray Shanahan. Event calculus planning revisited. In *Proceedings of the 4th European Conference on Planning (ECP'97)*, number 1348 in Lecture Notes in Artificial Intelligence, pages 390–402. Springer, 1997.

[104] Murray Shanahan and Mark Witkowski. High-Level Robot Control Through Logic. In *Proceedings of the 7th International Workshop on Agent Theories, Architectures, and Languages (ATAL2000)*. Springer, 2000.

[105] S. Shapiro and M. Pagnucco. Iterated Belief Change and Exogenous Actions in the Situation Calculus. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI'04)*, pages 878–882, 2004.

[106] S. Shapiro, Y. Lespérance, and H. J. Levesque. Specifying Communicative Multi-Agent Systems. In *Agents and Multi-Agent Systems - Formalisms, Methodologies, and Applications*, volume 1441 of *Lecture Notes in Artificial Intelligence*, pages 1–14, 1998.

[107] S. Shapiro, Y. Lesperance, and H. Levesque. Goal Change in the Situation Calculus. *Journal of Logic and Computation*, 17:983–1018, 2007.

[108] Steven Shapiro and Yves Lespérance. Modeling Multiagent Systems with the Cognitive Agents Specification Language — A Feature Interaction Resolution Application. In *Intelligent Agents Volume VII — Proceedings of the 2000 Workshop on Agent Theories, Architectures, and Languages*, pages 244–259. Springer-Verlag, 2001.

[109] Steven Shapiro, Maurice Pagnucco, Yves Lespérance, and Hector J. Levesque. Iterated Belief Change in the Situation Calculus. In *Proceedings of the 7th International Conference on Principles of Knowledge Representation and Reasoning (KR'00)*, pages 527–538, 2000.

[110] Steven Shapiro, Yves Lespérance, and Hector J. Levesque. The Cognitive Agents Specification Language and Verification Environment for Multiagent Systems. In *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'02)*, pages 19–26, 2002.

[111] Tran Cao Son, Chitta Baral, and Le-Chi Tuan. Adding Time and Intervals to Procedural and Hierarchical Control Specifications. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI'04)*, pages 92–97, 2004.

[112] M. Tambe. Towards Flexible Teamwork. *Journal of Artificial Intelligence Research*, 7:83–124, 1997.

[113] M. Thielscher. A Unifying Action Calculus. *Artificial Intelligence*, :, 2007. (in submission).

[114] Michael Thielscher. Logic-Based Agents and the Frame Problem: A Case for Progression. In V. Hendricks, editor, *First-Order Logic Revisited: Proceedings of the Conference 75 Years of First Order Logic (FOL75)*, pages 323–336, Berlin, Germany, 2004. Logos.

[115] Michael Thielscher. Introduction to the Fluent Calculus. *Electronic Transations on Artificial Intelligence*, 2:179–192, 1998.

[116] Michael Thielscher. From Situation Calculus to Fluent Calculus: State update axioms as a solution to the inferential frame problem. *Artificial Intelligence*, 111:277–299, 1999.

[117] Johan van Bentham. Modal Logic meets Situation Calculus. Technical Report PP-2007-04, University of Amsterdam, 2007.

[118] Johan van Benthem and Eric Pacuit. The Tree of Knowledge in Action: Towards a Common Perspective. In *Advances in Modal Logic*, volume 6, pages 87–106, 2006.

[119] Johan van Benthem, Jan van Eijck, and Barteld Kooi. Logics of Communication and Change. *Information and Computation*, 204(II):1620–1662, 2006.

[120] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, March 2004.

[121] Peter Van Roy and Seif Haridi. Mozart: A Programming System for Agent Applications. In *Proceedings of the International Workshop on Distributed and Internet Programming with Logic and Constraint Languages*, November 1999. (ICLP 99).

[122] Peter Van Roy, Per Brand, Denys Duchier, Seif Haridi, Martin Henz, and Christian Schulte. Logic programming in the context of multiparadigm programming: the Oz experience. *Theory and Practice of Logic Programming*, 3 (6):717–763, 2003.

[123] Stavros Vassos and Hector Levesque. Progression of Situation Calculus Action Theories with Incomplete Information. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 2024–2029, 2007.

[124] Stavros Vassos and Hector Levesque. On the Progression of Situation Calculus Basic Action Theories: Resolving a 10-year-old Conjecture. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI'08)*, pages 1004–1009, 2008.

[125] Stavros Vassos, Gerhard Lakemeyer, and Hector J. Levesque. First-Order Strong Progression for Local-Effect Basic Action Theories. In *Proceedings of the 11th International Conference on Principles of Knowledge Representation and Reasoning (KR'08)*, pages 662–671, 2008.

Appendix A

Detailed Proofs

This appendix contains complete proofs for various lemmas and theorems throughout the paper, along with some additional lemmas.

Theorem 5. *For any $n \in \mathbb{N}$, $\mathcal{P}^n_{\mathcal{D}}(\phi, \alpha)[\sigma]$ iff ϕ holds in σ and in all successors of σ reached by performing at most n actions satisfying α:*

$$\mathcal{D} \models \mathcal{P}^n_{\mathcal{D}}(\phi, \alpha)[\sigma] \equiv \bigwedge_{i \leq n} \forall a_1, \ldots, a_i : \left(\bigwedge_{j \leq i} \alpha[a_j, do([a_1, \ldots, a_{j-1}], \sigma)] \rightarrow \phi[do([a_1, \ldots, a_i], \sigma)] \right)$$

Proof. By induction on the natural numbers. For $n = 0$ we have $\phi[\sigma] \equiv \phi[\sigma]$ by definition. For the inductive case, we expand the definition of $\mathcal{P}^n_{\mathcal{D}}(\phi, \alpha)[\sigma]$ to get the following for the LHS:

$$\mathcal{P}^{n-1}_{\mathcal{D}}(\phi, \alpha)[\sigma] \wedge \forall a : \mathcal{R}_{\mathcal{D}}(\alpha[a, \sigma]) \rightarrow \mathcal{R}_{\mathcal{D}}(\mathcal{P}^{n-1}_{\mathcal{D}}(\phi, \alpha)[do(a, \sigma)])$$

By the inductive hypothesis we can equate $\mathcal{P}^{n-1}_{\mathcal{D}}(\phi, \alpha)[\sigma]$ in this LHS with all but the $i = n$ clause from the RHS conjunction, and we suppress them on both sides. If we also drop the regression operators we are left with the following to establish:

$$\mathcal{D} \models \forall a : \alpha[a, \sigma] \rightarrow \mathcal{P}^{n-1}_{\mathcal{D}}(\phi, \alpha)[do(a, \sigma)] \equiv \forall a_1, \ldots, a_n : \left(\bigwedge_{j \leq n} \alpha[a_j, do([a_1, \ldots, a_{j-1}], \sigma)] \rightarrow \phi[do([a_1, \ldots, a_n], \sigma)] \right)$$

We can again use the inductive hypothesis on $\mathcal{P}^{n-1}_{\mathcal{D}}$ in the LHS of this equivalence. If we then distribute the $\alpha[a, \sigma]$ implication over the outermost conjunction

and collect quantifiers, we obtain the following for the LHS:

$$\bigwedge_{i \leq n-1} \forall a, a_1, \ldots, a_i : \left(\alpha[a, \sigma] \wedge \bigwedge_{j \leq i} \alpha[a_j, do([a, a_1, \ldots, a_{j-1}], \sigma)] \rightarrow \phi[do([a, a_1, \ldots, a_i], \sigma)] \right)$$

Renaming $a \Rightarrow a_1$, $a_1 \Rightarrow a_2$, $\ldots$,$a_i \Rightarrow a_{i+1}$, we see that each of the $i < n-1$ clauses on the LHS is equivalent to one of the $i < n$ clauses that have been suppressed on the RHS. The remaining $i = n-1$ clause is equivalent to the required RHS:

$$\forall a_1, \ldots, a_n : \left(\bigwedge_{j \leq n} \alpha[a_j, do([a_1, \ldots, a_{j-1}], \sigma)] \rightarrow \phi[do([a_1, \ldots, a_n], \sigma)] \right)$$

We therefore have the desired equivalence. □

Lemma 7. *For situation terms s and s', and agent agt:*

$$\mathcal{D} \cup \mathcal{D}_K^{obs} \models Legal(s) \wedge s \leq_{LbU(agt)} s' \rightarrow Legal(s')$$

Proof. Since LbU implies $Legal$, $s \leq_{LbU(agt)} s'$ implies $s \leq_{Legal} s'$. Given the definition of $Legal(s)$ as $root(s) \leq_{Legal} s$, we have the implication as desired. □

Lemma 8. *For situation terms s and s', and agent agt:*

$$\mathcal{D} \cup \mathcal{D}_K^{obs} \models s \leq_{LbU(agt)} s' \rightarrow View(agt, s') = View(agt, s)$$

Proof. By induction on situations, using the definition of $View$ from equation (4.1). In the base case of $s' = s$ the lemma is trivial. For the inductive case let $s' = do(c, s'')$, with $s \leq_{LbU(agt)} s''$ and $View(agt, s'') = View(agt, s)$ by the inductive hypothesis. By the definition of $\leq_{LbU(agt)}$ we have that $Obs(agt, c, s'') = \{\}$. By the definition of $View$ we then have that $View(agt, s') = View(agt, s'')$, which equals $View(agt, s)$ giving the lemma as required. □

Lemma 9. *For situation terms s and s'', and agent agt:*

$$\mathcal{D} \cup \mathcal{D}_K^{obs} \models K(agt, s'', s) \rightarrow K_0(agt, root(s''), root(s))$$

Proof. By induction on situations. The lemma is trivial in the base case of $Init(s)$. For the $do(c, s)$ case, suppose that we have $K(agt, s'', do(c, s))$. Then by equation (7.6) there is some s' such that $s' \sqsubseteq s''$ and $K(agt, s', s)$. Then $root(s'') = root(s')$, and $K_0(root(s'), root(s))$ by the inductive hypothesis, giving the required result. □

Lemma 10. *For situation terms s and s'', and agent agt:*

$$\mathcal{D} \cup \mathcal{D}_K^{obs} \models K(agt, s'', s) \rightarrow Legal(s'')$$

Proof. By induction on situations. All s' such that $K_0(agt, s', s)$ are initial and therefore legal. So using equation (7.7) in the base case, if $K(agt, s'', s)$ then there must be an s' such that $Init(s')$ and $s' \leq_{LbU(agt)} s''$, making s'' legal by Lemma 7. For the $do(c, s)$ case, equation (7.6) ensures that $do(c', s') \leq_{LbU(agt)} s''$ for some s',c' satisfying $K(agt, s', s)$ and $Legal(c', s')$. So s' is legal by the inductive hypothesis, making s'' legal by Lemma 7 as required. □

Lemma 11. *For situation terms s and s'', and agent agt:*

$$\mathcal{D} \cup \mathcal{D}_K^{obs} \models K(agt, s'', s) \rightarrow View(agt, s'') = View(agt, s)$$

Proof. By induction on situation terms. For the base case of $Init(s)$, using equation (7.7), $K(agt, s'', s)$ implies that there must be an s' such that $Init(s')$ and $s' \leq_{LbU(agt)} s''$. Therefore $View(s'') = View(s') = \epsilon = View(s)$ as required.

For the $do(c, s)$ case, suppose $Obs(agt, c, s) = \{\}$. We have $View(agt, do(c, s)) = View(agt, s)$, while equation (7.6) gives us $K(agt, s'', s)$, which yields $View(agt, s'') = View(agt, s)$ by the inductive hypothesis.

Alternately, suppose $Obs(agt, c, s) \neq \{\}$, then equation (7.6) gives us s',c' such that $do(c', s') \leq_{LbU(agt)} s''$, $Obs(agt, c, s) = Obs(agt, c', s')$, and $K(agt, s', s)$. By the inductive hypothesis $View(agt, s') = View(agt, s)$, and we have the following: $View(agt, s'') = View(agt, do(c', s')) = Obs(agt, c, s) \cdot View(agt, s')$. This is in turn equal to $View(agt, do(c, s))$ as required. □

Theorem 8. *For any agent agt and situations s and s'':*

$$\mathcal{D} \cup \mathcal{D}_K^{obs} \models K(agt, s'', s) \equiv \\ K_0(root(s''), root(s)) \wedge Legal(s'') \wedge View(agt, s'') = View(agt, s)$$

Proof. For the *if* direction, we simply combine Lemmas 9, 10 and 11. For the *only-if* direction we proceed by induction on situations. In the base case of $Init(s)$, the $\exists s'$ part of equation (7.7) is trivially satisfied by $root(s'')$ and the equivalence holds as required.

For the inductive case with $do(c, s)$, we have two sub-cases to consider. Suppose $Obs(agt, c, s) = \{\}$: then $View(agt, s'') = View(agt, do(c, s)) = View(agt, s)$ and $K(agt, s'', s)$ holds by the inductive hypothesis, satisfying the equivalence in equation (7.6). Alternately, suppose $Obs(agt, c, s) \neq \{\}$: then we have:

$$View(agt, do(c, s)) = Obs(agt, c, s) \cdot View(agt, s) = View(agt, s'')$$

Since s'' is legal, this implies there there is some s',c' satisfying $Obs(agt, c', s') = Obs(agt, c, s)$, $View(agt, s') = View(agt, s)$ and $do(c', s') \leq_{LbU(agt)} s''$. This is

enough to satisfy the $\exists s', c'$ part of equation (7.6) and so the equivalence holds as required. □

Theorem 11. *Given a basic action theory $\mathcal{D} \cup \mathcal{D}_K^{obs}$ and a uniform formula ϕ:*

$$\mathcal{D} \cup \mathcal{D}_K^{obs} \models \mathbf{Knows}(agt, \phi, s) \equiv \mathcal{R}(\mathbf{Knows}(agt, \phi, s))$$

Proof. To obtain this result, we must establish that our new regression rules in equations (7.10) and (7.11) are equivalences under the theory of action $\mathcal{D} \cup \mathcal{D}_K^{obs}$. The mechanics of the proof mirror the analogous proof in [98], but with the addition of a persistence condition application.

For notational clarity we define the abbreviation $\mathbf{PEO}(agt, \phi, o, s)$ (for "persists under equivalent observations") which states that ϕ holds in all legal futures of s compatible with observations o:

$$\begin{aligned}&\mathbf{PEO}(agt, \phi, o, s) \stackrel{\text{def}}{=} \\ &\quad \forall c' : Obs(agt, c', s) = o \wedge Legal(c', s) \rightarrow \left[\forall s' : do(c', s) \leq_{LbU(agt)} s' \rightarrow \phi[s']\right]\end{aligned}$$

Expanding the definition of the **Knows** macro at $do(c, s)$, and applying the successor state axiom from equation (7.6) to the $K(agt, s'', do(c, s))$ term, we can produce the following:

$$\begin{aligned}\mathbf{Knows}(agt, \phi, do(c, s)) \equiv &\forall s'' : K(agt, s'', do(c, s)) \rightarrow \phi[s''] \\ \equiv &\exists o : Obs(agt, c, s) = o \\ &\wedge \left[o = \{\} \rightarrow \forall s' : K(agt, s', s) \rightarrow \phi[s']\right] \\ &\wedge \left[o \neq \{\} \rightarrow \forall s' : K(agt, s', s) \rightarrow \mathbf{PEO}(agt, \phi, o, s')\right]\end{aligned}$$

Noting that both conjuncts contain sub-formulae matching the form of the **Knows** macro, it can be substituted back in to give:

$$\begin{aligned}\mathbf{Knows}(agt, \phi, do(c, s)) \equiv &\exists o : Obs(agt, c, s) = o \\ &\wedge [o = \{\} \rightarrow \mathbf{Knows}(agt, \phi, s)] \\ &\wedge [o \neq \{\} \rightarrow \mathbf{Knows}(agt, \mathbf{PEO}(agt, \phi, o), s)]\end{aligned}$$

For $\mathbf{PEO}(agt, \phi, o, s')$ to legitimately appear inside the **Knows** macro it must be uniform in the situation variable s'. Applying the persistence condition and regressing to make the expression uniform, we develop the following equivalence:

$$\mathbf{PEO}(agt, \phi, o, s) \equiv \forall c' : Obs(agt, c', s) = o \wedge Legal(c', s) \rightarrow \mathcal{R}(\mathcal{P}(\phi, LbU(agt)), c')$$

Suppressing the situation term in this uniform formula gives the regression rule from equation (7.10) as required.

For S_0, a straightforward transformation of equations (2.2) and (7.7) gives:

$$\mathbf{Knows}(agt, \phi, S_0) \equiv \forall s : K_0(agt, s, S_0) \rightarrow \left[\forall s' : s \leq_{LbU(agt)} s' \rightarrow \phi[s']\right]$$

Applying the persistence condition operator, this can easily be re-written as:

$$\mathbf{Knows}(agt, \phi, S_0) \equiv \forall s \,.\, K_0(agt, s, S_0) \rightarrow \mathcal{P}(\phi, LbU(agt))[s]$$

This matches the form of the definition for $\mathbf{Knows_0}$, which we can substitute in to give:

$$\mathbf{Knows}(agt, \phi, S_0) \equiv \mathbf{Knows_0}(agt, \mathcal{P}(\phi, LbU(agt)), S_0)$$

Since all situations reachable by K_0 are initial, and since regression preserves equivalence, it is valid to use $\mathcal{R}(\psi[S_0])^{-1}$ on the enclosed formula to give:

$$\mathbf{Knows}(agt, \phi, S_0) \equiv \mathbf{Knows_0}(agt, \mathcal{R}(\mathcal{P}(\phi, LbU(agt))[S_0])^{-1}, S_0)$$

This is the regression rule from equation (7.11) as required. Our regression rules are thus equivalences under the theory $\mathcal{D} \cup \mathcal{D}_K^{obs}$, and the theorem holds. □

Theorem 12. *Given a basic action theory $\mathcal{D}$ and a uniform formula ϕ:*

$$\mathcal{D} \cup \mathcal{D}_K^{obs} \models \mathbf{Knows}(agt, \phi, v) \equiv \mathcal{R}(\mathbf{Knows}(agt, \phi, v))$$

Proof. Recall the definition of $\mathbf{Knows}(agt, \phi, v)$ as follows:

$$\mathbf{Knows}(agt, \phi, v) \stackrel{\text{def}}{=} \forall s : View(agt, s) = v \wedge root(s) = S_0 \wedge Legal(s) \rightarrow \mathbf{Knows}(agt, \phi, s)$$

We also have the following simple corollary of Theorem 8:

$$\mathcal{D} \cup \mathcal{D}_K^{obs} \models \forall s, s', s'' : root(s) = root(s') \wedge View(s) = View(s') \wedge K(agt, s'', s) \rightarrow K(agt, s'', s')$$

The definition of $\mathbf{Knows}(agt, \phi, v)$ is thus equivalent to:

$$\mathbf{Knows}(agt, \phi, v) \equiv \neg\exists s : View(agt, s) = v \wedge root(s) = S_0 \wedge Legal(s) \vee \exists s : View(agt, s) = v \wedge root(s) = S_0 \wedge Legal(s) \wedge \mathbf{Knows}(agt, \phi, s)$$

We thus need to find a single witness situation rather than examining all situations with that view. We proceed by induction over views. For the ϵ case, S_0 serves as an appropriate witness since it is always legal, $View(agt, S_0) = \epsilon$ and $root(S_0) = S_0$. Applying the regression rule for $\mathbf{Knows}(agt, \phi, S_0)$ gives us the same expression as applying the regression rule for $\mathbf{Knows}(agt, \phi, \epsilon)$. So if $\mathcal{R}(\mathbf{Knows}(agt, \phi, \epsilon))$ holds then so does $\mathbf{Knows}(agt, \phi, S_0)$. Using S_0 as a witness we conclude that $\mathbf{Knows}(agt, \phi, \epsilon)$ iff $\mathcal{R}(\mathbf{Knows}(agt, \phi, \epsilon))$ as desired.

For the inductive $o \cdot v$ case we split on whether there is any situation having that view. Suppose there is no such situation, then the definition of $\mathbf{Knows}(agt, \phi, o \cdot v)$ is trivially satisfied and the agent must know all statements. We need to show that the regression of $\mathbf{Knows}(agt, \phi, o \cdot v)$ is always entailed by the domain in this case. The regressed expression is:

$$\mathbf{Knows}(agt, \forall c : Obs(agt, c) = o \wedge Legal(c) \rightarrow \mathcal{R}(\mathcal{P}(\phi, LbU(agt)), c), v)$$

If there is no situation having view v, then there is also no situation having view $o \cdot v$, and the above is entailed by the inductive hypothesis in this case.

Alternately, suppose there is a situation s having view v but no legal situation having view $o \cdot v$. Then all situations s' that have view equal to v must satisfy $\neg\exists c : Obs(agt, c, s') = o \wedge Legal(c, s')$, otherwise we could construct a situation with view $o \cdot v$. Since these situations s' are the only ones that can be K-related to s, the antecedent in the above implication is falsified at all such situations, and the regressed expression is equivalent to $\mathbf{Knows}(agt, \top, v)$ which is trivially entailed.

Finally, suppose there is a legal situation $do(c, s)$ having view $o \cdot v$. We can assume without loss of generality that $Obs(agt, c, s) = o$ and $View(agt, s) = v$. Regressing $\mathbf{Knows}(agt, \phi, do(c, s))$ in this case will produce:

$$\begin{aligned}\mathcal{R}(\mathbf{Knows}(agt, \phi, do(c, s)) = \exists o' : Obs(agt, c, s) = o' \\ \wedge \left[o' = \{\} \rightarrow \mathbf{Knows}(agt, \phi, s)\right] \\ \wedge \left[o' \neq \{\} \rightarrow \mathbf{Knows}(agt, \forall c' : Obs(agt, c') = o' \right. \\ \left. \wedge Legal(c') \rightarrow \mathcal{R}(\mathcal{P}(\phi, LbU(agt)), c'), s)\right]\end{aligned}$$

We have that $Obs(agt, c, s) = o$ and $o \neq \{\}$, so this is equivalent to:

$$\mathbf{Knows}(agt, \forall c' : Obs(agt, c') = o \wedge Legal(c') \rightarrow \mathcal{R}(\mathcal{P}(\phi, LbU(agt)), c'), s)$$

Since this matches the form of $\mathcal{R}(\mathbf{Knows}(agt, \phi, o \cdot v))$, and we have that the view of s is v, this will be entailed by the domain precisely when the regression of $\mathbf{Knows}(agt, \phi, o \cdot v)$ is entailed by the domain.

Thus if there is no legal situation with view v then $\mathcal{R}(\mathbf{Knows}(agt, \phi, v))$, is always entailed, while if there is such a situation s then $\mathcal{R}(\mathbf{Knows}(agt, \phi, v))$ is equivalent to $\mathcal{R}(\mathbf{Knows}(agt, \phi, s))$. The regression rules over observations are thus equivalences as desired. □

Lemma 12. *For any epistemic path π:*

$$\begin{aligned}\mathcal{D} \cup \mathcal{D}_K^{obs} \models \mathbf{KDo_0}(\pi, do(c, s), s'') \equiv \exists \mu, \mu', c', s' : \\ \mu(x) = c \wedge \mu'(x) = c' \wedge \left(c' \neq \{\} \wedge s'' = do(c', s') \vee c' = \{\} \wedge s'' = s'\right) \\ \wedge \mathbf{KDo_0}(\mathcal{T}_a(\pi, x), \mu, s, \mu', s')\end{aligned}$$

Proof. Proceed by cases, covering each path operator in turn. For the base case of

an individual agent, we have:

$$\mathbf{KDo_0}(agt, do(c,s), s'') \equiv K_0(agt, s'', do(c,s))$$

$$\mathcal{T}_a(agt, x) = \; z \Leftarrow Obs(agt, x)\,;\, agt\,;\, \exists x\,;\, ?Legal(x) \vee x = \{\}\,;\, ?Obs(agt, x) = z$$

Expanding $\mathbf{KDo_0}(\mathcal{T}_a(agt, x), \mu, s, \mu', s')$ thus produces:

$$\begin{aligned}\mathbf{KDo_0}(\mathcal{T}_a(agt, x), \mu, s, \mu', s') \equiv z \Leftarrow Obs(agt, \mu(x), s) \wedge \exists s'' : K_0(agt, s'', s) \wedge \\ \big(Legal(\mu'(x), s'') \vee \mu'(x) = \{\}\big) \wedge Obs(agt, \mu'(x), s'') = z \wedge s'' = s'\end{aligned}$$

Note that μ and μ' are never applied to a variable other than x. When we substitute this into the RHS of the hypothesis, $\mu(x)$ and $\mu'(x)$ are asserted to be c and c' respectively, so they can be simplified away to give:

$$\begin{aligned}\mathcal{D} \cup \mathcal{D}_K^{obs} \models K(agt, s'', do(c,s)) \equiv \\ \exists c', s' : \big(c' \neq \{\} \wedge s'' = do(c', s') \vee s'' = s' \wedge c' = \{\}\big) \\ \wedge K_0(agt, s, s') \wedge \big(Legal(c', s') \vee c' = \{\}\big) \wedge Obs(agt, c, s) = Obs(agt, c', s')\end{aligned}$$

This is the successor state axiom for K_0, which is trivially entailed by the domain.

For the $?\phi$ case, we have:

$$\mathbf{KDo_0}(?\phi, do(c,s), s'') \equiv \phi[do(c,s)] \wedge s'' = do(c,s)$$

$$\mathcal{T}_a(?\phi, x) = ?\mathcal{R}(\phi, x)$$

Giving:

$$\mathbf{KDo_0}(\mathcal{T}_a(?\phi, x), \mu, s, \mu', s') \equiv \mathcal{R}(\phi, x)[s] \wedge s = s' \wedge \mu = \mu'$$

Substituting into the RHS of the hypothesis, this asserts that $c = c'$ and hence $s'' = do(c,s)$, so the hypothesis is clearly entailed.

The case for $\exists y$ is trivial as $\mathbf{KDo_0}(\exists y, s, s') \equiv s = s'$.

The inductive cases are straightforward as $\mathcal{T}_a$ is simply pushed inside each operator. We will take the π^* case as an example. The inductive hypothesis gives:

$$\begin{aligned}\mathbf{KDo_0}(\pi, do(c,s), s'') \equiv \exists \mu, \mu', c', s' : \mu(x) = c \wedge \mu'(x) = c' \\ \wedge \big(c' \neq \{\} \wedge s'' = do(c', s') \vee s'' = s' \wedge c' = \{\}\big) \wedge \mathbf{KDo_0}(\mathcal{T}_a(\pi, x), \mu, s, \mu,' s')\end{aligned}$$

We can apply reflexive transitive closure to both sides of this equivalence, along with two rearrangements: the LHS is expanded to the four-argument form with $\exists \mu, \mu''$ at its front, and the rigid tests on the RHS are taken outside the RTC operation.

This produces the following equivalence:

$$\exists \mu, \mu'' : RTC[\mathbf{KDo_0}(\pi, \mu, do(c, s), \mu'', s'')] \equiv \exists \mu, \mu', c', s' : \mu(x) = c \wedge \mu'(x) = c' \wedge (c' \neq \{\} \wedge s'' = do(c', s') \vee s' = \{\} \wedge s'' = s'c \wedge RTC[\mathbf{KDo_0}(\mathcal{T}_a(\pi, x), \mu, s, \mu,' s')] \tag{A.1}$$

Using the definitions of $\mathbf{KDo_0}$ and $\mathcal{T}_a$ we have:

$$\mathbf{KDo_0}(\pi^*, do(c, s), s'') \equiv \exists \mu, \mu'' : RTC[\mathbf{KDo_0}(\pi, \mu, do(c, s), \mu'', s'')]$$

$$\mathbf{KDo_0}(\mathcal{T}_a(\pi^*, x), \mu, s, \mu', s') \equiv RTC[\mathbf{KDo_0}(\mathcal{T}_a(\pi, x), \mu, s, \mu', s')]$$

Substituting these into the RTC of the inductive hypothesis from equation (A.1) gives us the $\mathbf{KDo_0}(\pi^*)$ cases we need to satisfy the theorem. □

Theorem 14. *For any epistemic path π:*

$$\mathcal{D} \cup \mathcal{D}_K^{obs} \models \mathbf{KDo_0}(\pi, do(c, s), s'') \equiv \exists c', s' : \left(c' \neq \{\} \wedge s'' = do(c', s') \vee c' = \{\} \wedge s'' = s'\right) \wedge \mathbf{KDo_0}(\mathcal{T}(\pi, c, c'), s, s')$$

Proof. Recall the rule for $\mathcal{T}(\pi, c, c')$:

$$\mathcal{T}(\pi, c, c') \stackrel{\text{def}}{=} x \Leftarrow c \,;\, \mathcal{T}_a(\pi, x) \,;\, ?x = c'$$

Expanding $\mathbf{KDo_0}$ for this rule:

$$\mathbf{KDo_0}(\mathcal{T}(\pi, c, c'), s, s') \equiv \exists \mu, \mu' : \mu(x) = c \wedge \mu'(x) = c' \wedge \mathbf{KDo_0}(\mathcal{T}_a(\pi, x), \mu, s, \mu', s')$$

We can thus substitute $\mathbf{KDo_0}(\mathcal{T}(\pi, c, c'), s, s')$ into the RHS of Lemma 12 to get the required result. □

Theorem 15. *For any epistemic path π, uniform formula ϕ and action c:*

$$\mathcal{D} \cup \mathcal{D}_K^{obs} \models \mathbf{PKnows_0}(\pi, \phi, do(c, s)) \equiv \forall c' : \mathbf{PKnows_0}(\mathcal{T}(\pi, c, c'), \mathcal{R}(\phi, c'), s)$$

Proof. The mechanics of this proof mirror that of Theorem 11: we expand the $\mathbf{PKnows_0}$ macro, apply Theorem 14 as a successor state axiom for $\mathbf{KDo_0}$, re-arrange to eliminate existential quantifiers, then collect terms back into forms that

match $\mathbf{PKnows_0}$. We begin with the following:

$$\begin{aligned}
&\mathbf{PKnows_0}(\pi, \phi, do(c,s)) \equiv \forall s'' : \mathbf{KDo_0}(\pi, do(c,s), s'') \rightarrow \phi[s''] \\
&\equiv \forall s'' : \big[\exists c', s' : \big(c' \neq \{\} \wedge s'' = do(c', s') \vee c' = \{\} \wedge s'' = s'\big) \\
&\qquad\qquad \wedge \mathbf{KDo_0}(\mathcal{T}(\pi, c, c'), s, s')\big] \rightarrow \phi[s''] \\
&\equiv \forall s'', c', s' : \big[\big(c' \neq \{\} \wedge s'' = do(c', s') \vee c' = \{\} \wedge s'' = s'\big) \\
&\qquad\qquad \wedge \mathbf{KDo_0}(\mathcal{T}(\pi, c, c'), s, s')\big] \rightarrow \phi[s'']
\end{aligned}$$

Case-splitting on the disjunction, we see that:

$$s'' = do(c', s') \wedge c' \neq \{\} \rightarrow \big(\phi[s''] \equiv \mathcal{R}(\phi, c')[s']\big)$$

$$s'' = s' \wedge c' = \{\} \rightarrow \big(\phi[s''] \equiv \mathcal{R}(\phi, c')[s']\big)$$

This allows us to remove the variable s'' from the consequent of the implication, making it redundant in the antecedent and allowing us to eliminate it entirely. Folding the quantification over s' back into the $\mathbf{PKnows_0}$ macro completes the proof:

$$\begin{aligned}
&\mathbf{PKnows_0}(\pi, \phi, do(c,s)) \\
&\equiv \forall s'', c', s' : \big[\big(c' \neq \{\} \wedge s'' = do(c', s') \vee c' = \{\} \wedge s'' = s'\big) \\
&\qquad\qquad \wedge \mathbf{KDo_0}(\mathcal{T}(\pi, c, c'), s, s')\big] \rightarrow \mathcal{R}(\phi, c')[s'] \\
&\equiv \forall c', s' : \mathbf{KDo_0}(\mathcal{T}(\pi, c, c'), s, s') \rightarrow \mathcal{R}(\phi, c')[s'] \\
&\equiv \forall c' : \mathbf{PKnows_0}(\mathcal{T}(\pi, c, c'), \mathcal{R}(\phi, c'), s)
\end{aligned}$$

This is the theorem as required. □

Lemma 13. *For any epistemic path π:*

$$\mathcal{D} \cup \mathcal{D}_K^{obs} \models \mathbf{KDo_0}(\mathcal{T}(\pi, \{\}, \{\}), s, s') \rightarrow \mathbf{KDo_0}(\pi, s, s')$$

Proof. By a case analysis on the epistemic path operators. For the base case of an individual agent, we have:

$$\begin{aligned}
\mathcal{T}(agt, \{\}, \{\}) &= x \Leftarrow \{\}\,;\, \exists z\,;\, ?Obs(agt, x) = z\,;\, agt\,; \\
&\qquad \exists x\,;\, ?Legal(x) \vee x = \{\}\,;\, ?Obs(agt, x_a) = z\,;\, ?x = \{\} \\
&= \exists z\,;\, ?z = \{\}\,;\, agt\,;\, ?Legal(\{\}) \vee \{\} = \{\}\,;\, ?z = \{\} \\
&= agt
\end{aligned}$$

So the hypothesis is clearly entailed. For the $?\phi$ case:

$$\begin{aligned}
\mathcal{T}(?\phi, \{\}, \{\}) &= x \Leftarrow \{\}\,;\, ?\mathcal{R}(\phi, x)\,;\, ?x = \{\} \\
&= ?\mathcal{R}(\phi, \{\}) \\
&= ?\phi
\end{aligned}$$

So the hypothesis is clearly entailed. For the $\exists z$ case:

$$\begin{aligned}\mathcal{T}(\exists z, \{\}, \{\}) &= x \Leftarrow \{\}\,;\, \exists z\,;\, ?x = \{\} \\ &= \exists z\end{aligned}$$

So the hypothesis is clearly entailed. The inductive cases are then straightforward, by choosing $x = \{\}$ uniformly whenever $\exists x$ is encountered in the translated path. □

Theorem 16. *For any epistemic path π:*

$$\mathcal{D} \cup \mathcal{D}_K^{obs} \models \mathbf{PKnows_0}(\pi, \phi, \mathcal{E}^1(s)) \rightarrow \mathbf{PKnows_0}(\pi, \phi, s)$$

Proof. Expanding the macros, we have:

$$\big(\forall s'' : \mathbf{KDo_0}(\pi, do(\{\}, s), s'') \rightarrow \phi[s'']\big) \;\rightarrow\; \big(\forall s' : \mathbf{KDo_0}(\pi, s, s') \rightarrow \phi[s']\big)$$

Using equation 14 on the LHS gives:

$$\begin{aligned}\big(\forall s'' : \exists c', s' : \big(c' \neq \{\} \wedge s'' = do(c', s') \;\vee\; c' = \{\} \wedge s'' = s'\big) \\ \wedge \mathbf{KDo_0}(\mathcal{T}(\pi, \{\}, c'), s, s') \rightarrow \phi[s'']\big) \;\rightarrow\; \big(\forall s' : \mathbf{KDo_0}(\pi, s, s') \rightarrow \phi[s']\big)\end{aligned}$$

We can weaken the antecedent by dropping the $do(c', s')$ case; if the implication holds with this weaker antecedent then it must hold in its stronger form above. We obtain:

$$\begin{aligned}\big(\forall s'' : \exists c', s' : s'' = s' \wedge c' = \{\} \wedge \mathbf{KDo_0}(\mathcal{T}(\pi, \{\}, c'), s, s') \rightarrow \phi[s'']\big) \;\rightarrow \\ \big(\forall s' : \mathbf{KDo_0}(\pi, s, s') \rightarrow \phi[s']\big)\end{aligned}$$

Simplifying away the variables s'' and c' gives:

$$\big(\forall s' : \mathbf{KDo_0}(\mathcal{T}(\pi, \{\}, \{\}), s, s') \rightarrow \phi[s']\big) \;\rightarrow\; \big(\forall s' : \mathbf{KDo_0}(\pi, s, s') \rightarrow \phi[s']\big)$$

This implication is a trivial consequence of lemma 13, so the theorem holds. □

Theorem 17. *Given a basic action theory $\mathcal{D}$ and a uniform formula ϕ:*

$$\mathcal{D} \cup \mathcal{D}_K^{obs} \models \mathbf{PKnows}(\pi, \phi, s) \equiv \mathcal{R}(\mathbf{PKnows}(\pi, \phi, s))$$

Proof. By induction on situation terms and the natural numbers. In the base case of S_0 we have:

$$\mathcal{R}(\mathbf{PKnows}(\pi, \phi, S_0)) \stackrel{\text{def}}{=} \mathcal{P}(\mathbf{PKnows_0}(\pi, \phi), Empty)[S_0]$$

The key part of this proof is to demonstrate that for any n:

$$\mathcal{P}^n(\mathbf{PKnows_0}(\pi,\phi),Empty)[S_0] \equiv \mathbf{PKnows_0}(\pi,\phi,\mathcal{E}^n(S_0))$$

Begin with $n=1$, and we have:

$$\mathcal{P}^1(\mathbf{PKnows_0}(\pi,\phi),Empty)[S_0] \equiv \\ \mathbf{PKnows_0}(\pi,\phi,S_0) \wedge \forall c:\ Empty(c) \rightarrow \mathbf{PKnows}(\pi,\phi,do(c,S_0))$$

By the definition of $Empty$, we need only consider $c=\{\}$ and this is equivalent to:

$$\mathbf{PKnows_0}(\pi,\phi,S_0) \wedge \mathbf{PKnows}(\pi,\phi,do(\{\},S_0)) \\ \equiv \mathbf{PKnows_0}(\pi,\phi,S_0) \wedge \mathbf{PKnows_0}(\pi,\phi,\mathcal{E}^1(S_0))$$

By Theorem 16 this is equivalent to $\mathbf{PKnows_0}(\pi,\phi,\mathcal{E}^1(S_0))$. A similar construction will produce the general case for $\mathcal{P}^n$ and $\mathcal{E}^n$. Since $\mathcal{P}(\mathbf{PKnows_0}(\pi,\phi),Empty)$ is equivalent to $\mathcal{P}^n(\mathbf{PKnows_0}(\pi,\phi),Empty)$ for all n, it is also equivalent to the definition of **PKnows** and the regression rule is an equivalence as required.

In the $do(c,s)$ case, we can repeat the above reasoning to demonstrate that the persistence condition accounts for all empty actions inserted *after* c. Pushing the application of $\mathcal{E}^n$ past c, we obtain:

$$\mathbf{PKnows}(\pi,\phi,do(c,s)) \equiv \bigwedge_{n\in\mathbb{N}} \mathcal{P}(\mathbf{PKnows_0}(\pi,\phi),Empty)[do(c,\mathcal{E}^n(s))]$$

Applying the regression rule for $\mathbf{PKnows_0}$ to handle c, we obtain:

$$\mathbf{PKnows}(\pi,\phi,do(c,s)) \equiv \bigwedge_{n\in\mathbb{N}} \mathcal{R}(\mathcal{P}(\mathbf{PKnows_0}(\pi,\phi),Empty),c)[\mathcal{E}^n(s)]$$

Finally, we need the following property of $\mathcal{Z}(\mathbf{PKnows_0}(\pi,\phi,s))$, which is a direct consequence of the definition of **PKnows** :

$$\bigwedge_{n\in\mathbb{N}} \mathbf{PKnows_0}(\pi,\phi,\mathcal{E}^n(s)) \equiv \mathbf{PKnows}(\pi,\phi,s) \equiv \mathcal{Z}(\mathbf{PKnows_0}(\pi,\phi,s))$$

From Theorem 16 and the definition of $\mathcal{R}(\mathbf{PKnows_0}(\pi,\phi,s))$, the expression generated by $\mathcal{R}(\mathcal{P}(\mathbf{PKnows_0}(\pi,\phi),Empty),c)$ will be in the form of a finite conjunction of $\mathbf{PKnows_0}$ statements, so we can apply $\mathcal{Z}$ to remove the infinite conjunction by capturing it within **PKnows** from the inductive hypothesis:

$$\mathbf{PKnows}(\pi,\phi,do(c,s))) \equiv \mathcal{Z}(\mathcal{R}(\mathcal{P}(\mathbf{PKnows_0}(\pi,\phi),Empty),c)[s])$$

This is the theorem as required. □

Theorem 18. *Let $\mathcal{D}_{sync}$ be a synchronous basic action theory, then:*

$$\mathcal{D}_{sync} \cup \mathcal{D}_K^{obs} \models \forall s : \mathbf{PKnows}(\pi, \phi, s) \equiv \mathbf{PKnows_0}(\pi, \phi, s)$$

Proof. It suffices to show that:

$$\mathcal{D}_{sync} \cup \mathcal{D}_K^{obs} \models \mathbf{PKnows_0}(\pi, \phi, s) \rightarrow \mathbf{PKnows_0}(\pi, \phi, \mathcal{E}^1(s))$$

Then by Theorem 16 we have that $\mathbf{PKnows_0}(\pi, \phi, s)$ is enough to establish $\mathbf{PKnows_0}(\pi, \phi, \mathcal{E}^n(s))$ for any n, which establishes the infinite conjunction in the definition of $\mathbf{PKnows}(\pi, \phi, s)$ as required. Regressing $\mathbf{PKnows_0}(\pi, \phi, \mathcal{E}^1(s))$:

$$\mathcal{R}(\mathbf{PKnows_0}(\pi, \phi, do(\{\}, s))) \Rightarrow \forall c : \mathbf{PKnows_0}(\mathcal{T}(\pi, \{\}, c), \mathcal{R}(\phi, c), s)$$

$\mathcal{T}(\pi, \{\}, c)$ will have the following form:

$$\mathcal{T}(\pi, \{\}, c) \Rightarrow \quad x \Leftarrow \{\}\,;\, \mathcal{T}_a(\pi, x)\,;\, ?x = c$$

For x to take on a new value while traversing this path, it will have to cross one of the regressed agt steps in $\mathcal{T}_a(\pi, x)$, which have the following form:

$$z \Leftarrow Obs(agt, x)\,;\, agt\,;\, \exists x\,;\, ?Legal(x) \vee x = \{\}\,;\, ?Obs(agt, x) = z$$

Entering this path with x set to $\{\}$ will bind z to $\{\}$. x can then take on any new value that is legal and has $Obs(agt, x) = z = \{\}$. But the domain is synchronous, so by definition there are no such legal actions, and x therefore remains set to $\{\}$ along the entire path.

For any value of c other than $\{\}$, there will be no situations reachable by this regressed path and $\mathbf{PKnows_0}$ will be vacuously true. We can thus simplify away the quantification over c to get:

$$\mathbf{PKnows_0}(\mathcal{T}(\pi, \{\}, \{\}), \mathcal{R}(\phi, \{\}), s)$$

Since x is aways bound to $\{\}$, the tests in the regressed agt steps in $\mathcal{T}_a(\pi, x)$ are always satisfied. Likewise, the regressed $?\phi$ steps in $\mathcal{T}_a(\pi, x)$ always have the form $?\mathcal{R}(\phi, x)$. Since $\mathcal{R}(\phi, \{\})$ is always equivalent to ϕ, we conclude that in synchronous domains the path $\mathcal{T}(\pi, \{\}, \{\})$ is precisely equivalent the path π. This gives us the equivalence between $\mathbf{PKnows}$ and $\mathbf{PKnows_0}$ as required. □

Lemma 4. *For any agt and ϕ:*

$$\mathcal{D} \cup \mathcal{D}_K^{obs} \models \mathbf{PKnows}(agt, \phi, S_0) \equiv \mathbf{PKnows_0}(agt, \mathcal{P}(\phi, LbU(agt)), S_0)$$

Proof. Recall the regression rule for $\mathbf{PKnows}$ at S_0:

$$\mathbf{PKnows}(agt, \phi, S_0) \equiv \mathcal{P}(\mathbf{PKnows_0}(agt, \phi), Empty)[S_0]$$

Begin by considering the sequence of calculations required to calculate the regression

of $\mathcal{P}^1(\mathbf{PKnows_0}(agt, \phi))$. First, we perform some simplification on $\mathcal{T}(agt, \{\}, c')$:

$$\begin{aligned}\mathcal{T}(agt, \{\}, c') = \; & x \Leftarrow \{\} \,;\, z \Leftarrow Obs(agt, x) \,;\, agt \,; \\ & \exists x \,;\, ?Legal(x) \vee x = \{\} \,;\; ?Obs(agt, x) = z \,;\, ?x = c' \\ = \; & z \Leftarrow Obs(agt, \{\}) \,;\, agt \,;\, ?Legal(c') \vee c' = \{\} \,;\, ?Obs(agt, c') = z \\ = \; & agt \,;\, ?Obs(agt, c') = \{\} \wedge \big(Legal(c') \vee c' = \{\}\big) \\ = \; & agt \,;\, ?\left(LbU(agt, c') \vee c' = \{\}\right)\end{aligned}$$

Now we can use this in the calculation of $\mathcal{P}^1(\mathbf{PKnows_0}(agt, \phi))$, recalling the simplifications used in Theorem 17:

$$\begin{aligned}& \mathcal{P}^1(\mathbf{PKnows_0}(agt, \phi))[S_0] \equiv \; \mathcal{R}(\mathbf{PKnows}_0(agt, \phi, do(\{\}, S_0))) \\ & \equiv \; \forall c' : \mathbf{PKnows_0}(\mathcal{T}(agt, \{\}, c'), \mathcal{R}(\phi, c'), S_0) \\ & \equiv \; \forall c' : \mathbf{PKnows_0}(agt \,;\, ?\left(LbU(agt, c') \vee c' = \{\}\right), \mathcal{R}(\phi, c'), S_0) \\ & \equiv \; \mathbf{PKnows_0}(agt, \forall c' : \left(LbU(agt, c') \vee c' = \{\}\right) \rightarrow \mathcal{R}(\phi, c'), S_0) \\ & \equiv \; \mathbf{PKnows_0}(agt, \phi \wedge \forall c' : LbU(agt, c') \rightarrow \mathcal{R}(\phi, c'), S_0) \\ & \equiv \; \mathbf{PKnows_0}(agt, \mathcal{P}^1(\phi, LbU(agt)), S_0)\end{aligned}$$

Using the same construction, we can show that in general:

$$\begin{aligned}\mathcal{P}^n(\mathbf{PKnows_0}(agt, \phi))[S_0] &\equiv \mathbf{PKnows_0}(agt, \mathcal{P}^1(\mathcal{P}^{n-1}(\phi, LbU(agt)), LbU(agt))) \\ &\equiv \mathbf{PKnows_0}(agt, \mathcal{P}^n(\phi, LbU(agt)))[S_0]\end{aligned}$$

Clearly the fixpoint calculation in the regression of **PKnows** at S_0 is the same as the fixpoint calculation used to find $\mathcal{P}(\phi, LbU(agt))$. Therefore, we have the required:

$$\mathbf{PKnows}(agt, \phi, S_0) \equiv \mathbf{PKnows_0}(agt, \mathcal{P}(\phi, LbU(agt)), S_0)$$

□

Lemma 5. *For any agt, ϕ, c and s:*

$$\begin{aligned}\mathbf{PKnows}(agt, \phi, do(c, s)) \equiv \; & \exists z : Obs(agt, c, s) = z \\ & \wedge \; [z = \{\} \rightarrow \mathbf{PKnows}(agt, \mathcal{P}(\phi, LbU(agt)), s)] \\ & \big[z \neq \{\} \rightarrow \mathbf{PKnows}(agt, \forall c' : \big(Legal(c') \wedge Obs(agt, c') = z\big) \\ & \qquad \rightarrow \mathcal{R}(\mathcal{P}(\phi, LbU(agt)), c), s)]\end{aligned}$$

Proof. Repeating the calculations from Lemma 4 on $\mathcal{P}(\mathbf{PKnows_0}(agt, \phi))[do(c, s)]$, and pushing the application of $\mathcal{E}^n$ past the actions c, we obtain the following:

$$\mathbf{PKnows}(agt, \phi, do(c, s)) \equiv \bigwedge_{n \in \mathbb{N}} \mathbf{PKnows_0}(agt, \mathcal{P}(\phi, LbU(agt)), do(c, \mathcal{E}^n(s)))$$

Regressing the RHS over the actions c, we obtain:

$$\begin{aligned}
&\mathbf{PKnows}(agt, \phi, do(c, s)) \\
&\equiv \forall c' : \bigwedge_{n \in \mathbb{N}} \mathbf{PKnows_0}(\mathcal{T}(agt, c, c'), \mathcal{R}(\mathcal{P}(\phi, LbU(agt)), c'), \mathcal{E}^n(s)) \\
&\equiv \forall c' : \mathbf{PKnows}(\mathcal{T}(agt, c, c'), \mathcal{R}(\mathcal{P}(\phi, LbU(agt)), c'), s)
\end{aligned}$$

Now, let us expand and re-arrange $\mathcal{T}(agt, c, c')$:

$$\begin{aligned}
\mathcal{T}(agt, c, c') = {} & x \Leftarrow c\,;\, z \Leftarrow Obs(agt, x)\,;\, agt\,; \\
& \exists x\,;\, ?Legal(x) \vee x = \{\}\,;\; ?Obs(agt, x) = z\,;\, ?x = c' \\
= {} & z \Leftarrow Obs(agt, c)\,;\, agt\,;\, ?Legal(c') \vee c' = \{\}\,;\, ?Obs(agt, c') = z \\
= {} & z \Leftarrow Obs(agt, c)\,; \\
& \big(z = \{\}\,;\, agt\,;\, ?Legal(c') \vee c' = \{\}\,;\, ?Obs(agt, c') = \{\}\big) \\
& \cup \big(z \neq \{\}\,;\, agt\,;\, ?Legal(c') \vee c' = \{\}\,;\, ?Obs(agt, c') = z\big) \\
= {} & z \Leftarrow Obs(agt, c)\,;\, \big(z = \{\}\,;\, agt\,;\, ?c' = \{\}\big) \\
& \cup \big(z = \{\}\,;\, agt\,;\, ?LbU(agt, c')\big) \\
& \cup \big(z \neq \{\}\,;\, agt\,;\, ?Legal(c') \wedge Obs(agt, c') = z\big)
\end{aligned}$$

Substituting this back into the RHS, we can bring the leading tests outside the macro and split the $\cup$ into a conjunction to give:

$$\begin{aligned}
\mathbf{PKnows}(agt, \phi, do(c, s)) \equiv {} & \forall c' : \exists z : Obs(agt, c, s) = z \\
& \wedge \mathbf{PKnows}((?z = \{\};\, agt), \mathcal{R}(\mathcal{P}(\phi, LbU(agt)), \{\}), s) \\
& \wedge \mathbf{PKnows}(\big(?z = \{\}; agt; ?LbU(agt, c')\big), \mathcal{R}(\mathcal{P}(\phi, LbU(agt)), c'), s) \\
& \wedge \mathbf{PKnows}(\big(?z \neq \{\}; agt; ?Legal(c') \wedge Obs(agt, c') = z\big), \mathcal{R}(\mathcal{P}(\phi, LbU(agt)), c'), s)
\end{aligned}$$

Extracting the remaining tests from these paths, removing regression over the empty action, and pushing the quantification over c' into its narrowest scope:

$$\begin{aligned}
\mathbf{PKnows}(agt, \phi, do(c, s)) \equiv {} & \exists z : Obs(agt, c, s) = z \\
& \wedge [z = \{\} \rightarrow \mathbf{PKnows}(agt, \mathcal{P}(\phi, LbU(agt)), s)] \\
& \wedge \big[z = \{\} \rightarrow \mathbf{PKnows}(agt, \forall c' : LbU(agt, c') \rightarrow \mathcal{R}(\mathcal{P}(\phi, LbU(agt)), c'), s)\big] \\
& \wedge \big[z \neq \{\} \rightarrow \mathbf{PKnows}(agt, \forall c' : \big(Legal(c') \wedge Obs(agt, c') = z\big) \\
& \qquad \rightarrow \mathcal{R}(\mathcal{P}(\phi, LbU(agt)), c'), s)\big]
\end{aligned}$$

To complete the proof, we need the following property of the persistence condition, which follows directly from its definition:

$$\mathcal{D} \models (\forall c : \alpha[c, s] \rightarrow \mathcal{R}(\mathcal{P}(\phi, \alpha), c)[s]) \equiv \mathcal{P}(\phi, \alpha)[s]$$

Using this we see that the two $z = \{\}$ clauses are equivalent, and we can drop the more complicated one to get the theorem as required. □

Lemma 6. *For any agt, ϕ and s:*

$$\mathcal{D} \cup \mathcal{D}_K^{obs} \models \mathbf{Knows}(agt, \phi, s) \equiv \mathbf{Knows}(agt, \mathcal{P}(\phi, LbU(agt)), s)$$

Proof. By induction on situations, using the regression rules for knowledge, and the following straightforward properties of the persistence condition:

$$\forall s' : \mathcal{P}(\phi, \alpha)[s] \wedge s \leq_\alpha s' \rightarrow \mathcal{P}(\phi, \alpha)[s']$$

$$\mathcal{P}(\mathcal{P}(\phi, \alpha), \alpha)[s] \equiv \mathcal{P}(\phi, \alpha)[s]$$

For $s = S_0$ the regression rule in equation (7.11) gives us the following:

$$\begin{aligned}\mathbf{Knows}(agt, \phi, S_0) \equiv & \mathbf{Knows_0}(agt, \mathcal{P}(\phi, LbU(agt)), S_0) \\ \equiv & \forall s' : K_0(agt, s', S_0) \rightarrow \mathcal{P}(\phi, LbU(agt))[s']\end{aligned}$$

Which, by the above properties of $\mathcal{P}$, yields:

$$\mathbf{Knows}(agt, \phi, S_0) \equiv \forall s', s'' : K_0(agt, s', S_0) \wedge s' \leq_{LbU(agt)} s'' \rightarrow \mathcal{P}(\phi, LbU(agt))[s'']$$

This matches the form of the **Knows** macro, and can be restructured to give the required:

$$\mathbf{Knows}(agt, \phi, S_0) \equiv \mathbf{Knows}(agt, \mathcal{P}(\phi, LbU(agt)), S_0)$$

For the $do(c, s)$ the inductive hypothesis gives us:

$$\mathbf{Knows}(agt, \phi, s) \equiv \mathbf{Knows}(agt, \mathcal{P}(\phi, LbU(agt)), s)$$

We have two sub-cases to consider. If $Obs(agt, c, s) = \{\}$ then the regression rule in equation (7.10) gives us:

$$\mathbf{Knows}(agt, \phi, do(c, s)) \equiv \mathbf{Knows}(agt, \phi, s)$$

$$\mathbf{Knows}(agt, \mathcal{P}(\phi, LbU(agt)), do(c, s)) \equiv \mathbf{Knows}(agt, \mathcal{P}(\phi, LbU(agt)), s)$$

These can be directly equated using the inductive hypothesis, so the theorem holds in this case. Alternately, if $Obs(agt, c, s) \neq \{\}$ then the regression rule gives:

$$\begin{aligned}&\mathbf{Knows}(agt, \phi, do(c, s)) \equiv \exists o : Obs(agt, c, s) = o \wedge \\ &\quad \mathbf{Knows}(agt, \forall c' : Legal(c') \wedge Obs(agt, c') = o \rightarrow \mathcal{R}(\mathcal{P}(\phi, LbU(agt)), c'), s)\end{aligned}$$

$$\begin{aligned}&\mathbf{Knows}(agt, \mathcal{P}(\phi, LbU(agt)), do(c, s)) \equiv \exists o : Obs(agt, c, s) = o \wedge \\ &\qquad \mathbf{Knows}(agt, \forall c' : Legal(c') \wedge Obs(agt, c') = o \\ &\qquad\qquad \rightarrow \mathcal{R}(\mathcal{P}(\mathcal{P}(\phi, LbU(agt)), LbU(agt)), c'), s)\end{aligned}$$

Simplifying the second equation using the properties of $\mathcal{P}$ gives:

$$\begin{aligned}&\mathbf{Knows}(agt, \mathcal{P}(\phi, LbU(agt)), do(c, s)) \equiv \exists o : Obs(agt, c, s) = o \wedge \\ &\quad \mathbf{Knows}(agt, \forall c' : Legal(c') \wedge Obs(agt, c') = o \rightarrow \mathcal{R}(\mathcal{P}(\phi, LbU(agt)), c'), s)\end{aligned}$$

This matches the equivalence for $\mathbf{Knows}(agt, \phi, do(c, s))$, as required. □

Appendix B

Available Software Implementations

Three major software implementations have been developed during the course of this research. They are each made available under the terms of the GNU General Public License, and are available for download at the author's website:

`http://www.rfk.id.au/research/thesis/`

Each software package comes with comprehensive instructions on running the code and reproducing the results found in this thesis. The following is a brief description of each system.

MIndiGolog v1: This is the MIndiGolog implementation described in Chapter 3, which performs online execution planning but is limited to synchronous domains. This software runs on the Mozart platform version 1.3.2 or later. Its key feature is the use of Mozart's parallel search functionality to distribute the execution planning workload.

MIndiGolog v2: This is the MIndiGolog implementation described in Chapter 5, which produces joint executions as the output of its planning process, but is limited to only offline planning. It is able to render a graphical representation of joint executions in the DOT graph description language, from which the diagrams in Chapter 5 were generated. This version also runs on the Mozart platform version 1.3.2 or later.

PKnows: This is our preliminary implementation of an epistemic reasoning system using the techniques developed in Chapters 7 and 8. It is implemented using SWI-Prolog to perform symbolic manipulation, calling a modified version of the PDL prover from the Tableaux Workbench suite [2] to handle the resulting modal logic queries.

Appendix C

Axioms for the "Cooking Agents"

This appendix provides the axioms for the "cooking agents" example domain used in Chapters 3 and 5. While the different chapters use slightly different variants of the domain, the major details are unchanged between chapters.

Synchronous, with Time and Natural Actions

In this domain there are three agents named Jon, Jim and Joe:

$$\forall agt: \ agt = Jim \vee agt = Jon \vee agt = Joe$$

There are various types of ingredient and utensil, and types are represented explicitly as terms such as $Lettuce$ and $Bowl$. Individual objects of these types are named e.g. $Lettuce1$, $Bowl2$: We have a rigid predicate $ObjIsType(obj, typ)$ that relates these two sorts of object. It is defined as the completion of the following clauses:

$$IsType(obj, Bowl) \rightarrow obj = Bowl1 \vee obj = Bowl2 \vee Bowl3$$
$$IsType(obj, Board) \rightarrow obj = Board1 \vee obj = Board2$$
$$IsType(obj, Egg) \rightarrow Egg1$$
$$IsType(obj, Tomato) \rightarrow obj = Tomato1 \vee obj = Tomato2 \vee obj = Tomato3$$
$$IsType(obj, Lettuce) \rightarrow obj = Lettuce1 \vee obj = Lettuce2$$
$$IsType(obj, Carrot) \rightarrow obj = Carrot1 \vee obj = Carrot2 \vee obj = Carrot3$$
$$IsType(obj, Cheese) \rightarrow obj = Cheese1 \vee obj = Cheese2$$

Different object super-types are identified using:

$$IsContainer(obj) \equiv IsType(obj, Bowl) \vee IsType(obj, Board)$$
$$IsIngredient(obj) \equiv IsType(obj, Egg) \vee IsType(obj, Lettuce) \vee \ldots$$

The available actions are $release$, $acquire$, $placeIn$, and $transer$, along with

beginTask and *endTask*. There are two tasks: *chop*(*cont*) chops the contents of a container, and *mix*(*cont*, *t*) mixes the contents of a container for the given time.

We have the following fluents and successor state axioms. An ingredient is used if the agent places it in some container:

$$Used(obj, do(c\#t, s)) \equiv IsIngredient(obj) \\ \wedge (\exists agt, cnt : placeIn(agt, obj, cnt) \in c) \vee Used(obj, s)$$

An agent has an object after he acquires it, and ceases to have it when it is released or becomes used:

$$HasObject(agt, obj, do(c\#t, s)) \equiv acquire(agt, obj) \in c \\ \vee HasObject(agt, obj, s) \wedge \neg (release(agt, obj) \in c \\ \vee IsIngredient(obj) \wedge \exists cnt : placeIn(agt, obj, cnt) \in c)$$

The contents of a container is simply the set of things that have been placed into it. For this simple example, we do not represent the *state* of those ingredients, e.g. mixed or chopped:

$$Contents(obj, cnts, do(c\#t, s)) \equiv (\exists cnts_n, cnts_o : \mathbf{NewConts}(obj, cnts_n, c, s) \\ \wedge Contents(obj, cnts_o, s) \wedge cnts = cnts_n \cup cnts_o) \\ \vee (cnts = \{\} \wedge \mathbf{LostContents}(obj, c)) \\ \vee (Contents(obj, cnts, s) \wedge \\ \neg(\exists cnts_n, cnts_o : \mathbf{NewConts}(obj, cnts_n, c, s) \vee \mathbf{LostContents}(obj, c))$$

$$\mathbf{NewConts}(obj, cnts, c, s) \equiv \exists agt, igr : placeIn(agt, igr, obj) \in c \wedge cnts = \{igr\} \\ \vee \exists agt, obj' : transfer(agt, obj', obj) \in c \wedge Contents(obj', cbts, s)$$

$$\mathbf{LostConts}(obj, c) \equiv \exists agt, obj' : transfer(agt, obj, obj') \in c$$

An agent can be doing a long-running task, with time t_r remaining until completion. The rigid function *duration*(*tsk*) gives the running time of a task:

$$DoingTask(agt, tsk, t_r, do(c\#t, s)) \equiv \\ beginTask(agt, tsk) \in c \wedge t_r = duration(tsk) \\ \vee \exists t'_r : DoingTask(agt, tsk, t'_r, s) \wedge t_r = t'_r - t \wedge endTask(agt, tsk) \notin c$$

The possibility axioms for individual actions are:

$$Poss(acquire(agt, obj)\#t, s) \equiv \neg Used(obj) \wedge \neg\exists agt' : HasObject(agt, obj, s)$$

$$Poss(release(agt, obj)\#t, s) \equiv HasObject(agt, obj, s)$$

$$Poss(placeIn(agt, obj, cnt)\#t, s) \equiv HasObject(agt, obj, s) \wedge HasObject(agt, cnt, s)$$

$$Poss(transfer(agt, cnt, cont')\#t, s) \equiv$$
$$HasObject(agt, cnt, s) \wedge HasObject(agt, cnt', s)$$

$$Poss(beginTask(agt, tsk)\#t, s) \equiv$$
$$\exists cnt, t : tsk = mix(cnt, t) \wedge HasObject(agt, cnt, s) \wedge ObjIsType(cnt, Bowl)$$
$$\vee \exists cnt : tsk = chop(cnt) \wedge HasObject(agt, cnt, s) \wedge ObjIsType(cnt, Board)$$

$$Poss(endTask(agt, tsk)\#t, s) \equiv \exists t_r : DoingTask(agt, tsk, t_r, s) \wedge t = start(s) + t_r$$

Concurrent actions are possible if they are all individually possible and no pair of action is in conflict:

$$Poss(c\#t, s) \equiv \forall a, a' \in c : Poss(a\#t, s) \wedge Poss(a'\#t, s) \wedge \neg Conflicts(a, a', s)$$

Actions conflict if they are performed by the same agent, or are attempts to acquire the same resource:

$$Conflicts(a, a', s) \equiv actor(a) = actor(a')$$
$$\vee \exists agt, agt', obj : a = acquire(agt, obj) \wedge a' = acquire(agt', obj)$$

Initially all containers are empty, no-one has any objects, and all ingredients apart from possibly the egg are not used:

$$\forall cnt : IsContainer(cnt) \rightarrow Contents(cnt, \{\}, S_0)$$
$$\forall agt, obj : \neg HasObject(agt, obj, S_0)$$
$$\forall igr : ObjIsType(igr, Egg) \vee \neg Used(igr, S_0)$$

These axioms suffice for the example domain used in Chapter 3

Asynchronous, without Time or Natural Actions

For Chapter 5 we drop the temporal component, and collapse the tasks *mix* and *chop* into primitive actions:

$$Poss(mix(agt, cnt), s) \equiv HasObject(agt, cnt, s) \wedge ObjIsType(cnt, Bowl)$$

$$Poss(chop(agt, cnt), s) \equiv HasObject(agt, cnt, s) \wedge ObjIsType(cnt, Board)$$

We introduce a sensing action $checkFor(agt, typ)$ which determines whether all objects of that type are unused:

$$Poss(checkFor(agt, typ), s) \equiv \top$$

$$SR(checkFor(agt, typ), s) = r \equiv$$
$$r = "T" \wedge \forall obj : ObjIsType(obj, typ) \rightarrow \neg Used(obj, s)$$
$$\vee\, r = "F" \wedge \exists obj : ObjIsType(obj, typ) \wedge Used(obj, s)$$

We adopt the $CanObs/CanSense$ axioms for observability and make all actions private except *acquire* and *release*:

$$CanObs(agt, a, s) \equiv actor(a) = agt \vee \exists agt', obj : a = acquire(agt', obj)$$
$$\vee\, \exists agt', obj : a = release(agt', obj)$$

$$CanSense(agt, a, s) \equiv actor(a) = agt$$

Finally, we identify independent actions as those that deal with different objects, which much be axiomatised by enumerating the each possible case. We will not present such an enumeration here.

These axioms suffice for the example domain in Chapter 5.