Automated Norm Synthesis in Planning Environments

George Dimitri Christelis

Doctor of Philosophy

Centre for Intelligent Systems and their Applications

School of Informatics

University of Edinburgh

2011

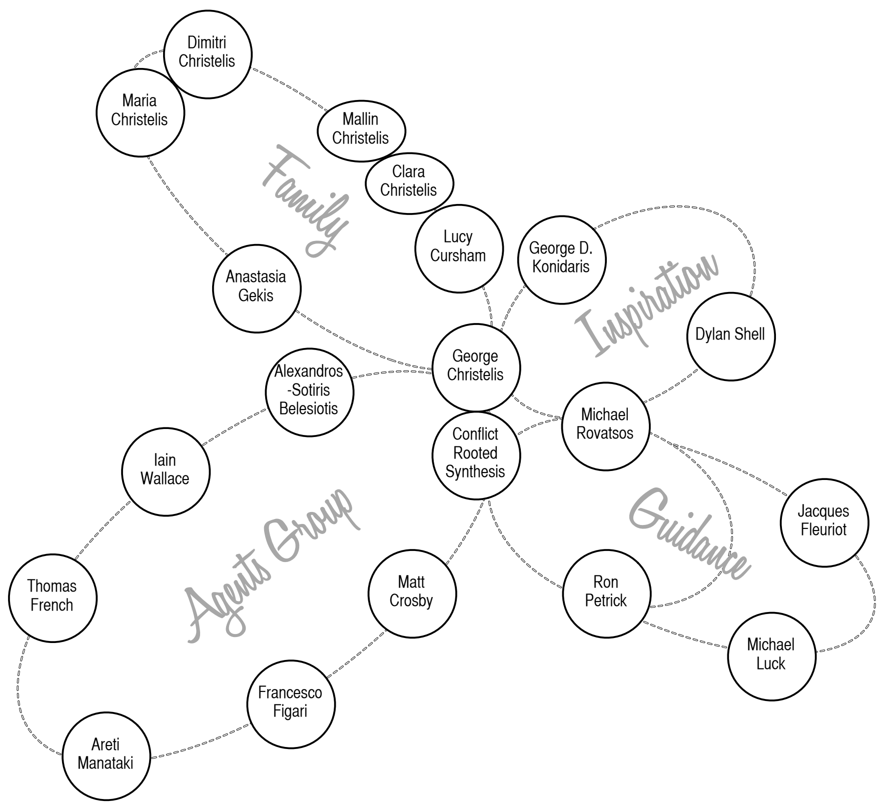
Abstract

Multiagent systems offer a design paradigm used to conceptualise and implement systems composed of autonomous agents. Autonomy facilitates proactive independent behaviour yet in practice agents are constrained in order to ensure the system satisfies a desired social objective. Explicit constraints on agent behaviour, in the form of social norms, encourage this desirable system behaviour, yet research has largely focused on norm representation languages and protocols for norm proposal and adoption. The fundamental problem of how to automate the process of norm synthesis has largely been overlooked with norms assumed provided by the designer. Previous work has shown that automating the design of social norms is intractable in the worst case. Existing approaches, relying on state space enumerations, are effective for small systems but impractical for larger ones. Furthermore, they do not produce a set of succinct, general norms but rather a large number of state-specific restrictions.

This work presents conflict-rooted synthesis, an automated norm synthesis approach that utilises a planning-based action schemata to overcome these limitations. These action schemata facilitate localised searches around specifications of undesirable states, using representations of sets of system states to avoid a full state enumeration. The proposed technique produces concise, generalised social norms that are applicable in multiple system states while also providing guarantees that agents are still able to achieve their original goals in the constrained system. To improve efficiency a set of theoretically sound, domain-independent optimisations are presented that reduce the state space searched without compromising the quality of the norms synthesised.

A comparison with an alternative model checking based technique illustrates the advantages and disadvantages of our approach, while an empirical evaluation highlights the improved efficiency and quality of norms it produces at the cost of a less expressive specification of undesirable states. We empirically investigate the effectiveness of each of the proposed optimisations using a set of benchmark domains, quantifying how successful each of them is at reducing search complexity in practice. The results show that, with all optimisations enabled, conflict-rooted synthesis produces more generally applicable and succinct norms and consumes fewer system resources. Additionally, we show that this approach synthesises norms in systems where the competing approach is intractable. We provide a discussion of our approach, highlighting the impact our abstract search approach has on the fields of multiagent systems and automated planning, and discuss the limitations and assumptions we have made. We conclude with a presentation of future work.

Acknowledgements



Michael Rovatsos. My excellent primary supervisor. You make the complex seem simple, and display the eloquence to make them interesting. I respect you as a supervisor and researcher. Thank you.

Ron Petrick. My secondary supervisor. Your input and guidance provided balance and perspective. It is an honour to be your first graduate.

Jacques Fleuriot and Michael Luck. My examiners and first victims. Your high-level insights and low-level rigour improved this document immeasurably.

The Agents Group: Alexandros-Sotiris Belesiotis, Francesco Figari, Iain Wallace, Matt Crosby, Tommy French and Areti Manataki. What a journey! Thank you for your feedback, support, criticism and general revelry. The Goggles are free now...

George Konidaris for providing the necessary pokes and prods. I almost missed the opportunity. Thanks cuz. Dylan Shell, for saying “Do it...”, over and over again.

My family. My sister, Anastasia Gekis, for paving the way, and then ushering me along. My mother, Maria Christelis, for absolutely everything.

Thanks to my sponsors, the Commonwealth Scholarship and Fellowship Plan.

Finally, Lucy. Not once did you think this was a bad idea. This would never have happened without your support, encouragement, perspective and sacrifice. Thank you.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*George Dimitri Christelis*)

For Mitsos, who was so proud of everything I did.

Table of Contents

Chapter 1

Introduction

The complexity involved in the design and implementation of modern software systems continues to grow, partly due to a continued increase in computational ubiquity and distributed computation. Computer systems no longer solely adopt a centralised approach to computation but are increasingly composed of interconnected, independent computational agents (Milner, 2006). These intelligent agents cooperate to solve problems, communicate to share knowledge and expertise, and are often independently implemented and managed. A key challenge for designers of these distributed systems is how to shape the global computation of the system so that the system achieves a preferred objective.

Multiagent systems have been proposed as a system paradigm to conceptualise and tackle this new class of problems (Weiß, 1999). Systems are composed of intelligent agents that operate independently of their designer. They may be self-interested entities, capable of selecting which goals they aim to achieve and the actions required to bring about these goals. The system is no longer constructed by merging the underlying implementations of these agents into a single executable, but is characterised through the interactions that occur between these agents.

The most widely accepted property of intelligent agents is autonomy: agents act independently of any external entity, thereby exhibiting control over their own internal state (Wooldridge and Jennings, 1995). This control allows them to deliberate on their own private goal selection and achievement and to independently decide when and how to interact with other agents, while simultaneously allowing the system to adapt and change in ways that the agent designer may not have considered. The result is a systems design paradigm that is distributed, flexible and adaptive.

From a designer's perspective agent autonomy leads to a set of new challenges. Multiagent systems encourage agent autonomy and independence, yet a designer may wish to ensure some control over the system, thereby designing the system to meet their objectives even if these are not in line with the goals of the agents. These social objectives might encourage system efficiency, stability, social welfare or may simply introduce predictability. The key challenge is in designing a system that encourages predictable system behaviour while ensuring that agents are sufficiently autonomous to achieve their goals, cooperate efficiently and resolve any unforeseen conflicts.

Social norms have been proposed as a means of coordinating agents by specifying rules that govern their behaviour (Shoham and Tennenholtz, 1995). These rules are placed on all members of a society to encourage desirable actions and outcomes. A level of global predictability is introduced, since behaviour is restricted, and coordination results as agents incorporate the expected behaviour of others in their reasoning. Research in the field has been extensive: there are many norm representation languages, approaches to building agents capable of reasoning with norms and dialogue protocols for agents to propose new norms. In agent-based institutional models agents may even alter the system to propose new norms, or may choose to accept or reject proposed norms (Dignum, 1999), and it has been argued that an explicit representation of norms is essential to defining truly autonomous agents (Boella et al., 2006).

Algorithmic processes used to design social norms have largely been overlooked with norms typically assumed to be provided by the designer. In practice, hand coding social norms for large systems is not feasible, particularly in systems where agents are required to propose and adopt norms (Shoham and Tennenholtz, 1995), or in adaptive systems that change over time. An algorithmic process that automates the design of social norms is therefore a key tool for system designers and a fundamental component of autonomous agents. This thesis details an approach to synthesising norms in environments where agents utilise succinct action descriptions to construct plans to achieve their goals. Our technique utilises state and action abstractions to produce general norms that apply in many different system states, while preserving goal reachability and agent autonomy, and without requiring any additional knowledge of the goals of the agents.

To our knowledge, this work represents the first viable approach to synthesising useful norms in the absence of goal knowledge, and the first scalable approach that synthesises concise, generally applicable norms.

1.1 The Problem of Social Norm Synthesis

When agents operate in a shared environment it is possible that the actions of one agent have a direct impact on the actions of others, resulting in undesirable conflict situations (Malone and Croston, 1994). Social norms are templates for socially acceptable behaviour that, if adhered to, enable agents to coordinate their actions based purely on how they expect others to act in the system. The knowledge that other agents will obey the social norms allows the agent to behave in a way that avoids conflict.

Consider the real-world example of a human traffic network. By adhering to the norm to keep on one side of the road humans travel between destinations quickly, even in the presence of other vehicles. The social objective is one of travel efficiency and the avoidance of collisions. There is no guarantee that drivers will keep to their side, yet the incentive for shorter travel times coupled with the ramifications of a fine encourage norm-abiding behaviour.

The problem of social norm synthesis is concerned with how social norms can be designed to bring about a social objective. Consider a system designer who wishes to avoid a specific type of conflict situation in the system. The social objective here is clear, yet the rules that each agent should follow to bring about the social objective are not. A design process is followed to create the set of rules that ensure the objective is met. This forms the first challenge of the problem of norm synthesis.

If a variety of social norms bring about the social objective then norms can be ranked by the negative impact they have on the autonomy of the agents within the system. A social norm which avoids collision states but results in a large portion of desirable states being no longer reachable is less desirable than one which simply avoids the collision states. Revisiting our example, the norm to keep to one side is preferred over the norm to simply not drive, even though both avoid collisions. Designing norms that preserve goal reachability is the second challenge of norm synthesis. We now provide a summarised definition of the problem:

> *Given a social objective, the problem of automated norm synthesis is to algorithmically design a set of social norms in a multiagent system that guarantee:*
>
> 1. *to bring about the social objective, and*
> 2. *to have no impact on the achievability of agent goals.*

This problem description assumes no specific knowledge of agent goals. This allows the resulting algorithm to be applicable in adaptive, open systems, where agent goals may vary and change with time. Ensuring the achievability of agent goals results in the strictest form of the problem definition. In reality the problem may be relaxed to provide an enumeration of the goals that are no longer achievable, or to allow some reachability compromise. This relaxation is particularly useful in situations where synthesised norms conflict with an agent's internal goals.

The problem of norm synthesis was formalised by Shoham and Tennenholtz (1992a). A number of approaches have subsequently been presented yet are limited as follows:

- **Domain Specific**: Approaches to norm synthesis are tailored to specific domains reducing their general applicability (Onn and Tennenholtz, 1997).
- **Complexity**: A complete joint state enumeration of the system is often required, consuming significant resources for even small systems (Shoham and Tennenholtz, 1995; van der Hoek et al., 2007).
- **Lack of Generality**: The norms are not abstracted from the underlying systems, resulting in many state-specific norms (van der Hoek et al., 2007).
- **Goal Knowledge**: Knowledge of agent goals is required and enumerating these goals is computationally expensive or impractical (Fitoussi and Tennenholtz, 2000; Koeppen and López-Sánchez, 2010).

Abstraction is a key tool in the design and implementation of multiagent systems, often utilising high level specifications to define the individual capabilities of the agents (Vázquez-Salceda and Dignum, 2003). Abstractions over agent actions and system states simplify the design process, allowing designers to capture complex system behaviour through concise, intuitive behavioural specifications. Our approach to norm synthesis preserves these abstractions to synthesise improved social norms.

1.2 A Planning-Based Solution

In order to develop a solution to the problem of norm synthesis we require a declarative language for the specification of a multiagent system. Our approach, called *conflict-rooted synthesis*, utilises planning-based domain representations, where agent behaviour is encapsulated as operator schemata specifying what must hold for an action to be applicable, and what the effects of applying the action are. These schemata describe in what ways the world can change through the actions of agents in the system.

Utilising a planning-based approach to norm synthesis is appealing, as both are concerned with scalable search-based procedures that operate on abstract domain representations. In both fields, the approaches developed are not only theoretically shown to be sound, but are empirically shown to be useful in practice (Nau et al., 2004a). Additionally, automated planning is a mature, well respected field, providing not only a clear specification language for our domains but one that is generally accepted within the community, and one for which a host of benchmark domains exist.

1.2.1 Research Statement

The core objective of this research is to utilise declarative specifications of agent behaviour to synthesise social norms that meet a social objective. We now provide some simple state-based system semantics in order to ground our research statement. Let a social objective in such a system be the avoidance of a set of undesirable system states, perhaps those where agent actions conflict. The hypothesis of this work follows:

> *We can devise an algorithmic process that automates the synthesis of social norms given a declarative description of a planning domain and a specification of undesirable conflict states so that:*
>
> 1. *the process is more scalable than state enumeration approaches, since a complete joint state enumeration is not always necessary,*
> 2. *the social norms produced are of a higher quality since they are abstract and generally applicable, and*
> 3. *it ensures that the social norms do not prevent agents from achieving their goals.*

The term *scalable* refers to computational efficiency, not in the execution time of a single problem, but in how the execution time changes when the domain size is increased, allowing for an efficiency argument that is independent of any specific implementation.

Abstract social norms that can be applied to sets of systems states are termed *generally applicable*, and in this work are deemed to be of *higher quality* than norms that are specific to individual system states. These higher quality norms are more expressive: a single abstract norm governs behaviour in groups of system states. A result of this expressivity is that fewer norms are required to bring about the social objective. Presented with two sets of norms that ensure identical behavioural in a system, we determine the smaller set to be of higher quality.

Compare our approach to the limitations of competing methods presented above. Firstly, our approach is *domain independent* since it is applicable in any domain that can be specified using the adopted planning formalism. It is *less complex* than state enumeration based approaches since it performs a localised, abstract search of a portion of the state space. The resulting norms produced are *generally applicable* as they abstract from the underlying system, while the approach is *goal independent* as it ensures that all goals are achievable in the norm governed system.

1.2.2 Research Contributions

This thesis on the synthesis of social norms makes the following core contributions to the field of multiagent systems:

- **Norm Synthesis without Goal Knowledge**: The problem of synthesising norms in domains where the goals of the agents are unknown is a key contribution of this work. Typically, alternative norm synthesis approaches utilise a specification of the goals of the agents to check whether these goals are achievable in the restricted system. Our approach to norm synthesis allows for system designers to design norms that guarantee the reachability of agent goals without knowing these goals at design time.
- **Conflict-Rooted Synthesis**: The next contribution is the conflict-rooted synthesis algorithm, showing that it is possible to synthesise norms in domains without goal knowledge and to still provide guarantees over goal reachability. The presented technique is shown to be theoretically sound, and designed to act as the formal base upon which future work can be developed.
- **Optimisations**: A key contribution of this work is a set of theoretically sound and accurate algorithm optimisations, that increase the range of challenging domains in which the conflict-rooted synthesis approach can be applied.
- **CRS Implementation**: CRS is the main software contribution of this work. It is a default implementation of the conflict-rooted synthesis algorithm that can be adopted into future agent-based tooling, and provides a standard implementation against which competing approaches to norm synthesis can be evaluated.

More generally, this work contributes to Automated Planning and Artificial Intelligence, by illustrating a problem in which an approach based on abstract search is favourable those that enumerate and search domains at a lower level. We provide essential theory that can act as inspiration in applying ungrounded search-based techniques to new problems.

1.3 Conflict-Rooted Synthesis

This introductory overview of *conflict-rooted synthesis* avoids many of the finer technical details yet is sufficient to communicate the essence of our approach. Figure 1.1 illustrates two ways in which conflict-rooted synthesis is used in practice. System designers use conflict-rooted synthesis offline to synthesise social norms that constrain the behaviour of agents within a multiagent system, or autonomous agents utilise conflict-rooted synthesis to design new norm proposals for systems they operate within.

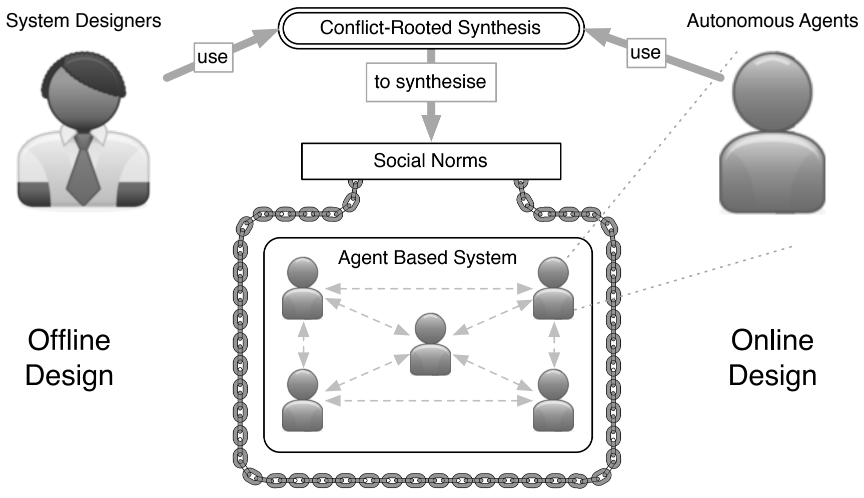


Figure 1.1: Two typical conflict-rooted synthesis use cases: offline norm design by system designers and online norm design by autonomous agents.

Consider a designer responsible for managing a shared system. This shared system is populated by a set of agents which act independently of the designer to achieve their goals by executing actions that change the state of the system. Assume now that the designer has additional knowledge which allows them to classify the states of the system as being either desirable or undesirable, and that they wish to alter the system in such a way that the undesirable states are avoided. Altering the system to avoid undesirable states may have implications on what goals each of the agents within the system can bring about. It may be the case that modifying the systems no longer allows a subset of the agents to achieve their goals and objectives.

Conflict-rooted synthesis is an approach to designing rules that can be used to avoid undesirable states. These rules guarantee that the modified system cannot enter any undesirable states, while ensuring that agents are still able to achieve their goals despite the alterations. Given the set of undesirable states, our approach creates rules that prevent agents from executing any actions that lead from a desirable state, to an undesirable state. These rules are social norms that, if abided by, ensure that undesirable states are avoided.

A key consideration to the system designer is what impact the synthesised rules will have on the society of agents. To quantify this our approach conducts a search that identifies all sequences of actions that traverse the space of undesirable states, simulating undesirable agent behaviour in order to identify what outcomes can be achieved if the system is allowed to enter these undesirable states. For each of these outcomes, an alternative sequence of actions is searched for in the modified, restricted system, in order to show that the agents are able to achieve their goals in the constrained system. If each of the agents is guaranteed to achieve their goals the process completes successfully and the designer knows that the rules can safely be incorporated into the multiagent system.

In order to develop a sound solution to this problem we make the following assumptions in our work which may limit its applicability in certain scenarios:

- Conflict situations are defined as sets of undesirable system states, which are less expressive than logic-based approaches which allow for temporal relations between states and action-based representations.
- We focus on prohibitionary social norms and state-based obligatory norms at the agent's action level. We do not synthesise norms that influence which goals an agent chooses to achieve, but rather only govern how these goals are achieved.
- Our work assumes that norms will be adhered to and is independent of incentive or sanction mechanisms used to enforce the norms in practice.

In systems where these assumptions are acceptable our approach requires fewer resources to synthesise norms of higher quality.

We conclude the overview of conflict-rooted synthesis by commenting on our use of *social norms* rather than *social laws*. Social laws, as presented by Shoham and Tennenholtz (1995), are rules defining how a system should be changed to meet a social objective and are designed in an offline fashion. We distinguish our work from this approach in two key ways:

1. Our rules are explicitly represented, allowing not only for offline synthesis by system designers but also for online synthesis by agents in the system. Related considerations, such as the quantity of rules produced, play a factor in the evaluation of this work.

2. Our approach not only synthesises rules, but also evaluates them with respect to agent autonomy and goal reachability. Agents can utilise this work to reason about whether to adopt rules proposed by other agents.

While we adopt the term social norms, we emphasise that this work does not consider many additional properties typically associated with social norms research. In particular, we do not consider any form of agent violation, nor do we present appropriate enforcement mechanisms that can be used to encourage agents to abide. These additional considerations must be satisfied when designing and implementing normative systems and agents.

1.4 Parcel Delivery Domain

We use a simple running example to illustrate the core concepts of this thesis derived from a mobilisation domain initially presented by Shoham and Tennenholtz (1995). The multiagent Parcel Delivery problem involves a set of agents whose goal it is to navigate a world in order to retrieve and deliver parcels. The world is defined as a graph structure, with nodes depicting locations in the world and edges representing paths that agents can follow to traverse between locations. Parcels appear at random locations in the world, and it is the goal of each of the agents to retrieve a parcel from its source location and to deliver it to a target location. We do not preclude the existence of any other goals. Agents have three core capabilities: `move` between connected locations, `pickup` parcels from their current location and `drop` parcels into their current location. We use the following notation throughout this thesis. We write $\mathtt{a_i}$ to represent Agent i, $\mathtt{node_j}$ to represent Location j and $\mathtt{parcel_k}$ to represent Parcel k.

We commonly wish agents within the Parcel Delivery domain to not concurrently occupy the same location. This thesis is concerned with the follow key questions:

1. How do we represent this social objective?

2. Can we automatically design rules to enforce this objective?
3. What effects do these rules have on the agents in the system?

1.5 Thesis Overview

Our approach to the problem of norm synthesis is heavily influenced by research in multiagent systems and automated planning. We devote the next two chapters to presenting the relevant required background literature in each of these fields. Chapter 2 focuses on social norms, beginning with an overview of social norms as a coordination mechanism before detailing approaches to norm synthesis. Of the alternative methods introduced, an approach based on model checking is presented in detail. Chapter 3 follows with required background knowledge from automated planning. Specifically, we detail two planning domain formalisms of varying expressiveness and the associated language used to express these input domains. We conclude our background on automated planning with a discussion of related work in the field.

The background overview is followed by two theoretical chapters. Chapter 4 theoretically details the core norm synthesis approach, providing a high level intuitive introduction before delving into the formalism-specific details. Chapter 5 outlines a set of optimisations that improve the performance of the core synthesis approach without jeopardising the results produced.

With the theoretical details complete, we next present details of the empirical evaluation performed. We separate details of the implementation in Chapter 6 from the actual empirical results generated in Chapter 7. Our discussion of the results with respect to the research statement of this work is presented in Chapter 8 before concluding.

Chapter 2

Background: Coordination using Social Norms

A multiagent system is comprised of a set of agents acting within a shared environment in order to achieve some private or shared goals. The actions of agents are interdependent since the actions of one agent may have a positive, or negative effect, on the outcomes of another. Coordinating agent action is a fundamental concern when designing agent-based systems, as it allows the community to behave in a coherent manner that avoids undesirable interactions. A range of solutions to the coordination problem exist, with the following extremes (Moses and Tennenholtz, 1995):

- Coordination is imposed through *centralised* control where the actions of all the agents are dictated by a central entity. Such approaches suffer from a single point of failure, and are inhibited by the complexity of managing the actions of all agents. Furthermore, fully prescribing agent behaviour may be contrary to the objectives of the multiagent system, particularly where agents are self-interested as centralised control severely limits agent autonomy.
- Coordination is *decentralised*, where agents are responsible for identifying conflict situations and managing these as they arise. This places additional onus on the agents requiring specific machinery to identify conflicts and to negotiate solutions to these conflicts. While such approaches are robust and adaptive, they require agents to communicate to resolve conflicts.

Typically coordination approaches in multiagent systems are decentralised and implemented at runtime (Wooldridge, 2002). We discuss some common approaches to coordination at a very high level before detailing coordination via social norms.

2.1 Coordination Approaches in Multiagent Systems

We begin by discussing how agents might coordinate through the sharing of local plan information. *Partial Global Planning* (PGP) (Durfee and Lesser, 1991) and subsequently *Generalised Partial Global Planning* (GPGP) (Decker and Lesser, 1995), are approaches based on this principle. Agents looking to cooperate construct local plans in order to achieve their own private goals and communicate details of these plans to construct a partial view of the global system. Once all agents have agreed to the plan they may execute their individual components in a coordinated manner.

A number of PGP related approaches have been proposed in the literature (Decker and Li, 2000; Clement and Barrett, 2003; Wagner et al., 1998), yet all are designed to work in domains where agents are assumed cooperative. Furthermore, since coordination is communication-based, resources are consumed each time coordination is required. In systems where agents interact frequently this may be overly expensive.

A second approach to coordinating behaviour in a multiagent system is through the development of models of mental state. Cohen and Levesque (1991) introduced their notion of *teamwork* based on the study of human coordination where team members have individual intentions to achieve their own persistent goals, but that they also share a joint intention to bring about a shared goal. Agents commit to the shared goal, and act responsibly towards other team members when the goal is not achievable. Jennings (1993) extended this joint intention theory by defining *commitments* as pledges to other agents and *conventions* as a means of monitoring commitments so as to act responsibly towards others.

Other approaches to coordination, including techniques for opponent modelling and persuasion techniques based on game theoretic concepts, share limitations with those presented above. Repeatedly coordinating behaviour is expensive: to coordinate agent behaviour via PGP requires agents to continually create and refine global plans. In situations where goals change rapidly, or where the system itself is variable, this repeated computation is not desirable.

2.2 Coordination using Social Norms

One established coordination theory inspired by social science and legal theory is that of *social norms* (Lewis, 1969) where socially responsible behaviour is specified for each agent to follow. Social norms provide an intermediate approach bridging the gap between centralised and decentralised coordination mechanisms. Consider that prior to interacting within a system agents established a convention that governs their interactions so as to avoid conflict. The purpose of agreeing on such a convention beforehand is that it defines what behaviour agents *should* exhibit in states that might lead to conflict. Should all agents abide by the convention then conflicts are minimised and the efficiency of the system increased. Such a convention is an example of a social norm (Jennings, 1993), allowing agents to act independently while still achieving coordination. Here, coordination hinges on a common expectation of agent behaviour in the system, allowing agents to coordinate their behaviour without the need for explicit communication. Social norms are fundamental to the design of multiagent systems, and key for all activities that require coordinated participation of all agents (López y López and Luck, 2003).

2.3 Social Norms

Even though norms have been widely discussed in social theory and legal theory, we mainly focus our presentation on the application of social norms to multiagent systems research. While earlier discussion of social norms exist, Ullmann-Margalit (1979) provided the following definition:

> *A social norm is a prescribed guide for conduct or action that is complied with by the members of society.*

While this statement has been taken from the social sciences, it is clear that there is a strong relation to the field of multiagent systems. Agents operating within a system are in a social setting, and it is this prescription of behaviour that results in coordination within the system. This normative expectation of behaviour regulates the system since agents are following a common code of conduct. Systems in which social norms regulate behaviour of a society of agents are termed *normative multiagent systems* (Boella et al., 2006).

In these systems norms commonly define the behaviour that is *obligatory* and *prohibited*. Obligatory norms specify actions and states of the world that agents are expected to bring about, while prohibitory norms prohibit access to states or the execution of actions. Additional notions such as permissions, rights and power in normative systems can be included as normative concepts when modelling more complex systems, such as institutions.

If we supplement the standard notion of a multiagent system with concepts of social norms, then we should also specify the meaning of *autonomy* in such a system.

The most common reference regarding agent autonomy is provided by Wooldridge and Jennings (1995), where agents must be capable of proactive, independent action, and control over their internal state. A more subtle definition of agent autonomy is provided by Verhagen (2000), where the degree of agent autonomy is the independence the agent shares from any external entity, allowing for a more subtle classification of agents by their autonomy. Intelligent agents are commonly *goal autonomous* since they are able to synthesise their own goals, and to perform the means-end reasoning required to achieve these goals. However, in normative multiagent systems social norms have an effect on what is achievable: it might be the case that certain goals are prohibited. Agents that are unable to reason about the effects of social norms on their practical reasoning cycle have limited autonomy in such systems. As such, *norm autonomous* agents are capable of incorporating norms into their deliberation and means-end reasoning cycles, thereby requiring an explicit representation for norms. We now detail this, and other key properties of social norms:

- **Explicit Representation**: Social norms are explicitly represented and communicated (Boella et al., 2006). It has been argued that while implicit constraints on the behaviour of agents do introduce high reliability, they give the agents no possibility to reconsider the norms of the system, or to adopt new norms (Castelfranchi et al., 2000) while unforeseen circumstances might make the implicit hard-wired norms obsolete (Conte et al., 1999a). An explicit representation of norms allows agents to reason about the normative position of other agents, particularly in institutions where agents reason about which norms to adopt at runtime (Cortés, 2004). In sociological terms we are interested in *formal* social norms, rather than informal social standards.

- **Persistent**: Norms are designed to be valid for the long term, and are not restricted to the current behaviour of agents within a system, but rather all future behaviour. Norms differ from contracts or short term agreements in this sense.

- **Generality**: A social norm is a general rule designed to govern multiple related undesirable situations within a system (Grossi et al., 2007), thereby regulating a range of situations. This is most common in human law theory, where laws are termed “open textured” and abstract over the specifics of each situation that they govern. Social norms in a multiagent system are no exception: norms must be general to regulate a wide range of situations over time, and to reduce the number of norms that govern the system (Grossi and Dignum, 2005).

- **Incentives for Compliance**: In systems where compliance is not guaranteed agents are encouraged to act in accordance with the norms through the use of incentives or sanctions (Fornara and Colombetti, 2006). These mechanisms play a central role in an agents reasoning on whether to adopt or comply with a proposed social norm (Tuomela and Bonnevier-Tuomela, 1995).

2.3.1 Ensuring Compliance in Normative Systems

Moses and Tennenholtz (1995) presented the first computational model of normative multiagent systems called *artificial social systems* where they observed that through the imposition of social patterns of behaviour they were able to improve system efficiency by avoiding conflict. In this system social norms are represented as *social laws* and are system-wide restrictions on agent behaviour. We provide further details on this work in Section 2.4.1, but for now we note that these social laws are hard constraints imposed by the designer on all agents within the system. Systems where agents have no choice but to perform according to social norms, either through altering the mental states of agents, or through enforced action execution, are said to be *regimented* (Grossi and Dignum, 2005).

Regimented systems ensure norm compliance with no room for deviation yet Grossi et al. (2007) argue that regimented norms are simply details of the implementation of the system. Since agents have no capacity to identify regimented norms, let alone deviate from them, social norms are indistinguishable from other properties of the system. Furthermore, Castelfranchi et al. (2000) argues that an agent's ability to deviate from a norm is essential to the fundamental function of the society, especially in adaptive, changing systems where agents may need to violate norms in order to continue to function. Kollingbaum and Norman (2003) state that regimented systems place a strong onus on the system designer to ensure correct, conflict-free restrictions since conflicting norms prescribing contradictory actions for a single state result in agents not being able to perform any action, for fear of violating one or the other norm. Finally, Dignum (1999) argues against the practical implications of regimentation by noting that the system designer must alter the agent implementation whenever the set of norms changes. Human systems are an example of normative systems where regimentation is not possible. In these systems agents comply with social norms due to: authority of power (Axelrod, 1986; Jones and Sergot, 1993); rational appeal and incentive (Savarimuthu and Purvis, 2007); emotions or social pressure (Elster, 1989) and follow-the-crowd behaviour (Epstein, 2001).

A number of incentive mechanisms have been proposed in the literature. Ågotnes et al. (2007) present a game theoretic approach that utilises knowledge of the goals of agents in order to ensure that norm compliance is the rational choice, while Boella and van der Torre (2005) present a mechanism to penalise violating agents by adjusting their utilities. In situations where goal knowledge is not available, or where it is not possible to interest all agents in the social objective, an alternative means of ensuring compliance is required.

One option is to introduce sanctions or penalties for deviating behaviour. Axelrod (1986) argues for this approach based on its existence in human systems, however he emphasises the difficulty in implementing such systems, requiring advanced violation detection and penalty mechanisms. Fornara and Colombetti (2006) proposed a mechanism that allows sanctions to be applied by having agents play particular roles to themselves impose the sanction. Similar agent-imposed sanction mechanisms have been proposed where the system designer alters agent utilities through sanctions and reward (Boella and van der Torre, 2005), and the use of trust and reputation mechanisms to avoid interaction with norm violators (Grizard et al., 2006; Walker and Wooldridge, 1995). Examples of these strategies include:

- *utility sanctions*: if norm violation is detected, the offending agent is fined by a monitoring agent,
- *trust reduction*: a system policing agent that detects a norm violation might reduce the system-wide trust and reputation of the violating agent,
- *reciprocation*: if a norm is violated by an offending agent, the interaction partner can reciprocate by violating obligations with the offending agent,
- *ostracism/blacklisting*: if a norm is violated by an agent during an interaction, the interaction partner excludes the offending agent from all further interactions.

Since agents are able to violate norms fewer restrictions are placed upon the autonomy of the agents. These systems are more flexible and dynamic, allowing agents to pursue unforeseen goals and allowing agent action even in the event of norm conflict. In practice however, a mixture of regimentation and enforcement is required to ensure that violation will not go unpunished through further violation (Grossi et al., 2007). The severity of sanctions forms an integral part of the design of social norms. Excelente-Toledo et al. (2001) propose a framework to reason about coordination mechanisms. In this framework, a classification of sanctions is proposed related to the cooperative task that must be performed, however no indication is given of a process to automatically compute which class of sanction is best for which system or situation.

2.3.2 Benefits of Social Norms

We have highlighted how norms are central to achieving pre-planning coordination in multiagent systems and now present a few key benefits of this coordination approach:

- social norms require *less communication* since norms govern not only the immediate behaviour of agents, but are persistent and long term,
- social norms provide a *balanced* approach respecting both agent autonomy and social conformity,
- social norms are *flexible* allowing for offline design or online norm creation, and
- social norms can *reduce the reasoning requirements* of agents by restricting the behaviour of agents to that which is deemed socially acceptable.

Social norms are central to the implementation of electronic institutions (Vázquez-Salceda, 2003). We detail formal representations of social norms next and subsequently institutional models of normative multiagent systems.

2.3.3 Formally Representing Social Norms

Initial attempts to formally represent social norms were presented in legal theory where the specification of rules and laws in natural language led to ambiguity in interpretation. The creation of *deontic logic*, a branch of symbolic logic used to define notions of obligation, permission and prohibition, and to specify relationships between them was developed (von Wright, 1951). In von Wright's first system, obligations and permissions were treated as features of acts, yet it was subsequently refined and respecified as a normal modal logic, leading to the generally accepted *standard deontic logic*.

Standard deontic logic is an extension of propositional logic, with sentences of the form $O(a)$ representing propositions that *a ought to be the case*. Here O represents the modal necessity operator. From this we define the basic deontic logic KD, composed of propositional logic axioms together with the formulae:

$$\begin{aligned} K: &\quad \big[O(p) \wedge O(q)\big] \rightarrow O(p \wedge q) \\ D: &\quad O(p) \rightarrow \neg O(\neg p) \end{aligned}$$

From these axioms we introduce the modal operators P and F representing propositions that are permitted to hold, or forbidden to hold respectively. Interestingly, standard deontic logic allows us to rephrase both of these operators in terms of O:

$$P(p) \equiv \neg O(\neg p)$$
$$F(p) \equiv \neg P(p)$$

Deontic logic allows for an expressive formal characterisation of social norms, provides a coherent mechanism for identifying inconsistencies between norms and has been extensively adopted in the multiagent systems community (Dignum, 1999; Boella et al., 2006). Yet it is devoid of operational semantics and contains little information to guide the designer toward an implementation of the norms (Cortés, 2004).

Numerous other approaches have been proposed to model norms. Meyer (1988) reduces deontic logic to a variant of dynamic logic modelling the semantics of obligation, permissions and prohibitions relative to agent action. Here, the necessity modal operator is $[a]$, and the expression $[a]\phi$ equates to the conditional *if action a is performed, ϕ will hold afterwards*. Again, O, F, and P are phrased in terms of the modal operator. Importantly, this allows for the conditional specification of norms:

$$[a]O(p)$$

with the implication that once a is performed it is obligatory to bring about p. As we will see when we introduce the implementation details of norms it becomes clear that conditions for norm activation are key in designing a normative system.

2.3.4 Implementing Normative Multiagent Systems

Regulating the behaviour of agents is increasingly problematic in larger systems, especially in open systems where agents are able to enter and leave the system at will. Jones and Sergot (1993) argue that regimenting open systems is not practical due to these concerns. *Electronic institutions* are structured, open multiagent systems where heterogenous agents are grouped together based on their capabilities within the system. These descriptions of capabilities are referred to as *roles* and allow the specification of large-scale systems independent of the agents in the system.

Social norms define what behaviour is expected by agents performing a particular role. Vázquez-Salceda and Dignum (2003) identify two key approaches to implementing norms in multiagent systems:

1. **Agent perspective**: The designer is concerned with how social norms affect the agent's reasoning cycle.
2. **System perspective**: The designer is interested in implementing norm mechanisms that result in the system meeting a social objective.

Numerous approaches have been proposed to incorporate norms into agent reasoning and deliberation cycles, and we mention a subset here. López y López and Arenas Marquez (2004) proposed an architecture for normative BDI agents where agents decide whether norms should be adopted, choose whether to comply with adopted norms, and update their internal goals to reflect their choice of adopted norms. Similar work on incorporating norms into the BDI cycle was detailed by Dignum et al. (2002), where BDI was extended to incorporate explicit distinct representations for norms, desires and goals. Here the focus is on capturing the interactions that social norms have on the internal goals and desires of BDI agents. A second approach to incorporate norms into the BDI cycle was presented by Meneguzzi and Luck (2009), where the BDI action language is modified at runtime to incorporate newly adopted norms. A non-BDI approach, the NoA framework presented by Kollingbaum (2005), simplifies these approaches by defining obligations as the key motivation for actions as opposed to goals. Reasoning about norm adoption in this system is akin to deliberation in BDI approaches resulting in increased emphasis on ensuring norms are consistent and non-conflicting.

Research on norm implementations from the system perspective occurs largely in the context of institutions, although some work has been presented independently. Aldewereld et al. (2007) presented implementation mechanisms to ensure norm compliance and enforcement, where norm abstractions are a key property this work shares with Vázquez-Salceda (2003). Social norms utilise two distinct representations: *declarative* level norms are more abstract and generally applicable, and are divorced from finer implementation details, while *operational* level norms are mapped into rules and procedures that are invoked at runtime. Declarative norms do not contain concrete means for their implementation. This distinction between social norm levels was initially detailed by Conte et al. (1999b), and is a characteristic of many institution formalisms. What is required is a specification of norms that is both abstract and generally applicable in the system, yet has clear operational semantics for the agents involved.

2.3.4.1 Models of Electronic Institutions

Our investigation of multiagent institutional models and related research answers the following questions:

- What are the key properties of norms in institutions?
- How are norms created in these models?
- Are there declarative and operational representations, and an automated means of mapping between them?

Noriega (1997) proposed a formalisation of electronic institutions, based on the analysis of a fish market auction house. A range of subsequent models adopt Noriega's notion of *roles* and merge them with social norms to define role-based behaviour leading to a number of different approaches to institutional modelling.

López y López and Luck (2002, 2003) argue that in order for agents to reason about norms a general model of norms is required. Their aim was to formulate a specification that included all the essential components of social norms to help agents decide what to do at runtime. This work extended the SMART agent framework, specifying norms in the Z specification language with a key focus being the construction of a comprehensive model of social norms. The key properties that describe a norm in this model are:

- a set of *normative goals* describing behaviour that must be achieved or avoided,
- agent sets composed of *addressees* and *beneficiaries* of the norm,
- state descriptions defining the *context* under which the norm is active, and the *exceptions* in which it is not, and
- social incentives in the form of *reward* and *punishment* goals.

Importantly, this work only defines a model of norms: the question of how norms are synthesised is not investigated. Additionally, norms include no operational semantics, focusing rather on norms at the declarative level (López y López et al., 2006).

Esteva et al. (2002) also investigated how institutions and norms can be formally specified and modelled resulting in an operational model of norms, complete with a means to verify and visualise models. The resulting language, coupled with a development tool, was called ISLANDER. The institutional model ISLANDER adopts is based on modelling dialogues between agents, and allows agents to identify at what point a current dialogue is and whether new agents can join the dialogue. ISLANDER specifications are focused on realising agent-based institutions, and are therefore operational in nature. Norms are specified as obligations of the form:

$$Obl(x, \psi, s)$$

signifying that agent x is obliged to perform illocution ψ in dialogue s. *Normative rules* are a set of conditions on the current state of the system and define which obligations are currently active, and are required when defining norms in the system. Since ISLANDER specifications of norms contain no abstract representation of expected behaviour they are procedural in nature. Furthermore, norms are not synthesised automatically in this system, but are specified by the designer. An extension of ISLANDER by Vázquez-Salceda et al. (2004) approached the issues of norm implementation. In their extension they include temporal notions into the condition of a norm yet do not tackle the problem of norm synthesis.

In his work on HARMON*IA*, Vázquez-Salceda (2003) defines a unifying framework connecting the declarative, logic-based formalisation of norms to operational semantics. The framework is used to model the highly regulated medical domain of organ and tissue allocations for transplantations. Norms in this domain are prescribed at an abstract level that is independent of the implementation of the institution itself. For example, norms governing organ allocation are common across medical institutions or authorities. HARMON*IA* contains rules that govern how abstract norms are translated into concrete instantiations with operational semantics. While HARMON*IA* is a rich institutional model, it is designed to model an existing real-world organisation. Norms are not automatically synthesised, but simply specified as part of the modelling process, and a static mapping is used to translate norms from the declarative to operational form. We summarise our overview of norms in institutions below:

- Institutional models focus on the specification of social norms, and not on the *synthesis* of these norms.
- Institutions specify norms at different levels, from very abstract declarative to concrete operational representations.
- The mapping between levels is either non-existent, or is statically defined.
- Norms are commonly conditional on the state of the system.

2.4 Synthesis of Social Norms

Synthesising social norms is concerned with designing norms that achieve a social objective. Existing norm synthesis approaches can be classified into three categories:

1. *Emergent* social norms come about through repeated agent interactions.
2. *Online* social norms are synthesised at runtime, allowing agents to create new social norms that implement their social objectives, and are a key component of norm autonomous agents.
3. *Offline* social norms are created and subsequently incorporated into a multiagent system, and are suited to systems where agent behaviour is regimented.

Shoham and Tennenholtz (1992a) provided the first formal model of social norm emergence, where agents select a particular action *strategy* from a predefined set and act accordingly while monitoring the performance of their actions via a number of metrics. The authors identify under which conditions agents switch to a common strategy, with the common choice of behaviour termed a *social convention*. Kittock (1993) extends this preliminary work by identify what impact the structure of the system has on the emergence of norms, and Walker and Wooldridge (1995) investigated mechanisms to monitor convention evolution. The graph-based system representations were later extended by Delgado (2002) to include more complex structures. Griffiths and Luck (2010) study norm emergence with self-interested agents in tag-based cooperation systems. The authors analyse the effect that changes to the system and agents have on group formation and effectiveness, particularly in scenarios where agents are permitted to deviate from the norm.

Work on the emergence of norms predominantly deals with social norms as implicit changes in behaviour according to some environmental payoff as opposed to explicit changes in order to achieve a social objective. There is a strong correlation with mechanism design approaches to ensuring coordination yet there is no guarantee that these approaches can always be applied since they rely on making the norm compliant behaviour the rational choice. For example, Boella and van der Torre (2007) showed that emergent norms cannot be guaranteed to emerge at all, if they do they are difficult to alter, and that emergent norms are not practical when modelling behaviour of roles in institutions since no explicit representation of social norms exist.

Unlike emergent norms, approaches to online synthesis utilise explicit norm representations defined in institutions. Norms are not static constraints but contain properties that model the current state of the norms in the system. In their extension of the SMART agent framework, López y López et al. (2006) presented a specification of norms whose *state* is altered at runtime, where norms can be active, issued, fulfilled or violated. Similar approaches introduce norms when an event occurs. For example, Dignum (1999) describes a dynamic logic approach where new norms are activated when existing ones are fulfilled.

Related research on the offline design of social norms is most relevant to our work. The seminal *social law model* coupled with its associated complexity results are integral to this work and are presented in detail. We then present other approaches at a higher level, before analysing a second approach based on model checking.

2.4.1 The Social Law Model

Shoham and Tennenholtz (1992b, 1995) presented *the social law model* to investigate the complexity of designing behavioural constraints in an offline fashion. A *social law* is defined as a hard constraint on the behaviour of agents in a multiagent system: from a social norms perspective, social laws are norms that are designed in an offline fashion to regiment the behaviour of agents. Social laws are more restrictive than the wider definition of social norms yet the complexity of designing social laws is key when considering the design of norms.

In the social law model, given a set of states S, a set of actions A and some first order language $\mathcal{L}$, a *constraint* is said to be a restriction on an action of the form $\langle a, \varphi \rangle$ where $a \in A$ and $\varphi \in \mathcal{L}$. A *social law* is then defined as a set of such constraints $\langle a_i, \varphi_i \rangle$ with at most one for each action $a_i \in A$: should the current state satisfy the precondition φ_i, then the action a_i is forbidden. A *social agent* is a tuple $\langle S, \mathcal{L}, A, SL, T \rangle$ where:

- S, $\mathcal{L}$, A are defined as above,
- SL is a set of social laws, and
- $T : S \times A \times SL \to 2^S$ is a total transition function where for every $s \in S$, $a \in A$, $sl \in SL$, if $S \models \varphi$ and $(a, \varphi) \in sl$ then $T(s, a, sl) = \emptyset$. If the social law applies in the current state then the transition representing the prohibited action is disallowed.

Goal knowledge is incorporated into the model via *focal states* $F \subseteq S$. An agent must always be able to transition from one focal state to another, irrespective of what any other agent does. The social law model aims to restrict non-essential state transitions that an agent can make, thereby simplifying deliberation and planning without restricting agents from any goal states.

2.4.1.1 Complexity of the Social Law Model

Shoham and Tennenholtz show that the problem of finding laws that guarantee access between focal states of the system is NP-complete. In order to understand the complexity proof, and its implications, it is useful to introduce the three clause Boolean satisfiability problem (3-SAT). Given a set of Boolean clauses in conjunctive normal form, the Boolean satisfiability problem is concerned with finding a truth assignment to the Boolean literals such that every clause is simultaneously satisfied, and the entire expression is true. This general problem, and the restricted 3-SAT problem, is NP-complete (Cook, 1971).

The social law computational problem is shown to be intractable through a reduction from 3-SAT. Initially, it is shown that any set of prohibitions can be encoded in polynomial space. Next consider a reduction from a 3-SAT conjunctive normal form Boolean expression to the social law model. Let each clause in the expression represent the transition between two sequential states. The set of all possible actions within the system in each state can be represented as (c, l) where c represents the clause in the expression, and l the relevant literal within the clause. Therefore, given k clauses there are at most $3k$ actions. Two actions (c, l) and (c', l') are said to conflict if:

- either $l = \neg l'$ or $l' = \neg l$, or
- if $c = c'$ and $l \neq l'$.

It is now clear to see that a satisfiability solution will limit each agent to a single action per state, where no two actions will conflict with each other. In the worst case the problem is intractable, yet in general much depends on the state of the system, the complexity of the environment, the number of focal states and other factors. For the offline design of constraints the time taken to compute and verify the constraints is less inhibiting since the state of the system is not directly affected, whereas in online design scenarios the computation is more constrained through resource limitations. It is extremely advantageous to provide a partial solution in these situations. Finally, the introduction of domain dependent heuristics in the synthesis process could result in efficient synthesis procedures for particular subdomains of the general problem.

Shoham and Tennenholtz argue that even though the problem of effective norm synthesis is intractable in the worst case, there are situations where norms can be effectively synthesised in practice. In the case where computation time is exceeded, they note that partial results would be desirable, as they could still be used to guide the agent toward conflict-free social behaviour.

2.4.2 Alternative Synthesis Approaches

The social law model was extended by Onn and Tennenholtz (1997) where synthesis is considered in *robot mobilisation* domains. A robot network is defined as a graph $G = (V, E)$, a set of R robots and the length of each edge $(u; v) \in E$ is specified by a length function. Agents wish to visit particular nodes in the graph while avoiding collisions on graph nodes or edges. Social norms are a set of disallowed graph edges as well as velocity and direction of movement restrictions along each allowed edge and ensure goal reachability and collision avoidance. The underlying graph topology follows a set structure. Norms are efficiently synthesised using a set of network traffic laws and a vertex labelling mechanism defined as *graph routing*. While efficient, the algorithm is specific to domains that can be modelled using the same topologies, and the approach is not generally applicable and does not allow for any different specifications of conflict.

Fitoussi and Tennenholtz (2000) present the synthesis of *minimal* and *simple* social laws as as an extension of artificial social systems. This approach is not concerned with norm synthesis but rather presents a technique for choosing between alternative existing social laws. Two choice criteria are introduced. A social law is *minimal* if no other social law exists that is less restrictive than it, granting agents more freedom to choose their behaviour while ensuring conformity. A social norm is *simple* if it is only dependent on sensing information that agents are able to retrieve. To this end, simple social laws are laws which apply to a variety of agents with a range of sensors, as well as to agents without these sensors. The authors proceed to study these two criteria in their automated guided vehicle framework. No synthesis procedure is defined in this work: it is concerned with norm refinement rather than norm synthesis.

An interesting, alternative approach to norm synthesis is that presented by Koeppen and López-Sánchez (2010). Here, norms are introduced during the execution of the system by a regulatory body, where new norms are learnt from previous experience. Case-base reasoning is employed to construct these norms based on the outcomes of prior interactions, with the new norms designed to govern similar, future interactions. The authors evaluate their proposal in a road traffic scenario. Interestingly, this learning approach ensures that similar social situations are governed by similar norms, thereby providing some level of generality and predictability to the results. However, the entire process is based on feedback from an executing system: conflict situations must be encountered a number of times before norms are synthesised.

van der Hoek et al. (2007) show that Alternating-time Temporal Logic (ATL) can be used to express and analyse social laws. Norm *effectiveness* (does a given set of norms implement a social objective?), *feasibility* (does a set of norms exist that implements a social objective?) and *synthesis* (what norms implement a social objective?) are all framed as an ATL model checking problem. The process of synthesising and analysing social laws in this framework is compelling: the expressiveness of the action-based systems provides an effective framework with which to study and understand social laws. One key disadvantage to this approach is the lack of state abstractions which results in semantic structures that are effectively enumerated joint state transition systems. Since this approach is used for comparison throughout this thesis we now present it in significantly more detail.

2.5 Social Laws in Alternating Time

We begin by clearly stating that this approach solves a fundamentally different problem to that which is solved by our work, and as such our empirical evaluation is not a true head-to-head comparison. There are two key differences:

1. Our work assumes the set of desirable, focal states to be all conflict-free states as opposed to a smaller subset of the conflict-free states.
2. Our approach sacrifices expressivity in the representation of conflict for scalability benefits.

We compare our work with this approach as it is the most relevant, general approach that also has an algorithmic instantiation. Our aim is only to show our approach to be superior at solving the more specific norm synthesis problem outlined in this thesis.

An overview of branching time logics is essential in order to differentiate this model checking approach from our technique. We present an overview of computation tree logic followed by the more expressive alternating time logic. Readers familiar with the concepts of model checking and temporal logic can forward to Section 2.5.2.

2.5.1 Branching Time Logics and CTL

Model checkers verify that a set of properties hold in a succinct representation of the possible states of a system, called a model, where temporal logics are commonly used to express the properties to be checked. We are interested in logics that utilise a *branching time model*, where different possible futures are represented as branches of a computation tree. We discuss two branching time temporal logics: Computation Tree Logic (CTL) and the more expressive Alternating-time Temporal Logic (ATL).

We begin with the syntax of CTL. Consider $\phi = \{p, q, r, \dots\}$ to be a set of atomic propositions. All valid CTL expressions can be generated from the following grammar:

$$
\begin{aligned}
&\varphi ::= p \text{ where } p \in \phi \\
&\varphi ::= \top \mid \neg\varphi \mid \varphi \wedge \varphi \\
&\varphi ::= \mathsf{AX}\varphi \mid \mathsf{EX}\varphi \mid \mathsf{E}(\varphi\,\mathcal{U}\varphi) \mid \mathsf{A}(\varphi\,\mathcal{U}\varphi).
\end{aligned}
$$

CTL combines path quantifier operators with temporal operators. There are two path quantifiers: A universally quantifies over all possible paths that originate from the current state, while E existentially quantifies over at least one. The operators X, $\mathcal{U}$, F and G are path specific temporal operators: $\mathsf{X}\varphi$ indicates that φ holds in the next state of the path; $\varphi\,\mathcal{U}\psi$ indicates that φ holds in the current and all future states on the path, up until some point after which ψ holds; $\mathsf{F}\varphi$ indicates that φ holds in some future state on the path, and $\mathsf{G}\varphi$ states that φ holds on every state in the path.

The semantics of CTL expressions are modelled using *Kripke structures*. A Kripke structure K is a directed graph representation of possible worlds of a system. We write S to represent the set of possible system states, and $I \subseteq S$ to represent the initial states of the system. A transition relation $R \subseteq S \times S$ captures the possible transitions of the system between states. If an atomic proposition p holds in a state $s \in S$ for model K then we write $K, s \models p$. It is common to denote an *interpretation* function for states, where each state is associated with a set of properties. We define a path over a sequence of states as $\pi = s_0, s_1 \dots$ where $\pi[i]$ refers to the i'th state in π. Finally, $\Pi(s)$ is the set of all possible paths originating from s. The semantics of CTL in terms of a Kripke structure K and an initial state $s \in I$ are:

$$
\begin{aligned}
&K, s \models \top \\
&K, s \models \neg\varphi \text{ iff } K, s \not\models \varphi \\
&K, s \models \varphi \vee \psi \text{ iff } K, s \models \varphi \text{ or } K, s \models \psi \\
&K, s \models \mathsf{AX}\varphi \text{ iff } \forall\pi \in \Pi(s) : K, \pi[1] \models \varphi \\
&K, s \models \mathsf{EX}\varphi \text{ iff } \exists\pi \in \Pi(s) : K, \pi[1] \models \varphi \\
&K, s \models \mathsf{A}(\varphi\,\mathcal{U}\psi) \text{ iff } \forall\pi \in \Pi(s), \exists u \in \mathbb{N} \text{ s.t. } K, \pi[u] \models \psi \text{ and } \forall 0 \leq v \leq u : K, \pi[v] \models \varphi \\
&K, s \models \mathsf{E}(\varphi\,\mathcal{U}\psi) \text{ iff } \exists\pi \in \Pi(s), \exists u \in \mathbb{N} \text{ s.t. } K, \pi[u] \models \psi \text{ and } \forall 0 \leq v \leq u : K, \pi[v] \models \varphi
\end{aligned}
$$

From these core semantics we define the following:

$$
\begin{aligned}
\mathsf{AF}\varphi &\equiv \mathsf{A}(\top\,\mathcal{U}\varphi) & \qquad \mathsf{EF}\varphi &\equiv \mathsf{E}(\top\,\mathcal{U}\varphi) \\
\mathsf{AG}\varphi &\equiv \neg\mathsf{EF}\neg\varphi & \qquad \mathsf{EG}\varphi &\equiv \neg\mathsf{AF}\neg\varphi
\end{aligned}
$$

A Kripke structure models a multiagent system, with states representing joint system states and transitions action invoked changes in the world. We utilise model checking to verify that properties hold in the resulting system by checking the model of the system. It is useful at this point to discuss the interpretation of the path quantifier operators in such a model. A CTL expression such as $\mathsf{EF}\varphi$ states that some path exists where φ eventually holds. In order to bring about this path we require all agents to execute particular actions: if one agent were to execute different actions there is no guarantee that φ will come about. Conversely, $\mathsf{AF}\varphi$ states that φ will come about independent of what actions the agents perform. The path quantifiers in CTL allow us to reason about what is achievable through total cooperation, or the lack of cooperation, yet does not allow reasoning about the effects of partial cooperation between agents.

2.5.1.1 Alternating-time Temporal Logic

Alternating-time Temporal Logic (ATL) was developed by Alur et al. (2002) as a generalisation of CTL used to represent and reason about coalitions in multiagent systems. CTL allows us to verify properties regarding complete cooperation and no agent cooperation, yet not about computations that can be brought about by a subset of the agents. ATL replaces both CTL path quantifiers with a *cooperation modality*, $\langle\langle C\rangle\rangle$, where C represents the set of cooperating agents. We write $\langle\langle\rangle\rangle$ to represent situations where no agents cooperate. ATL is strictly more expressive than CTL. Consider the following ATL statements, where A_g represents the set of all agents:

- $\langle\langle A_g\rangle\rangle\mathsf{G}\varphi$ is equivalent to $\mathsf{EG}\varphi$ in CTL. The grand coalition can bring about any computation that always satisfies φ.
- $\langle\langle\rangle\rangle\mathsf{G}\varphi$ is equivalent to $\mathsf{AG}\varphi$ in CTL. Since no coalition exists every computation is a viable future path.

Consider the set of possible primitive propositions to be ϕ. The following grammar can be used to construct all valid ATL expressions:

$$\begin{aligned}
&\varphi ::= p \text{ where } p \in \phi\\
&\varphi ::= \neg\varphi | \varphi \vee \varphi\\
&\varphi ::= \langle\langle C\rangle\rangle\mathsf{G}\varphi | \langle\langle C\rangle\rangle\mathsf{F}\varphi | \langle\langle C\rangle\rangle\varphi\mathcal{U}\varphi | \langle\langle C\rangle\rangle\mathsf{X}\varphi.
\end{aligned}$$

The temporal modalities follow from CTL. Note that in these ATL expressions all temporal modalities must be preceded by a coalition modality in the same way as path quantifiers precede temporal modalities in CTL.

The semantic structures underpinning ATL are *Alternating Transition Systems*. van der Hoek et al. (2007) use a semantic model with an alternative representation, called *Action-based Alternating Transition Systems* (AATS), a semantically equivalent structure to alternating transition systems but which provides a clearer separation of actions from their associated preconditions, a desirable property when considering ATL as an approach to norm synthesis. Fundamentally, these systems are state automata similar to the semantic model for CTL presented above. We refrain from a complete presentation of AATS here but emphasise that the models are simply enumerations of the joint state space, and that their formal representation is not required in this work.

2.5.2 Synthesising Norms using Model Checking

We present the synthesis of social norms via ATL model checking as proposed in (van der Hoek et al., 2007). Given an AATS depicting an underlying multi-agent system and an initial start state, we are interested in identifying computation paths that abide by a given social objective. By expressing the permanent adherence to the social objective as a requirement we are able to compute whether social norms are required to enforce the social objective, and what these norms might be.

Let Ac_{Ag} be the set of all agent actions, and 2^Q be the set of all possible model states. The prohibition function $\beta : Ac_{Ag} \to 2^Q$ defines what agent actions are prohibited in any state. As such, the application of β can be seen as a restriction of the model since transitions that were previously possible are no longer allowed. We write the norm synthesis model checking approach as a function:

$$Synth : S, s_0, \varphi \mapsto \beta$$

comprising:

- the AATS *model* S representing the multiagent system,
- an *initial start state* s_0 from which the computations are generated,
- a *social objective* φ detailing the properties required in the normative system, and
- a resulting set of prohibitionary norms β, that if applied to S result in a system that ensures the social objective is always satisfied.

Given β, we create a restricted model S' simply by removing the prohibited transitions from the transition relation in S.

2.5.2.1 Norm Effectiveness, Feasibility and Synthesis in ATL

We begin by investigating how we can use a model checker to determine whether a given prohibition function β is *effective* at ensuring a social objective in a model S, such that the resulting modified system $S' = S\backslash\beta$ satisfies the following ATL expression:

$$S', s_0 \models \langle\langle\rangle\rangle \mathsf{G}\varphi.$$

The empty coalition modality above acts as a universal quantifier over computations originating from s_0, stating that for all computations originating from s_0, φ must hold in every state of each computation. In other words, no matter what strategies are adopted by agents in the system, every resulting state will satisfy φ. This ensures that the social objective always holds, and that the prohibitions β are *effective*.

Next we discuss norm *feasibility*: given a model S, starting state s_0, and social objective function φ, a social norm β exists if the following ATL expression holds:

$$S, s_0 \models \langle\langle Ag\rangle\rangle \mathsf{G}\varphi.$$

If the agents cooperate and agree to a particular strategy profile, and if the resulting computation always satisfies φ, then a social norm exists. Notice that the grand coalition above is essentially an existential path quantifier.

Synthesising a prohibition function is identical to model checking feasibility. If the feasibility expression holds then at least one computation exists where the social objective is always satisfied. The *positive witness* to the feasibility checking is such a computation which, if adhered to, will ensure the social objective. The prohibition function β can be constructed from the positive witness simply by prohibiting all behaviour that does not correspond to that prescribed by the resulting computation.

2.5.2.2 Maintaining Reachability through Focal States

Synthesising norms using a model checker identifies a single computation that ensures the social objective and regiments agent behaviour accordingly, effectively producing a master plan for all agents in the system. The prohibition function β is therefore likely to prohibit access not only to states that violate the social objective, but also states that are simply not part of the prescribed behaviour, as illustrated in the next example.

Example Consider a single agent Grid World example. Our agent can always perform one of two actions: to `move` between adjacent locations, and to remain `idle`. Assume our agent wishes to traverse a simple three location topology:

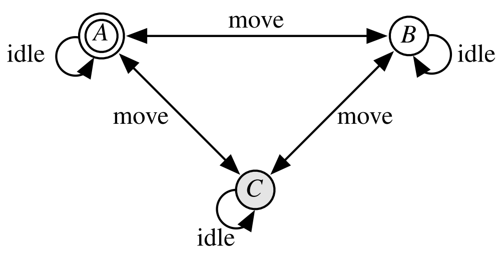


Figure 2.1: A three-state Grid World topology, with initial state A and conflict state C

Our agent begins in state A and we wish to have it avoid location C. A number of infinite computations satisfy the social objective, the simplest of which has the agent not leaving A by continually remaining `idle`. The computation satisfies the social objective, however it also affects the reachability of states that do not violate the objective: state B cannot be reached even though a computation that oscillates between A and B would not violate the objective. $\square$

van der Hoek et al. (2007) incorporate focal states into the social objective in order to ensure that useful norms are found. Recall that a set of focal states $\Sigma_F \subseteq \Sigma$ represents the states where an agent must be able to traverse from any focal state to any other. The authors incorporate this objective into their definition of a social objective by assigning a unique proposition s_i for every focal state $s_i \in \Sigma_F$. The resulting reachability objective can then be expressed as follows:

$$\bigwedge_{s_i \in \Sigma_F} \left[s_i \to \bigwedge_{s' \in \Sigma_F} \langle\!\langle Ag \rangle\!\rangle \mathsf{F} s' \right].$$

Intuitively, the above expression states that if a given state satisfies the proposition s_i, then it is focal, and a set of computations must exist whereby any other focal state is reachable. The reachability objective is merged with the social objective φ to form the new ATL objective:

$$S, s_0 \models \langle\!\langle Ag \rangle\!\rangle \mathsf{G} \Big(\varphi \wedge \bigwedge_{s_i \in \Sigma_F} \left[s_i \to \bigwedge_{s' \in \Sigma_F} \langle\!\langle Ag \rangle\!\rangle \mathsf{F} s' \right] \Big). \tag{2.1}$$

2.6 Summary of Social Norms in Multiagent Systems

Social norms provide an attractive means of coordinating agent behaviour that define what is expected of agents, and allow agents to reason about the behaviour of others. This form of coordination requires no runtime communication, producing persistent and long-term restrictions that differ from fixed-term contracts or agreements.

The ability to synthesise norms is a key capability of norm autonomous agents and a useful tool for system designers. While a considerable amount of work exists that formalises norms, particularly in institutional settings, much of this work requires the norms to be specified by hand. A number of alternative synthesis approaches have been presented, but many are domain specific or require particular knowledge of the goals of the agents. The most compelling alternative presented by van der Hoek et al. (2007), where norm synthesis is formalised as a model checking problem. This allows for great expressivity, but has a number of drawbacks:

- The propositional nature of the problem results in many, state-specific norms.
- Performing a complete joint-state enumeration to create the model of the system is computationally expensive, and requires significant resources.
- Encoding focal states into the model checking problem requires knowledge of what these focal states are.

Our planning-based approach attempts to address these core concerns. In the following chapter we present our planning background before progressing to our approach.

Chapter 3

Background: Automated Planning

Automated planning is concerned with synthesising a sequence of actions in a problem domain to achieve a goal. Through a process of deliberation agents select a goal to achieve and invoke a planner to perform the means-end reasoning to achieve it. Planning theory provides us with languages to describe agent action using succinct declarative representations which we use to formalise the domains in which we synthesise norms. By aligning our algorithm with planning we are able to harness not only domain representation languages but also a range of related tools and techniques.

A key property of planning-based state and action representations is that they abstract away from the underlying state-based system, allowing for more succinct representations. Planning techniques search these more compact domain representations for solutions to the planning problem, and our approach to norm synthesise adopts similar abstract search techniques. In this work we consider two languages of differing expressivity: a *Propositional* representation and a more expressive first order *Classical* representation.

Our domains are assumed to be fully observable, deterministic *classical planning* environments with time modelled discretely. We do not consider more advanced planning notions such as partially observable environments, or environments with non-deterministic outcomes.

3.1 State Transition Systems

We define planning semantics in state transition systems modelled as directed graphs composed of nodes and connecting edges. We assume a finite set of discrete and instantaneous environment states representing the nodes in our transition system where each state represents one unique configuration of the system. We write s to represent a *state* and Σ to represent the set of all possible states such that $s \in \Sigma$.

We model agent action as a transition between states. Let A be the set of all actions available to agents within the system. The transition effect of an action is a deterministic binary relation $\theta : \Sigma \times A \to \Sigma$. Executing some action in a current state of the system results in a transition to a new state. This transition system is fully grounded as no reference is made to transitions from, or to, sets or groups of states, and there is no uncertainty modelled. We assume agent actions to be asynchronous. We define a *state transition system* as the tuple $\langle \Sigma, A, \theta \rangle$.

3.2 Set Theoretic Representation

The domain formalisms that follow are based on the General Propositional Planning Formalism, an extension of traditional STRIPS that allows incomplete state specifications as described by Nebel (2000). We begin by defining a formal state space representation and follow it with representations of action.

3.2.1 Propositional State Space Representation

Let Σ be a finite set of propositional atoms. We use these atoms to uniquely identify states within our system. We present the set $\hat{\Sigma}$ consisting of all atoms in Σ, their negations, as well as $\top$ and $\bot$ denoting truth and falsity respectively:

$$\hat{\Sigma} = \{p, \neg p | p \in \Sigma\} \cup \{\bot, \top\}.$$

Given the set of atoms Σ we define the language of propositional logic over Σ as $\mathcal{L}_\Sigma$, using the following grammar:

$$\varphi ::= \varphi \wedge \varphi | \varphi \vee \varphi | \neg \varphi | \langle symbol \rangle$$

where $\langle symbol \rangle \in \Sigma$. Let L be an arbitrary set of literals where $L \subseteq \hat{\Sigma}$. We define $\neg L$ to be the *element-wise negation* of L such that:

$$\neg L = \{p | \neg p \in L\} \cup \{\neg p | p \in L\}.$$

Furthermore, we write *set difference* between two sets of literals L_1 and L_2 as $L_1 \backslash L_2$. With these definitions in place we can now present the theory of our state representation. Recall that the propositional atoms, and more specifically the literals over these atoms, are used to uniquely identify states within the system. A state s is defined as a complete truth assignment for every atom in Σ: states are described by the propositional atoms that hold in their description. We adopt a closed world assumption to define any unlisted propositions as being false.

Example Consider the set of atoms $\Sigma = \{a, b, c\}$. The state $s = \{b\}$ represents the unique truth assignment where b is true and a and c are false. □

3.2.2 State Abstraction through Specification

A *state specification* describes a set of states by the truth assignments that they have in common. Each specification is composed of a subset of the literals in $\hat{\Sigma}$. We define a state specification S to be:

- **Consistent** if $\perp \notin S$ and $\nexists l \in \hat{\Sigma}$ such that $l \in S$ and $\neg l \in S$. That is, a specification is consistent if there are no complementary literals in S.
- **Complete** if for every atom $a \in \Sigma$, either $a \in S$ or $\neg a \in S$. That is, a literal exists in S for every atom in Σ.

A complete state specification references a single state, while an empty specification represents all states. We write $s \models S$ to denote that the state s models state specification S, implying that the atoms in s satisfy the literals in S. The set of all states that model a particular specification S is written as $Mod(S)$.

Example Let $\Sigma = \{a, b, c\}$. A valid state description is again $s_1 = \{b\}$ where b holds. Now consider a state specification S over Σ where $S = \{\neg a, b\}$. s_1 and S are subtly different. The specification S represents all states where a does not hold and b holds. Our closed world assumption does not hold here: we represent states where c holds and does not hold. Therefore, S represents the states $\{b, c\}$ and $\{b\}$. □

3.2.3 Propositional Operator Representation

Operators are action schemata that specify transitions between specifications of states. Rather than defining transitions between states, we define schemata that model when actions can be performed, and how the system changes once these actions are executed. Operators are tuples of the form:

$$o = \langle name, pre, post \rangle$$

where:

- *name* is the *name* that uniquely identifies this operator schema,
- *pre* is a set of consistent literals representing the *preconditions* required for this operator to be applicable, where $pre \subseteq \hat{\Sigma}$, and
- *post* is a set of consistent literals representing the *effects* that this operator has once applied, where $post \subseteq \hat{\Sigma}$.

We write $name(o)$, $pre(o)$ and $post(o)$ to refer to the first, second and third elements of the tuple respectively and O to represent the set of all operator schemata. Operators intuitively represent transitions between states in the system where an operator o can be performed in state s if $s \models pre(o)$. The effects describe what manipulations will occur to s once o is performed to produce the successor state. We present the semantics of operators in two stages. First we describe how operators represent state transitions in a grounded transition system, and follow this with the semantics associated with applying operators to abstract state specifications.

Example Consider the Parcel Delivery domain with agents able to `move`, `pickup` and `drop` parcels. Let $\texttt{at}(\texttt{a}_1,\texttt{node}_1)$ be the literal denoting that $\texttt{a}_1$ is located at $\texttt{node}_1$ and `conn` represent the fact that two locations are connected. We model the agent's ability to `move` to $\texttt{node}_2$ using the operator:

$$\begin{aligned}
\text{OPERATOR:}\quad & \texttt{move}(\texttt{a}_1,\texttt{node}_1,\texttt{node}_2)\\
\text{PRE:}\quad & \{\texttt{at}(\texttt{a}_1,\texttt{node}_1),\texttt{conn}(\texttt{node}_1,\texttt{node}_2)\}\\
\text{POST:}\quad & \{\neg\texttt{at}(\texttt{a}_1,\texttt{node}_1),\texttt{at}(\texttt{a}_1,\texttt{node}_2)\}
\end{aligned}$$

□

3.2.3.1 Transition Semantics

We define the semantics of an operator in a state transition system as the set of transitions between states. We write $pos(L)$ to be the *positive literals* in L and $neg(L)$ to be the *negative literals*. Given a state $s \in \Sigma$ and operator $o \in O$ we can define the transition function $\theta : \Sigma \times O \to \Sigma$ as follows:

$$\theta(s,o) = \begin{cases} s \setminus \neg neg(post(o)) \cup pos(post(o)) & \text{if } s \models pre(o), \text{ and} \\ & post(o) \not\models \bot. \end{cases}$$

If the precondition of the operator o is satisfied in state s and the effects of o are consistent, then a transition occurs from s to a new state where all atoms in the negative effects are removed from s and atoms in positive literals are added. We have assumed that if the conditions do not hold that no transition exists.

Now consider a state specification S and operator schema o. We define the transition between state specifications as a function $R : 2^{\hat{\Sigma}} \times O \rightarrow 2^{\hat{\Sigma}}$ as follows:

$$R(S,o) = \begin{cases} (S \setminus \neg post(o)) \cup post(o) & \text{if } S \not\models \bot \text{, and} \\ & S \models pre(o)\text{, and} \\ & post(o) \not\models \bot \end{cases}$$

If the preconditions of o are satisfied by S and the effects of o are consistent, then a transition exists to a new state specification where all negated effect literals are removed, and the positive literals added.

Let a transition exist between two state specifications S_1 and S_2 such that $S_2 = R(S_1,o)$. For every state represented by S_1, a state transition exists to some state represented by S_2. That is, $\forall s_1 \in Mod(S_1)$, $\exists s_2 \in Mod(S_2)$ where $s_2 = \theta(s_1,o)$.

3.2.4 Defining the Planning Problem

We now formally define the automated planning problem using propositional state and operator representations. A planning problem is a tuple:

$$\Pi = \langle \Xi, S_I, S_G \rangle$$

where

- $\Xi = \langle \Sigma, O \rangle$ is the declarative *domain structure* consisting of the finite set of propositional atoms Σ and a set of operators O,
- $S_I \subseteq \hat{\Sigma}$ is the *initial state specification*, and
- $S_G \subseteq \hat{\Sigma}$ is the *goal state specification.*

A solution to the planning problem is a *plan*. If we consider the set of all possible, finite, operator sequences to be O^* then a plan Δ is an element of this set, $\Delta \in O^*$. We write $\Delta = \langle o_1, o_2, \ldots \rangle$ to represent the sequence of operators that compose a plan, where $\forall i \, . \, o_i \in O$. We write $\langle \rangle$ to represent the empty plan.

We have now defined what a plan is, but have not specified under what conditions a plan represents a solution to a given planning problem. To this end we must specify the result of a applying a plan Δ to a given state specification S. We adopt a recursive definition of the result function $Res : 2^{\hat{\Sigma}} \times O^* \rightarrow 2^{\hat{\Sigma}}$ defined as follows:

$$\begin{aligned} Res(S, \langle \rangle) &= S, \\ Res(S, \langle o_1, o_2 ... o_n \rangle) &= Res(R(S,o_1), \langle o_2, ..., o_n \rangle). \end{aligned}$$

Let $S' = Res(S_I, \Delta)$ be the state specification reached through the application of a plan Δ to the initial state specification of problem Π. We say that Δ is a *solution* to Π iff:

1. $S' \models S_G$, and
2. S' is consistent.

3.2.5 Extensions to Propositional Planning

A family of propositional planning formalisms has emerged in the planning literature all based on extensions of STRIPS (Fikes and Nilsson, 1971). The most restrictive formalism is the propositional variant of STRIPS which requires complete state specifications, unconditional effects and propositional atoms as formulae in the operator preconditions. However, less restrictive formalisms exist that relax these restrictions (Nebel, 2000). We detail the following subset of extensions for subsequent discussion:

- **Incomplete state specifications** ($\mathcal{S}_I$): solution plans are valid for all states represented by the incomplete state specifications in the planning problem.
- **Conditional effects** ($\mathcal{S}_C$): operator effects are conditional on the current state where effects with satisfied conditions are applied.
- **Literals as preconditions** ($\mathcal{S}_L$): we allow literals in the set of preconditions for operator schemata.

In Section 3.3 we present a formal theory $\mathcal{S}_{IL}$ that combines $\mathcal{S}_I$ and $\mathcal{S}_L$.

3.2.5.1 Complexity and Expressivity

The complexity of planning using each of the different propositional planning extensions is computationally equivalent (Nebel, 2000). We adopt the notion of *expressive power* to illustrate how concise a representation is in a particular formalism. A formalism is more expressive than another if the space required to represent the planning problem is less. We illustrate the expressivity relationships between planning formalisms in Figure 3.1. Additionally, we highlight two classes of formalism in grey, in which the cross compilation of representations between formalisms does not lead to an increase in any solution plans in the domain.

Given these complexity and expressivity relations we present the following related points:

- The complexity of planning is equivalent for all the presented formalisms: there is no theoretical benefit to using a more expressive representation.

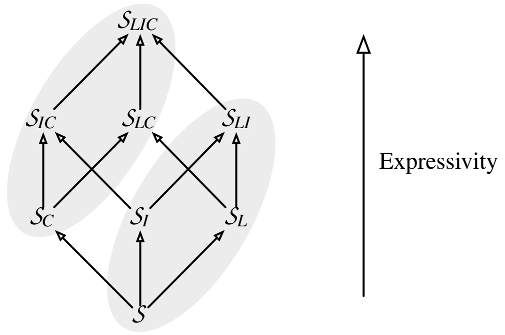


Figure 3.1: Expressivity Relations Between Planning Formalisms

- It is possible to compile conditional effects out of a representation, but this leads to polynomial growth in the size of the plan.
- When moving up in Figure 3.1 there exists a polynomial-time compilation scheme that preserves plan size exactly.

This completes our presentation of the propositional planning formalism. We present the classical extension next.

3.3 Classical Planning Representation

In this section we present a classical planning formalism that greatly increases the expressivity over the propositional formalism presented above by utilising state and operator abstractions through parameterised predicates. We detail the classical representation by extending our propositional notions of states and specifications, and introduce parameterised operator schemata.

Example A propositional state representation of an agent's location in the Parcel Delivery domain requires an atom to describe every location of the agent. If the agent is at location `A` then we might write `agentAt_A`. A unique operator is required for each possible movement between locations, such as `move_A_B`. The resulting domain representation is verbose, even for the simplest problems. The classical representation alleviates this as agent locations can be represented using a single variable predicate `agentAt`($\mathtt{x_1}$). Similarly, the `move` action is parameterised defining the conditions required and resulting effects in relation to variable start and end locations. □

3.3.1 States and State Specifications

Our classical representation is based on a restricted form of the language of first-order logic. We define the language $\mathcal{L}$ as:

$$\mathcal{L} = \mathcal{L}_p \cup \mathcal{L}_c \cup \mathcal{L}_v$$

composed of finitely many predicate symbols $\mathcal{L}_p$, constant terms $\mathcal{L}_c$ and variable terms $\mathcal{L}_v$. A *constant symbol* represents an object or element in the domain, such as $\texttt{node}_1$ or $\texttt{parcel}_2$. Constant terms are analogous to propositions in the propositional formalism, and are lowercase alpha-numeric character sequences. A *variable symbol* is an element that may represent an arbitrary constant symbol. Variable symbols begin with uppercase letters, such as `Agent` and `Parcel`. For this presentation we assume the names of the variables to be arbitrary, and that the set $\mathcal{L}_v$ contains all of these variations. A *predicate symbol* defines a relation between a set of parameters. Predicates are composed of the conjunction of the predicate symbol and the set of parameters (either variable symbols or constant symbols) that the relation applies between. For example, the predicate $\texttt{agentAt}(\texttt{a}_1,\texttt{node}_2)$ defines a relationship between the symbols $\texttt{a}_1$ and $\texttt{node}_2$ denoting the location of the agent.

The above language allows us to define predicates of an arbitrary structure. We let $\mathcal{A}$ be the set of *atoms* composed of predicates with variable and constant terms. For the purposes of this work it is useful to distinguish between predicates that contain variable parameter symbols. We call an atom:

- *ground* if it contains no variable symbols, such as $\texttt{agentAt}(\texttt{a}_1,\texttt{node}_3)$, and
- *unground* if it contains at least one, such as $\texttt{agentAt}(\texttt{Agent},\texttt{node}_3)$.

We use $\overline{bar}$ notation to represent sets containing only ground elements. It follows that $\overline{\mathcal{A}} \subseteq \mathcal{A}$: the set of atoms includes all ground atoms. The set of literals $L_{\mathcal{A}}$ follows:

$$L_{\mathcal{A}} = \{a|a \in \mathcal{A}\} \cup \{\neg a|a \in \mathcal{A}\} \cup \top \cup \bot.$$

We write $L_{\overline{\mathcal{A}}}$ to represent the set of *ground literals*. The definitions of set difference and element-wise negation follow as in Section 3.2.1.

A *state* is a subset of the possible ground atoms in $\overline{\mathcal{A}}$. As before, states follow a closed world semantics. Our definition of a state specification S follows where $S \subseteq L_{\overline{\mathcal{A}}}$.

The set of symbols that hold in every state are said to be *non-fluent*, while those that are true in a subset of the possible states are said to be *fluent*. If an agent might perform an action that alters the truth assignment of any atom, then this atom is fluent.

3.3.2 Parameterised Operators

An operator o is a parameterised action schema triple of the form $o = \langle name, pre, post\rangle$ where:

- *name* is of the form $o(X_1, \ldots, X_n)$ specifying an operator name o and set of n variable symbols, $X_1 \ldots X_n$, used in the rest of the operator definition,
- *pre* is the preconditions for the operator, a consistent (possibly ungrounded) subset of the literals formed by $L_{\mathcal{A}}$, and
- *post* is the effects for the operator, a consistent (possibly ungrounded) subset of the literals formed by $L_{\mathcal{A}}$.

We write $name(o)$, $pre(o)$ and $post(o)$ to represent the first, second and third elements in the tuple. A ground instance of an operator, termed an *action*, is obtained by substituting constant symbols for all variables in the operator schemata. From an operator we can derive a finite number of possible instantiated actions through substitutions.

3.3.3 Substitutions

Given a mapping from variable to constant symbols it is possible to *instantiate* the abstract operator schemata to ground action schemata by substituting constant symbols for each operator parameter. We define a *substitution* set σ as a set of mappings from variable to constant symbols:

$$\sigma \subseteq \{ (v \leftarrow c) \mid v \in \mathcal{L}_v \, , \, c \in \mathcal{L}_c \}.$$

Applying a substitution set to a set of atoms $\mathcal{X} \subseteq \mathcal{A}$ is denoted $\sigma[\mathcal{X}]$. A *grounding substitution* for an operator o is written $\overline{\sigma}$ and is a substitution that maps all operator parameters to constant symbols. The application of a grounding substitution to an operator o results in a ground instance a of the operator, where $\overline{\sigma}[o] = \overline{o} = a$.

Example Let operator o be $\texttt{move}(\texttt{Agent}, \texttt{From}, \texttt{To})$ and $\sigma_1 = \{(\texttt{Agent} \leftarrow \texttt{a}_1), (\texttt{From} \leftarrow \texttt{node}_1)\}$. Substitution results in the operator $\sigma_1[o] = \texttt{move}(\texttt{a}_1, \texttt{node}_1, \texttt{To})$. Note that this is still ungrounded, as no constant binding exists for the variable $\texttt{To}$. The substitution set $\overline{\sigma_2} = \sigma_1 \cup \{(\texttt{To} \leftarrow \texttt{node}_2)\}$ is a grounding substitution, since $\sigma_2[o] = \texttt{move}(\texttt{a}_1, \texttt{node}_1, \texttt{node}_2)$ is ground. □

3.3.4 Applying Parameterised Operators

Since the initial and goal specifications for the classical planning representation are always ground, and since we only consider the application of actions as opposed to unground operators, the ground nature of state specifications is always maintained, resulting in solution plans that are always ground. The semantic mapping from resulting plans to state and action sequences in the state transition system remains identical. One of the key attributes of the conflict-rooted synthesis method is that it does not depend on a ground state specification or any particular initial state. As a result, an extension of this core classical planning representation is required to allow the application of operators to specifications containing unground atoms. This extension is detailed in Section 4.3.2.

3.4 Planning Domain Definition Language

The *Planning Domain Definition Language* (PDDL) is a standardised encoding language for planning problems proposed by McDermott (2000). Originally proposed as a means of standardising planner input for the International Planning Competition, PDDL has evolved and increased in complexity along with the domains considered in the competition. We provide an introduction to the language and define key concepts and terminology that we utilise later. While an important component of the implementation of conflict-rooted synthesis, it should be stated that any classical planning language could have been used. In our case we incorporated PDDL for three reasons:

1. It contains a superset of the encoding we required for conflict-rooted synthesis.
2. It is independent of any particular planning implementation.
3. It is the language of choice for planning benchmark domains.

A PDDL planning task is an encoding of a single planning problem. We utilise a subset of PDDL composed of the following components:

- **Objects**: typed atoms present in the domain. Specifying a `postman` object of type `agent` in the Parcel Domain is written as:

  ```
  postman - agent
  ```

- **Predicates**: the set of predicate definitions in the domain, each of which includes the predicate name, and typed parameters. A predicate that encodes an agent's location is:

  ```
  (agentAt ?person - agent ?place - location)
  ```

  where `agentAt` is the predicate name, `?person` is a variable of type `agent` and `?place` is a variable of type `location`.

- **Operators**: A PDDL Action Definition is an operator schema composed of a name, the operator's parameters, the preconditions and effects. We assume the preconditions and effects to be STRIPS-style conjuncted lists of literal predicates. The `move` operator which moves an `agent` from location `?l1` to `?l2` is:

```
(:action move
     :parameters (?agent - agent ?l1 - location ?l2 - location)
     :precondition (and
                         (conn ?l1 ?l2)
                         (agentAt ?agent ?l1)
                    )
     :effect (and
                    (not (agentAt ?agent ?l1))
                    (agentAt ?agent ?l2)
             )
)
```

- **Initial State**: A set of ground atoms that are true in the initial state. If the `postman` agent starts in the location `depot` the following predicate is included:

```
(agentAt postman depot).
```

- **Goal Specification**: The goal state is an incomplete state specification. If the `postman` requires to get to location `delivery-point` then the goal is:

```
(agentAt postman delivery-point).
```

PDDL adopts a separation of the planning problem into two input files: a *domain file* that contains the predicates and action definitions, and a *problem file* for objects, initial state and goal specifications. While this is purely an implementation detail it does afford us the benefit of being able to divorce the domain structure from the problem specific details, allowing the domain structure to be reused by multiple different problems. Reasoning about domains purely on the domain structure leads to problem-independent results since all processing is independent of any particular problem instance. This is a recurring theme in our work as we present a subtle balance between maximising generality through domain structure reasoning, and ensuring applicability through the incorporation of problem specific knowledge when required. Our approach differs from many standard planning techniques which immediately merge domain and problem instance information prior to solving the planning problem, and we later argue that a more strategic approach to grounding and merging can lead to improved results for certain problems.

3.4.1 ADL Extensions

The subset of PDDL presented above contains the essential information required to understand our implementation of conflict-rooted synthesis. We do however provide some additional details on the more expressive language ADL (Pednault, 1987) which allows for disjunctions and quantifiers in preconditions and goals, and effects that can be conditional and themselves contain quantifiers. In this section we illustrate the use of universal quantification in operator preconditions, as it is utilised again in Section 4.3.6.1.

ADL allows for arbitrary variables in the preconditions, where each variable is quantified either universally or existentially. The result is an action definition language that is more expressive, allowing for actions to be conditional on elements in the domain that are not explicitly passed into the action definition. ADL uses the syntax `forall` and `exists` to represent the quantification. The scope of quantification is contained within the condition formula that follows. For example, the precondition of the `move` operator defined above might be extended as follows:

```
...
(not (exists (?agent2 - agent) (agentAt ?agent2 ?l2)))
...
```

stipulating that the `move` operator may only be applied if no agent is occupying location `?l2`. In situations where action definitions are altered and additional variables are introduced, these variables must be quantified appropriately in the action schema.

3.5 Automated Planning Related Work

The planning work presented here does not aim to solve the problem of automated norm synthesis, yet we reference this work since there are commonalities in the approaches adopted. We begin with approaches to incorporating control knowledge into the planning process, followed by approaches to generalised planning. We reference graphical problem representations and finally detail the Fast-Forward Planning System utilised in our evaluation.

3.5.1 Control Rules in Planning

We begin by presenting alternative approaches to incorporating designer domain-specific knowledge into the planning process. The PRODIGY system, presented by Veloso (1994), is based around planning and was one of the first approaches to use control knowledge in the form of control rules. These rules govern operator selection, allowing for the rejection and preference of particular operators. Although PRODIGY allows for these rules to be explicitly provided, it focuses on learning them at runtime, incorporating feedback from plan execution into subsequent planning steps to improve its efficiency. Control rules in PRODIGY are algorithmic in nature, akin to those found in expert systems. For example, the rule to always prefer the more specific operator (OP1) to a more general alternative (OP2) is represented as:

```
(CONTROL-RULE PREFER-SPECIFIC-OPERATOR
          (if   (and    (candidate-operator OP1)
                        (candidate-operator OP2)
                        (is-ancestor-of OP1 OP2)))
          (then prefer operator OP1 OP2))
```

Similarly to PRODIGY, UCPOP (Barret et al., 1995) utilises algorithmic, rule-based control knowledge, and was subsequently extended by Estlin and Mooney (1996) to include the learning of control rules in the DOLPHIN framework. For the purposes of our discussion the structure of control rules can be assumed identical to PRODIGY's.

Bacchus and Kabanza (2000) incorporated formalised control rules into TLPLAN, adopting a declarative representation for control knowledge. Rules are specified in Linear Temporal Logic, allowing the designer to use time modalities to govern future states of the system. For example, the LTL expression

$$\forall \mathtt{a_i}, \mathtt{a_j}, \mathtt{n_k} \,.\, \Box\neg\big(\mathtt{agentAt}(\mathtt{a_i},\mathtt{n_k}) \wedge \mathtt{agentAt}(\mathtt{a_j},\mathtt{n_k})\big)$$

ensures that agents never collide in the Parcel Delivery domain. TLPLAN extends PRODIGY by allowing control rules that are not only restrictions on the current state of the planner, but also on any previous states encountered. A main contribution of this work is the theory of progressions, which allows the planner to efficiently track the state of control rules without having to revaluate them on each planning iteration. TLPLAN differs from model checking as it performs no verification but simply controls the underlying plan search.

Inspired by TLPLAN, Kvarnström and Doherty (2001) presented TALplanner as a similar forward search planner with a declarative language for control rules. In TALplanner a control rule is a Temporal Action Logic expression of the form:

$$[\mathtt{t}]\mathtt{parcelAt}(\mathtt{p_1},\mathtt{n_1}) \wedge \mathtt{goal}\big(\mathtt{parcelAt}(\mathtt{p_1},\mathtt{n_1})\big) \rightarrow [\mathtt{t}+1]\mathtt{parcelAt}(\mathtt{p_1},\mathtt{n_1})$$

stating that, at some time *t* if a parcel exists at its location, and this forms part of the goal of the agent, then the parcel should remain at the location in the next time step. This rule therefore prevents agents from moving delivered parcels. Importantly, control rules in both TLPLAN and TALplanner reference not only the goal state, but the current and preceding states as well. Additionally, numerous approaches, such as those by Cresswell and Coddington (2004) and Edelkamp (2006), have investigated how logic-based control rules can be encoded as part of the goal specification.

In their work on domain control knowledge, Baier et al. (2007, 2008) use action-centric procedural domain knowledge to represent a template of actions in a solution plan. This GOLOG-based procedural domain language includes common programming language constructs, allowing the designer to specify what actions should be used at various stages of the plan. For example, the procedure:

if $\mathtt{agentAt}(\mathtt{a_1},\mathtt{Node}) \wedge \mathtt{parcelAt}(\mathtt{Parcel},\mathtt{Node})$
then $\mathtt{pickup}(\mathtt{a_1},\mathtt{Parcel},\mathtt{Node})$

dictates that if Agent $\mathtt{a_1}$ is in a location where a parcel exists, then the `pickup` action should be invoked to pick up the parcel.

There are benefits to incorporating control rules into the planning process. Bacchus and Kabanza (2000) showed that, with good control rules, TLPLAN is able to solve certain classical planning problems orders of magnitude more quickly. A downside of these approaches when applied to norm synthesis is that no explicit rules governing behaviour are produced as an output. Instead, the control rules are evaluated at each step of the planning process to ensure that a resulting plan satisfies the objectives.

3.5.2 Generalised Planning

A *sequential plan* is composed of a sequence of actions that, if executed, is guaranteed to transition a system from an initial state to a goal state. These guarantees are possible if the dynamics of the system are fully known: the effects of actions are deterministic, the system adapts in easily predicted ways, etc. In many domains there is insufficient knowledge with which to synthesise such a plan and *conditional planners* are used to synthesise tree-like plans that include sensing actions. At execution time an agent performs a sensing action and selects the branch of the plan corresponding to their result. Conditional planning solves a more general problem than sequential planning.

Now consider the problem of finding a plan for not only a single problem instance, but rather an entire class of problems. In the Parcel Delivery domain this equates to delivering all the parcels in a domain, irrespective of how many exist in the problem instance. The resulting plan is termed a *generalised plan*, since given a problem domain the generalised plan can be used to instantiate a sequential plan that solves the problem instance with the added benefit of being able to invoke the same generalised plan to solve different problems instances as well. Providing guarantees on correctness and termination on iterative plans is far more complex than sequential plans since it must be shown to solve all possible problem instances. As such, the majority of early work on generalised planning involves theorem proving where plans are viewed as programs and planning the task of program synthesis (Stephan and Biundo, 1996).

Plans with loops are termed *iterative plans*. In order to alleviate the difficulty of producing iterative plans Levesque (2005) proposed that the guarantees on these plans be reduced, and presented KPLANNER. His approach is two step: a generalised plan, based on a variation of the programming language $\mathcal{R}$ is first produced that solves only a small subset of problem instances, and once found this plan is verified against a larger subset. By decoupling synthesis and verification Levesque showed benefits over existing approaches, at the loss of general guarantees of correctness. Srivastava et al. (2010) proposes limiting the class of loops involved in a plan, and shows that under these constraints plans can be shown to be correct and applicable.

Work on generalised planning also focuses on learning a generalised plan from sequential plans. Srivastava et al. (2008) utilise a three-value logical state representation to identify when sequential states are sufficiently similar to represent an expanded loop. This three-valued logic (Sagiv et al., 1999) allows for the succinct representation of sets of states by representing propositions about the state with 1 if they are *present*, 0 if they are *not present*, and $\frac{1}{2}$ if *possibly present*. This is a very similar to state specifications in the planning formalism, where propositions are specified if they hold, negated and specified if they do not, and not specified if possibly present.

Srivastava et al. (2008, 2010) utilise this state abstraction technique to summarise state representations into more abstract sets of states that are then incorporated into the learnt generalised plan. As with our work, reasoning about actions applicable in abstract state representations requires that assumptions be made regarding *possibly present* predicates. The authors introduce *focus* and *coerce* operations to this end. While similarities exist with our process of state specification *refinement*, there are distinct differences too. The authors use state abstraction techniques to reduce their plan representation, while our traversal process begins with the most abstract representation and *refines* down to the more specific. Finally, the authors focus mainly on merging sequential plans into a generalised plan structure, rather than actually synthesising plans with loops.

3.5.3 Graphical Problem Representations

In Section 6.1.2 we present *traversal graphs* as succinct representations of sets of plans that are *similar* in their sequence of actions: they may start in the same specifications, or deviate only in the last operator. A variety of alternative graphical representations exist in the literature, yet these approaches differ in the following ways:

- Traversal graphs include the representation of unground predicates.
- They use state specifications to represent sets of states rather than individual literals or propositions.
- Initial or goal state information is not required to construct them and every represented plan is sound.

While an important contribution of this work, traversal graphs are simply a data structure with which it is convenient to implement our approach. Our theoretical presentation is appropriately divorced from this implementation detail, allowing us to use the more intuitive notion of *runs* when describing our approach. Traversal graphs draw inspiration from two existing approaches: Problem Space Graphs and Planning Graphs, and we present these below for completeness.

Etzioni (1993) introduced *Problem Space Graphs* (PSG)s as a graph-based representation of a problem space. A problem space is represented as a set of disjoint graphs, each rooted with an achievable literal of the problem. Nodes are either *operator* or *literal* nodes, where edges are *conjunctive* or *disjunctive*. The root literal node is connected, using disjunctive edges, to the operator nodes that include the literals as an effect. Each of the operator nodes are connected via conjunction edges to the literal nodes that appear in the operator's preconditions.

Example Consider the PSG for the literal `hold(A,P)` in the Parcel Delivery domain in Figure 3.2. Operator nodes are shaded, and disjunctive edges are dashed. This example PSG represents what operators can be performed (and what dependencies are required) to bring about the root literal. □

Etzioni showed that when constructing a PSG certain conditions could lead to the pruning of particular branches of the tree. He showed that literal branches need not be investigated if one of the following holds:

- **Unachievable**: : No operators contribute to the literal.
- **Goal Cycle**: : The literal is identical to an ancestor literal, a loop is identified.

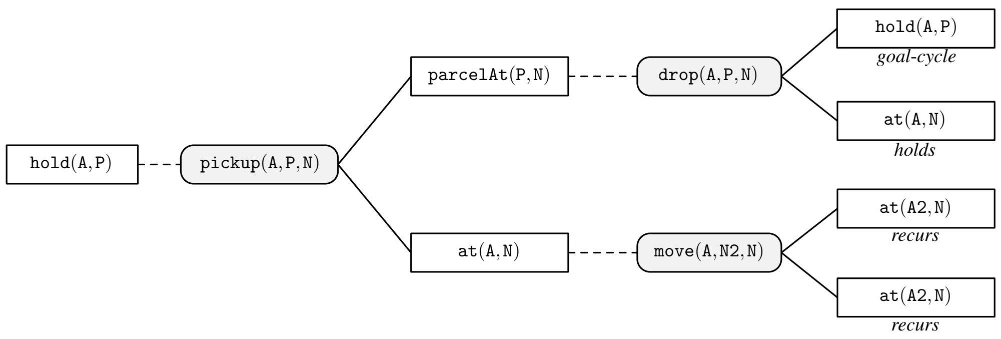


Figure 3.2: The PSG rooted with `hold(A,P)` in the Parcel Delivery domain.

- **Recurs**: : The literal unifies with, but is not identical to, one of its ancestors.
- **Holds**: : The literal is already necessarily satisfied, given that its ancestors in the PSG are subgoals waiting to be achieved.

These termination conditions were inspirational to our work, resulting in similar optimisations developed for conflict-rooted synthesis, and detailed in Chapter 5. PSGs had an impact on the automated planning community as well, leading to the development of planning techniques based on similar graph abstractions.

Planning graphs formed the basis for the Graphplan planning approach, and were originally proposed by Blum and Furst (1997). A planning graph is a data structure representing the search space for a relaxed version of the planning problem. It is important to note that it is not a state-space encoding: paths in a planning graph do not necessarily represent plans in the original problem instance, yet all possible plans in the original problem are included in those found in the relaxed space. As such a planning graph is akin to a constraint graph that encodes the original planning problem. Blum and Furst showed that, through an iterative deepening search of a planning graph, Graphplan could find solutions to the original planning problem more efficiently than existing planners. While Graphplan is now considered a dated planner, the planning graph representation is related to this work.

We detail planning graphs as proposed by Blum and Furst (1997), based on a STRIPS-like planning formalism. A planning graph is a directed level graph. The levels of the graph alternative between sets of *proposition* nodes and sets of *action* nodes. The root proposition level is populated with the propositions in the initial state of the planning problem. Edges represent relations between propositions and actions. Proposition nodes are connected to action nodes in the subsequent level via *precondition* edges if the proposition appears in the action's preconditions. Actions are connected to preconditions in the subsequent level through *add* edges if the action brings about the proposition, or *delete* edges if the proposition is deleted. Importantly, actions are included in an action level if *all* of the action's preconditions exist in the previous proposition level, however there is no requirement for independence between actions in the same level. Given a constructed planning graph to depth n, Graphplan guarantees to find a partially-ordered plan of length n if one exists, and will state if none exists.

Example Consider a planning problem in the Parcel Delivery domain with initial state specification $\{\mathtt{at(a_1,node_1),parcelAt(p_1,node_2)}\}$ and goal state specification $\{\mathtt{parcelAt(p_1,node_1)}\}$. We have omitted the conn predicates from the initial state specification for brevity, and have only shown two action levels. □

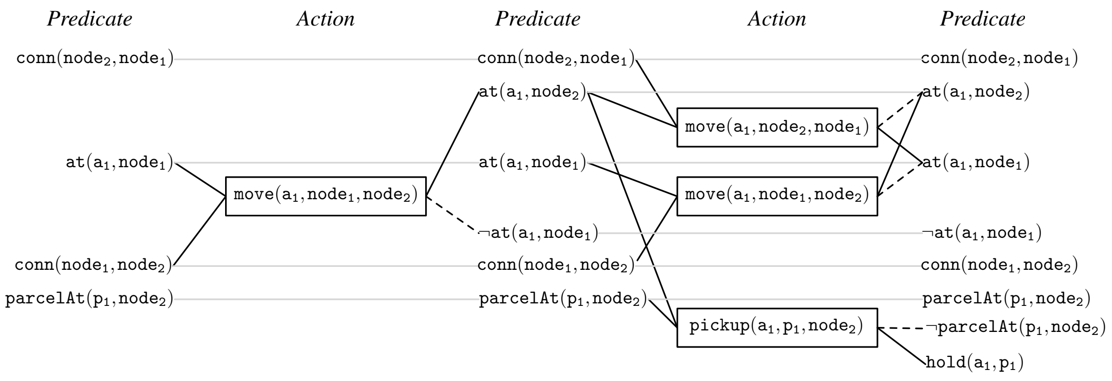


Figure 3.3: Example planning graph for plans of length 2 in the Parcel Delivery domain.

3.5.4 The Fast-Forward Planning System

One the of the benefits of conflict-rooted synthesis is that a substantial amount of computation is performed by a highly optimised planner. The Fast-Forward (FF) Planning System is a planner based on heuristic search proposed by Hoffmann and Nebel (2001) and is the planner integrated into our implementation. We utilise it extensively in our work to solve the planning problems produced during the reachability analysis portion of our algorithm, and therefore provide some key details on how solution plans are found. FF is a domain independent planner that searches for plans using two approaches:

1. **Enforced Hill Climbing**: The planner constructs a simpler, relaxed instance of the planning problem (in which STRIPS delete lists are ignored) and performs an enforced hill climbing search of this relaxed space. Successor states with the highest heuristic value are picked greedily. If no successor state has a higher heuristic value than the current, a breadth first search is adopted to find a sequence of actions that leads to a state with higher heuristic value.

2. **A* Search**: Heuristic hill climbing search may fail since no backtracking is performed. In this case FF falls back onto a standard A^* search strategy of the state space. This search process is guaranteed to find a solution if one exists.

FF quickly finds solutions through enforced hill climbing search, yet if this fails it can take a considerable amount of time searching the full state space using the A^* search. We adopt FF in our work for three main reasons:

1. It is very efficient at finding solutions when the heuristic search is successful.
2. It has an open source, efficient native implementation.
3. Although dated, it is still a strong performer on classical planning problems.

3.6 Conclusion

A key feature of conflict-rooted synthesis is the planning-based nature of the approach. By utilising state and action abstractions conflict-rooted synthesis is able to search a state space composed of sets of states rather than performing a complete state enumeration, and is able to produce norms that are generally applicable. We detailed two planning formalisms: a propositional set theoretic language followed by a more expressive classical representation. Conflict-rooted synthesis is detailed in the next chapter, first in propositional settings and subsequently extended to classical.

Conflict-rooted synthesis utilises a set of optimisations that simplify the resulting search performed. There are strong similarities between the way we reason about repeated applications of actions, and the subfield of generalised planning. The notions of planning with loops and operator iteration are strong themes in generalised planning. However the problems of plan synthesis and norm synthesis are very different: norm synthesis is not concerned with a single plan from initial to goal states, but rather any plan that traverses through the conflict-state space. In Chapter 8 we discuss the possible application of conflict-rooted synthesis to solving the planning problem.

Planning approaches to incorporate domain knowledge into the planning process are also related to norm synthesis. By specifying designer knowledge in the form of a social objective agents can synthesise norm-compliant plans, but these approaches do not allow for the synthesis of explicit norms. Additional similarities exist between Traversal Graphs as presented in Section 6.1.2 and work on problem space graphs and planning graphs, however our data structure allows us to efficiently represent sets of similar plans, using unground predicates that are independent of initial and goal states.

Chapter 4

Conflict-Rooted Synthesis

Conflict-rooted synthesis has two advantages over norm synthesis approaches based on complete system state enumerations:

1. It adopts a *localised search* of the undesirable state space, avoiding searching all states where possible.
2. It utilises *abstractions* implicit in operator schemata to search at a more general level between sets of states, rather than individual complete states.

In this chapter we present the theoretical details of our approach, with the core components previously published by Christelis and Rovatsos (2009). We begin by describing the process abstractly for state transition systems without confusing the core process with specific details regarding any particular planning formalism. We then provide two bindings between the abstract algorithm and specific planning formalism: first a mapping to the propositional formalism and later an extension to the classical formalism. We use the Parcel Delivery domain outlined in Section 1.4 to illustrate key concepts.

4.1 Synthesis Introduction

Given a domain and a specification of undesirable conflict states, norm synthesis is a procedure that creates a set of social norms for the provided domain that ensure the social objective. An additional key requirement of applicable norm synthesis is to provide guarantees on what effects the candidate norms have on goal reachability in the normative system. Our approach, called *Conflict-rooted synthesis*, synthesises norms while preserving agent autonomy and ensuring that the agents' goals are still achievable in the restricted system.

Intuitively, we can visualise the synthesis process as the construction and subsequent search of a directed graph, as in Figure 4.1. Nodes in the graph depict sets of system states that have common attributes, while edges depict the transition the system takes between states when an agent performs some action. Darker nodes depict sets of conflict states and all other nodes are conflict-free. Transitions in our model are deterministic and the world is closed implying that states reached through operator application are determined with certainty. In our systems there are no external forces that can alter the applicability or result of an operator.

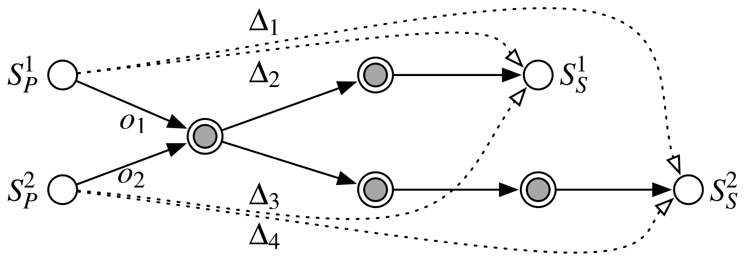


Figure 4.1: Norm synthesis depicted as reachability checking of a transition graph

We call the above representation a *transition graph*. The single edge depicting a transition between nodes represents a set of transitions between states contained in the connected sets, from *each* state in the source set, to *some* state in the target set. This is illustrated in Figure 4.2 where the set of states S_1 is connected to set S_2 by the transition operator o. This abstract transition models a set of four underlying transitions all invoked by o. Grouping states into sets based on common attributes allows us to reason about operator application on the set, rather than considering all states contained within it. One key benefit of our approach is that we actively maintain state abstractions to simplify the search process, and to produce more expressive results.

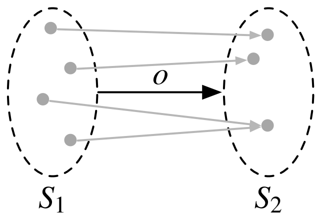


Figure 4.2: Single transitions between sets of system states

Suppose that we are given a set of conflict states S_C and a domain specification describing actions of agents in the system. From S_C we identify the *precursor* conflict-free sets that contain all states that, through the application of some operator, will lead to a state in S_C. In Figure 4.1 we have two precursor sets, labelled S_P^1 and S_P^2, and two actions that lead to conflict labelled o_1 and o_2. Similarly, we identify the *successor* conflict-free sets which contain all conflict-free states that are reachable from a state contained in the set S_C, through a traversal of conflict states. In the graph the states labelled S_S^1 and S_S^2 are future successor states. The transition graph represents a search of the system space from conflict-free states through conflict states. This search is termed localised, or *conflict-rooted*, since once a conflict-free successor state is encountered the forward search is terminated. The resulting synthesised norms prohibit any agent actions that lead from conflict-free to conflict states.

We next identify what effects the synthesised norms have on the reachability of conflict-free states. We regiment the original system with our prohibitionary norms to generate a restricted normative system, and then search for an alternative plan from each precursor to each successor conflict-free state. In Figure 4.1 reachability checks are depicted as dashed directed arcs labelled $\Delta_1 \ldots \Delta_4$ where each Δ_i represents a sequence of actions that form part of a detour plan that agents can follow to avoid the conflict states. If plans exist for all pairs of conflict-free states, then the imposition of these norms does not restrict the reachability of conflict-free states. Consider two arbitrary sequences of system states and actions as depicted in Figure 4.3.

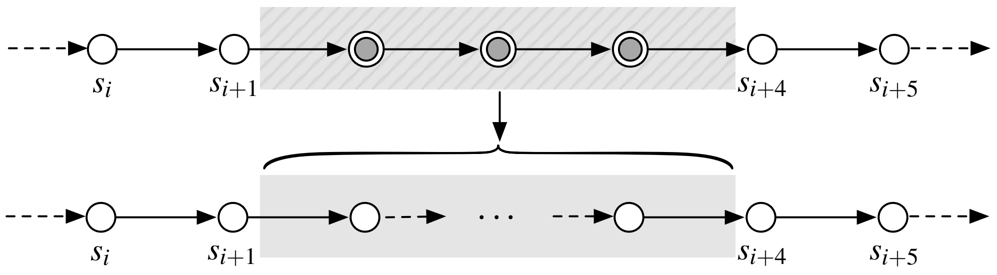


Figure 4.3: Reachability ensures that conflict sequences can be made conflict-free.

Darker nodes represent conflict states while all other nodes are conflict-free states. The upper sequence begins and ends in conflict-free states, but traverses through three conflict states. Under the synthesised norms this portion of the sequence is no longer applicable, but if we guarantee reachability to conflict-free states then an alternative sequence of operators exists that can be used as a substitute for the conflict sub-sequence. The combination of state search and reachability checks provide us with a synthesis process that not only achieves the social objective, but also ensures goal reachability in the restricted system.

4.1.1 Presentation Overview

We present conflict-rooted synthesis in stages beginning with a minimalist definition of the process, followed by additional theory for the propositional and classical planning formalism bindings. We present an overview in Figure 4.4. Conflict-rooted synthesis contains three stages: Conflict Traversal, Norm Synthesis and Reachability Analysis.

We describe the details of each of these steps in the context of a state transition system, and subsequently utilising propositional and classical planning domain formalisms.

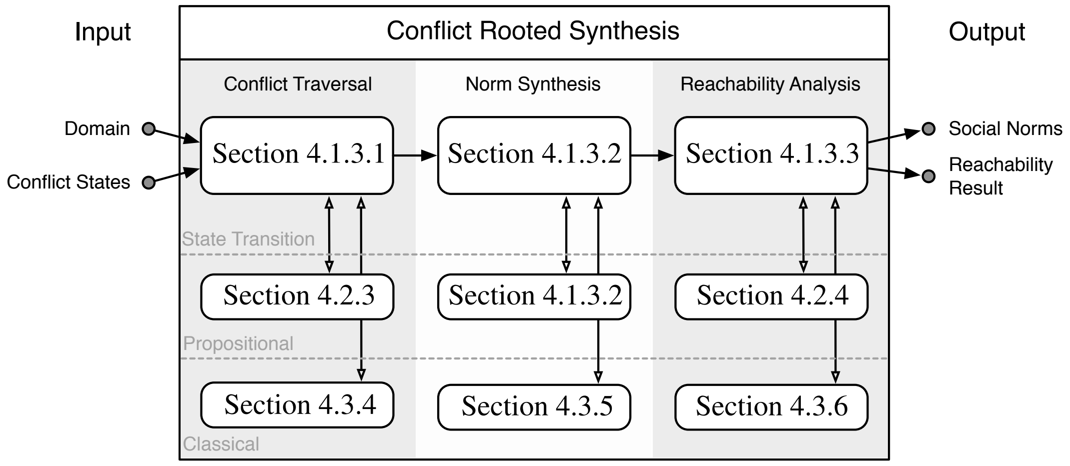


Figure 4.4: An overview of the presentation of the conflict-rooted synthesis algorithm.

We begin by presenting important notions and definitions involving search in a state transition system with no state or operator abstraction. In this work, we argue that searching a grounded system should only be performed as a last resort: if our domain contains implicit abstractions from the grounded system then synthesis should utilise this knowledge to simplify synthesis and to produce generally applicable norms. We follow the intuitive presentation of synthesis in a state transition system by illustrating the mapping to conflict search utilising state and operator abstractions, and follow this with a complete presentation of the conflict-rooted synthesis algorithm.

4.1.2 State Transition Definitions

Consider the definition of a simple state transition system presented in Section 3.1, where a transition system is a tuple $\langle \Sigma, A, \theta \rangle$ with Σ possible states, the set of actions A, and action invoked transition relation θ.

We begin by splitting the set of all possible states into two mutually exclusive subsets Σ_C and Σ_F where $\Sigma = \Sigma_C \cup \Sigma_F$ and $\Sigma_C \cap \Sigma_F = \emptyset$. Here, Σ_C is the set of all undesirable *conflict states* and Σ_F represents the set of desirable *conflict-free states*. As a shorthand, we write $s \xrightarrow{\mathtt{a}} s'$ to denote that a transition exists for action $a \in A$ between states s and s'. We chain operator and state sequences to form paths through the state transition system.

Definition 4.1.1. *A* path*, p, is defined as a traversal of the transition system:*

$$p = s_1 \xrightarrow{\mathtt{a}_1} s_2 \xrightarrow{\mathtt{a}_2} \dots \xrightarrow{\mathtt{a}_{n-1}} s_n$$

where $\forall i$, $s_i \in \Sigma$ and $a_i \in A$. We use indices, $p[i]$, to refer to the state i in the sequence and $p[\xrightarrow{\mathtt{a_i}}]$ to refer to action i.

Given a transition system and a clear notion of paths in the system we define the problem of prohibitionary norm synthesis in the context of a state transition system as:

Definition 4.1.2. *Given a state transition system $\langle \Sigma, A, \theta \rangle$ and the set of conflict states $\Sigma_C \subseteq \Sigma$, the problem of norm synthesis is to identify what actions must be forbidden to prevent access to these conflict states while ensuring that every previously connected pair of conflict-free states is still connected.*

We now consider a naive approach to solving this norm synthesis problem. Through repeated application of the transition relation from states specified in Σ it is possible to enumerate the entire transition system. Access to conflict states can be prohibited by denying transitions from conflict-free states to conflict-states. Checking reachability involves identifying all paths between conflict-free states in the original system, and ensuring that some alternative path exists in the normative system. This naive approach is infeasible in practice since:

1. systems defining the set of all states are unrealistic for even the smallest examples due to the large space requirements to represent the domain, and
2. the resulting norms are equally numerous since a unique norm is constructed for each transition that leads to a conflict state.

In order to solve the above issues a more succinct representation is required that abstracts away from the underlying state-based system.

4.1.2.1 State and Operator Abstractions

Instead of referring to each individual state of a system, we introduce concise descriptions of sets of states. By classifying states in the same set we are able to reason about transitions from the set, rather than transitions from each state.

We recount the notion of a *state specification* S (Nebel, 2000) as a succinct representation of a set of states and an operator specification o as a similar representation for sets of actions. We write $s \models S$ if the state s is one of the states represented by the specification S, and $a \models o$ if the action a represents an instance of the operator o. We write $Mod(S)$ to be the set of all states that model S. Furthermore we write 2^{Σ} to represent the set of all possible state specifications over the atoms Σ. A *conflict* *specification* is a specification where all represented states are conflict states, and a *conflict-free specification* represents only conflict-free states.

Example Suppose we wish for Agent $\mathtt{a}_1$ to avoid $\mathtt{node}_1$. Every state where Agent $\mathtt{a}_1$ is in $\mathtt{node}_1$ represents a conflict state. The set of these states forms the conflict specification, which we represent in natural language below:

$$S_C = \begin{cases} \text{state where } \mathtt{a}_1 \text{ is in } \mathtt{node}_1 \text{ and } \mathtt{a}_2 \text{ is in } \mathtt{node}_2 \\ \text{state where } \mathtt{a}_1 \text{ is in } \mathtt{node}_1 \text{ and } \mathtt{a}_2 \text{ is in } \mathtt{node}_3 \\ \text{state where } \mathtt{a}_1 \text{ is in } \mathtt{node}_1 \text{ and } \mathtt{a}_2 \text{ is in } \mathtt{node}_4 \\ \dots \end{cases}$$

□

Definition 4.1.3. *A run R is defined as a sequence of operators o_i and state specifications S_i:*

$$R = S_1 \xrightarrow{\mathtt{o}_1} S_2 \xrightarrow{\mathtt{o}_2} \dots \xrightarrow{\mathtt{o}_{n-1}} S_n$$

where $\forall i,\ S_i \in 2^S$ and $o_i \in O$.

Let 2^S represent the set of all state specifications. We write $|R|$ to represent the number of specifications in the run, $R[i]$ to refer to the i'th specification and $R[\xrightarrow{\mathtt{o}_i}]$ to refer to the i'th operator. We write *first*(R) and *last*(R) to represent the first and final state specifications in the sequence.

As visualised in Figure 4.5, a run represents a set of paths, one for every state represented by the initial specification S_1, where paths are traversals through states represented by specifications in R. Note that not every state in a subsequent specification is reachable from S_1.

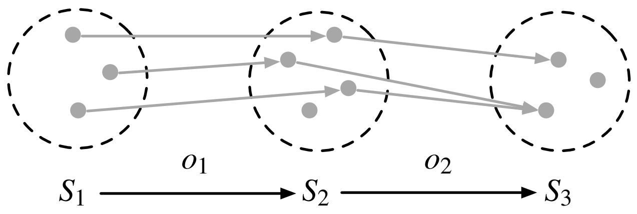


Figure 4.5: Mapping abstract runs to grounded paths.

Formally, we say a run R represents a path p ($p \models R$) if and only if $|R| = |p|$ and:

$$\forall j \leq |R| \,.\, p[j] \models R[j] \quad \wedge \quad \forall j < |R| \,.\, p[\xrightarrow{\mathtt{o}_j}] \models R[\xrightarrow{\mathtt{o}_j}].$$

Simply put, p models R if each of the specifications in R represents the corresponding state in p, and if each of the operators in R represents the corresponding action in p. We categorise a run by defining two mutually exclusive classes:

- A *complete* run is a run that originates and terminates in a conflict-free specification, but traverses only conflict specifications in between. Formally, a run R is complete if and only if $\mathit{first}(R)$ and $\mathit{last}(R)$ are conflict-free specifications, and $\forall_{1<i<|R|}\ R[i]$ is a conflict specification.
- An *incomplete* run is a run that originates in a conflict-free specification and subsequently only traverses conflict specifications. Formally, a run R is incomplete if and only if $\mathit{first}(R)$ is conflict-free and $\forall_{1<i}\ R[i]$ is a conflict specification

Runs are central to our presentation of conflict-rooted synthesis. As we progress through the search space we compile runs detailing what has been searched. Runs provide a simple and intuitive representation with which we can develop our approach, but more efficient data structures (such as the *traversal graphs* presented in Section 6.1.2) are used in practice. We utilise runs to represent what is achievable in the conflict state space of the system. From these runs we synthesise norms that prohibit the runs from occurring in the normative system. We introduce our social norm representation next and follow this final definition by introducing conflict-rooted synthesis.

4.1.2.2 Social Norm Representation

Prohibitionary social norms are behavioural constraints on the operators available to an agent. These behavioural constraints dictate whether an operator can be performed or not. Our norms are conditional on the current state of the system.

Definition 4.1.4. *We define a set of* prohibitionary norms *as* $\mathcal{N} = \{n_1, n_2 ...\}$ *where:*

$$n_i = \langle \varphi, o \rangle$$

- φ *is the* norm condition *which is a specification of a set of states, and*
- *o is the operator that is prohibited from being applied by any agent if the norm condition holds in the current state.*

We write $\varphi(n_i)$ and $o(n_i)$ to refer to the first and second components of the norm n_i.

Example We wish that agents do not collide in the Parcel Delivery world. Using natural language we construct a rule that prohibits this behaviour:

> *An agent is prohibited from moving to an adjacent node if this node is occupied by another agent.*

We construct a norm $n = \langle \varphi, o \rangle$ to represent this rule where φ is *if the adjacent node is occupied by another agent*, and o is that which results in the agent *moving* to this adjacent node. □

Previously we have only mentioned prohibitionary social norms, yet we can consider obligatory social norms as a more severe prescription of normative behaviour. A prohibitionary norm prohibits a single operator, yet obligatory norms define what operator must be performed. However, an obligation to perform some action is equivalent to a prohibition on all other actions and in this sense obligations are a form of restriction that is typically more severe than prohibitionary norms.

4.1.3 Conflict-Rooted Synthesis

We are now ready to define conflict-rooted synthesis in a state transition system. Let S_C be the provided *conflict state specification* representing the social objective. Given S_C and the set of operator schemata O we define synthesis as the function:

$$Synth(S_C, O) = \begin{cases} \mathcal{N} & \text{if the norms } \mathcal{N} \text{ prohibit } S_C \text{ and satisfy reachability,} \\ \bot & \text{otherwise.} \end{cases}$$

That is, can we produce a set of norms that avoids all states represented by S_C, given the operators the agents can perform? If so, the set of prohibitions $\mathcal{N}$ is produced as output, otherwise the function returns $\bot$ to denote failure. Conflict-rooted synthesis is modularised into three distinct stages:

1. **Conflict Traversal**: The search process conducted over the abstract state representation, identifying every achievable conflict run representing all action sequences that are prohibited in the normative system.
2. **Norm Synthesis**: A set of candidate norms is constructed that prohibits access to the identified conflict states.
3. **Reachability Analysis**: Ensure that each run identified in traversal is still achievable under the candidate norms.

Note that there is commonality between our notion of conflict-free states and the theory of focal states presented by Shoham and Tennenholtz (1995). Since we have assumed no knowledge of agent goals we assume all conflict-free states to be focal, and attempt to ensure that all are reachable.

4.1.3.1 Conflict Traversal

Conflict traversal is a *hypothetical* search of the conflict state space since we do not monitor an existing system but theorise about what agents could do. We make no assumptions about agent goals and weigh all paths through conflict states equally. The result of the traversal process is a set of runs, where each run begins and terminates in a conflict-free state, but where all intermediate states are conflict states.

Traversal may identify paths that are not reachable in the actual domain, or that may be disregarded once the goals of the agents are taken into consideration. This additional knowledge restricts which runs are achievable, yet in this work we do not assume this knowledge, implying that the runs generated during traversal represent a possible superset of the actual runs achievable. This distinction is important: by not considering domain specific knowledge we construct runs that are applicable in all domains that utilise the provided operator set. We present methods to restrict the set of generated runs in the presence of additional knowledge in Section 8.2.1.

We begin our presentation of conflict traversal by defining the following relationships between state specifications and operators as illustrated for a state specification S in Figure 4.6.

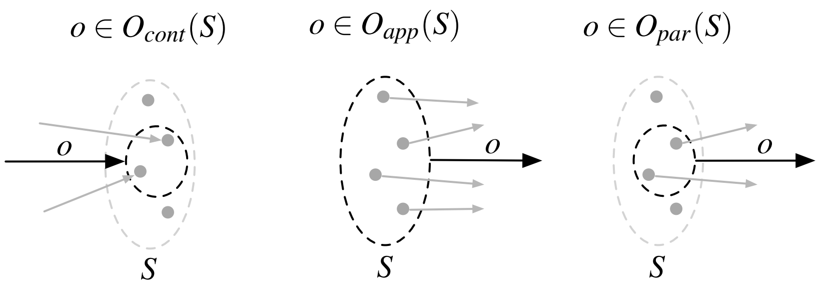


Figure 4.6: Contributing, applicable and partially-applicable operators.

We say that an operator *contributes* to a state specification if, through the application of the operator, the state specification is brought about.

Definition 4.1.5. *For state specification $S \in 2^{\hat{\Sigma}}$ and operator $o \in O$, o* contributes *to S if $\forall s \in Mod(S)$ $\exists s'$ such that* $\theta(s', o) = s$.

Given a specification S, we write the set of all contributing operators to be $O_{cont}(S)$ where contributing operators lead to a particular state specification. Mirroring this definition, we can specify the planning notion of *applicability*.

Definition 4.1.6. *For state specification $S \in 2^{\hat{\Sigma}}$ the operator $o \in O$ is* applicable *from S if $\forall s \in Mod(S)$ $\exists s'$ such that* $\theta(s, o) = s'$.

Given a specification S, we write the set of all applicable operators to be $O_{app}(S)$. It is beneficial for us at this point to refine this definition of applicability. Recall that we are interested in searching locally around conflict-state specifications which may represent an arbitrary number of system states and can be very abstract. By restricting our choice to applicable operators we are considering transitions that can be applied to *every* state represented by the specification. This causes two issues:

1. Since the specifications are abstract it is possible that there does not exist a single operator from which a transition can be made from every state.
2. Our desire is not to find operators that can be applied in all states, but to find all operators that can be applied in *any* of the states represented by the specification.

Example Suppose we wish to prohibit the set of states in the Parcel Delivery domain where Agent $\texttt{a}_1$ is at $\texttt{node}_1$. This very simple conflict specification is also very expressive: it accounts for all systems and possible worlds where $\texttt{a}_1$ is at $\texttt{node}_1$.

Consider that we wish to identify what $\texttt{a}_1$ might achieve in a conflict state, given the `move`, `drop` and `pickup` actions at their disposal. None of these operators are applicable directly in the conflict specification. There exists a subset of undesirable states where each operator is applicable, yet by considering the set of states as a monolithic entity we are unable to reason about what is achievable from any subsets. □

We require a notion of a *partially applicable* operator which is applicable in some subset of states represented by a specification.

Definition 4.1.7. *For any state specification $S \in 2^{\hat{\Sigma}}$, an operator o is partially applicable from S if $\exists s \in Mod(S)\ \exists s'$ where $\theta(s,o) = s'$.*

Given a specification S, we write the set of all partially applicable operators to be $O_{par}(S)$.

4.1.3.1.1 Inference and Refinement Conflict traversal utilises two functional building blocks when searching for complete runs: state inference and state refinement.

Example Recall the conflict specification S_C of undesirable states where Agent $\texttt{a}_1$ is at location $\texttt{node}_1$. We wish to identify the successor state specifications that are possible by applying actions in S_C. We have a means of identifying the set O_{par} of partially applicable operators, yet for each of these operators we require a means of identifying the states that result from applying each operator.

As before, there are states represented by S_C where the agent can `pickup` a parcel, `move` or `drop` a parcel. The following table lists possible states represented by S_C, and some of $\mathtt{a}_1$'s applicable operators in these states.

Example state represented by S_C	Applicable Action
Agent $\mathtt{a}_1$ at $\mathtt{node}_1$ and $\mathtt{parcel}_1$ at $\mathtt{node}_1$	`pickup` $\mathtt{parcel}_1$
Agent $\mathtt{a}_1$ at $\mathtt{node}_1$ and is carrying $\mathtt{parcel}_1$	`drop` $\mathtt{parcel}_1$
Agent $\mathtt{a}_1$ at $\mathtt{node}_1$ and $\mathtt{node}_1$ adjacent to $\mathtt{node}_2$	`move` to $\mathtt{node}_2$

We require a mechanism to identify subsets of a specification where operators are applicable, and a means of computing the effect of performing these operators. □

In order to reason about the application of partially applicable operators we *refine* the specification to only include states from where the operator is fully applicable, and subsequently *infer* the next specification in the sequence. We begin by introducing *forward* refinement and inference.

Definition 4.1.8. *Given a state specification S and a partially applicable operator $o \in O_{par}(S)$, we define* forward refinement *as a function $\overrightarrow{Refine} : 2^{\hat{\Sigma}} \times O \to 2^{\hat{\Sigma}}$ where:*

$$\overrightarrow{Refine}(S,o) = S' \textit{ where } o \in O_{app}(S') \textit{ and } Mod(S') \subseteq Mod(S).$$

Let S' be the subset of states represented by S in which o is applicable. The operator is partially applicable in S, but fully applicable in S'.

Example Let S_1 be the specification where Agent $\mathtt{a}_1$ is at $\mathtt{node}_1$ and o be the operator that `moves` $\mathtt{a}_1$ from $\mathtt{node}_1$ to $\mathtt{node}_2$. This operator is only applicable in a subset of the states represented by S_1: those states where $\mathtt{node}_1$ is adjacent to $\mathtt{node}_2$. We call this subset S_1' and define forward refinement as the means of calculating this specification from S_1. We depict this in Figure 4.7.

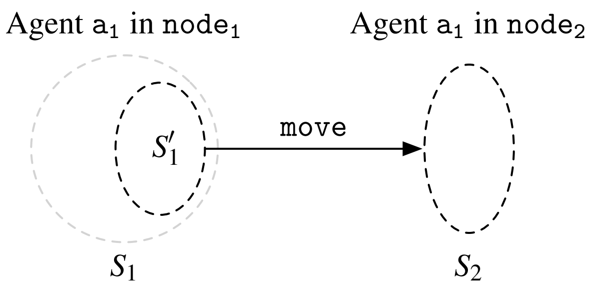

Agent $\mathtt{a}_1$ in $\mathtt{node}_1$
Agent $\mathtt{a}_1$ in $\mathtt{node}_2$
S_1'
move
S_1
S_2

Figure 4.7: Refining S_1 to S_1' so that operator `move` is applicable.

□

Next we introduce a function that allows us to construct the successor specification that represents the outcome of applying an operator in a state specification.

Definition 4.1.9. *We define* forward inference *as a function* $\overrightarrow{Infer} : 2^{\hat{\Sigma}} \times O \rightarrow 2^{\hat{\Sigma}}$ *where:*

$$\overrightarrow{Infer}(S_1, o) = S_2 \textit{ where } \forall s \in Mod(S_1)\ \exists s' = \theta(s, o) \textit{ such that } s' \in Mod(S_2).$$

Here, o is applicable in all states represented by S_1. This is a generalisation of the operator application presented in Sections 3.2.3.1. In Figure 4.7, the outcome of inference applied to S'_1 and operator `move` is the specification S_2.

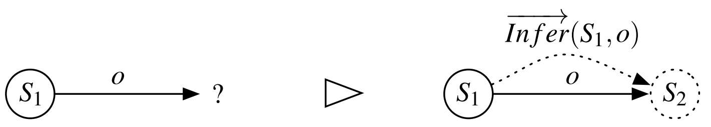


Figure 4.8: Inferring S_2 through the application of operator o in S_1

Forward refinement and inference provide tools with which we can begin a forward search of the state specification space to produce runs. This forward search alone is not sufficient for synthesis since our search begins within the conflict space. We present backwards refinement and inference next to identify the conflict-free precursor states.

Consider a given state specification S. The set of contributing operators that lead to states represented by S is O_{cont}. We introduce a reverse refinement operator that restricts the state specification S for a particular operator so that all states represented by the restricted specification are reached through the application of o.

Definition 4.1.10. *Formally, we define* reverse refinement *as a function* $\overleftarrow{Refine} : 2^{\hat{\Sigma}} \times O \rightarrow 2^{\hat{\Sigma}}$ *where:*

$$\overleftarrow{Refine}(S, o) = S' \textit{ where } \forall s' \in Mod(S') \textit{ it holds that } s' \in Mod(S) \textit{ and } \exists s.\theta(s, o) = s'.$$

The state specification S is refined into a more restrictive version S' where, for a given operator o, all states represented by S' can be reached through the application of o. Again, we note that $Mod(S') \subseteq Mod(S)$.

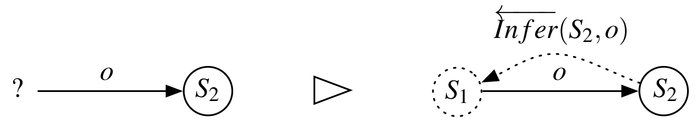


Figure 4.9: Inferring the specification S_1, from which the application of operator o leads to S_2.

While reverse refinement allows us to identify the portion of a state specification that can be reached by the application of an operator o, reverse inference allows us to identify the states from where o could be applied. We call these states the *precursor states*, and say that the *precursor specification* is modelled by these states. Here, reverse inference is a function that determines the precursor specification.

Definition 4.1.11. *Formally, we define* reverse inference *as a function* $\overleftarrow{Infer} : 2^{\hat{\Sigma}} \times O \to 2^{\hat{\Sigma}}$ *where:*

$$\overleftarrow{Infer}(S_2, o) = S_1 \text{ where } \forall s' \in Mod(S_2)\ \exists s \in Mod(S_1) \text{ such that } s' = \theta(s, o).$$

Just as forward inference is akin to operator applicability, so reverse refinement is akin to inverse operator application in backward-search methods (Nau et al., 2004b). We now detail a number of key features of inference and refinement:

- There is no need to distinguish between partially applicable, and applicable operators during inference and refinement, since fully applicable operators simply result in no subsequent refinement.
- Runs never contain partially applicable operators. Specification refinement ensures that operators used to construct future specifications are fully applicable in the refined specification, and included in the runs.

4.1.3.1.2 Run Refinement *Run refinement* occurs when applying partially applicable operators in the final state specification of a run and is an extension of forward refinement. Recall that a run is a specification and operator sequence of the form:

$$R = S_1 \xrightarrow{o_1} \dots \xrightarrow{o_{n-1}} S_n.$$

Consider a partially applicable operator o to be applied in S_n. Refining S_n has a potential impact on all other specifications in the run.

Definition 4.1.12. *Consider a run R and operator $o \in O_{par}(last(R))$. Here, o is partially applicable in the last state specification of R. Run refinement is a function:*

$$RunRefine(R, o) = S'_1 \xrightarrow{o_1} \dots \xrightarrow{o_{n-1}} S'_n$$

where

- $S'_n = \overrightarrow{Refine}(S_n, o)$ *using forward refinement, and*
- $\forall i < n$ *if* $\exists s \models S_i$, $s' \models S'_n$ *where a path exists from from s to s' then* $s \models S'_i$.

We illustrate run refinement in Figure 4.10 where the state specifications of R are larger ovals and the refined specifications of R' are internal, smaller ovals. The run R is refined when the operator o is considered as a successor in S_3. Since o is only applicable in one of the states represented by S_3 the state specifications of the refined run R' do not include references to paths from which o cannot be applied. As this figure illustrates, runs become more refined as more partially applicable operators are considered.

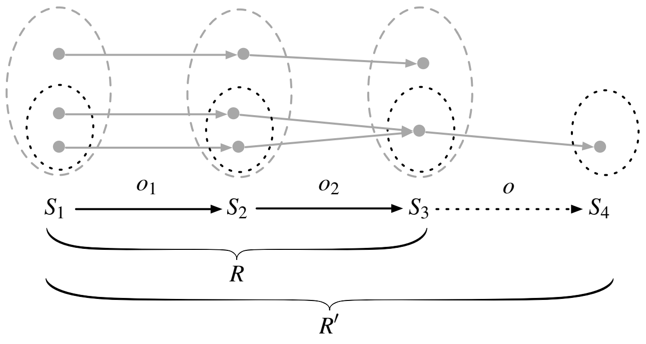


Figure 4.10: Run refinement illustrating how paths represented by the run R are discarded as a new partially applicable operator o is considered in the refined run R'.

Example Lets consider a very simple run in our Parcel Delivery domain:

$$R = S_1 \xrightarrow{\texttt{move}} S_2$$

where S_1 represents the set of states where Agent $\texttt{a}_1$ is at $\texttt{node}_1$, and S_2 the set of states where Agent $\texttt{a}_1$ is now at $\texttt{node}_2$. Consider a successor operator `pickup` where the agent picks a parcel up in $\texttt{node}_2$. For $\texttt{a}_1$ to pick up a parcel in S_2 it must be the case that a parcel exists at location $\texttt{node}_2$. Since the `move` operator did not introduce the parcel, then it must exist in S_1 as well. Run refinement is adopted to include the knowledge of the parcel into the run, thereby ensuring that the `pickup` action can be applied in S_2. □

4.1.3.1.3 Conflict Traversal Algorithm

With our state transition system semantics for inference, refinement and run refinement we now present the conflict traversal algorithm in its entirety.

Definition 4.1.13. *Conflict traversal is defined as a function*

$$Traversal(S_C, O) = \mathcal{R}$$

where:

- *S_C is a specification of conflict states,*
- *O is the set of operators defined in the domain, and*
- *$\mathcal{R}$ is a set of complete runs.*

We present the entire procedure in Algorithm 1 with explanation next:

Line	Explanation and Comments
3–4	Identify the set of precursor operators $O_P \subseteq O$ that contribute to S_C. Each operator leads to conflict.
5–6	For each operator o_i, identify the precursor states from which the operator can be applied. Apply reverse state refinement on the conflict specification followed by state inference to construct the precursor specification.
7	Check to ensure that the inferred state specification is conflict-free. If it is not, the specification is ignored.
8–9	Initialise a run with the inferred precursor specification, contributing operator, and the refined conflict specification. Initialise the set $\mathcal{U}$ containing all incomplete runs.
10	Iterate until there are no more incomplete runs to consider.
11–13	For each incomplete run retrieve the last state specification and all operators partially applicable in this specification.
14–19	For each partially applicable operator construct a refined version of the original run from which we append the operator and successor specification.
20	Check to ensure that all specifications in the new run are consistent, and that no loops exist. Inconsistent runs, or runs with loops, are discarded.
21–24	If the run is complete add it to $\mathcal{R}$, else add it to $\mathcal{U}$.
25	The algorithm terminates when there are no longer any incomplete runs in the $\mathcal{U}$ set. At this point, the set of complete runs found is returned.

As our search is exhaustive there is no benefit in examining runs with repeated specifications since all eventualities will have been considered the first time the specification was encountered. We detect loops by scanning for repeated state specifications in a run, where runs with loops are discarded. The role of loop detection is further discussed in Section 4.2.5.1.

Naturally, the parallels between conflict traversal and plan projection exist. Plan projection follows directly from the application of operators to state specifications. If o is applicable for some state specification S and $S' = R(S,o)$ then $\forall s \models S$, there exists an s' such that $(s \xrightarrow{o} s') \in \theta$ and $s' \models S'$. The important fragment here is that plan projection only considers operators that are fully applicable, whereas conflict-rooted

Algorithm 1: Conflict-Free Run Traversal

```
   Input: Conflict specification $S_C$, and list of operator schemata $O$
   Result: The set $\mathcal{R}$, containing complete runs
1  begin
2      $\mathcal{R} \longleftarrow \{\}$
       Identify all operators contributing to conflict
3      $O_P \leftarrow O_{cont}(S_C)$
4      for each precursor operator $o_i \in O_P$ do
           Reverse state refinement and inference
5          $S'_C \leftarrow \overleftarrow{Ref}(S_C, o_i)$
6          $S_P \leftarrow \overleftarrow{Inf}(o_i, S'_C)$
           Ensure that $S_P$ is not a conflict state specification
7          if $S_P$ is not a conflict state then
               Run initialisation
8              $R \leftarrow (S_P \xrightarrow{o_i} S'_C)$
9              $\mathcal{U} \longleftarrow \{R\}$
               Stop iterating when no more unsafe runs exist
10             while $|\mathcal{U}| > 0$ do
11                 $R \leftarrow RemoveFirst(\mathcal{U})$
12                 $S_{last} \leftarrow last\,(R_i)$
13                 $O_S \leftarrow O_{par}(S_{last})$
                   Consider each successor operator in turn
14                 for each successor operator $o_j \in O_S$ do
15                     $R_j \leftarrow R$
                       Forward state and run refinement and inference
16                     $last(R_j) \leftarrow \overrightarrow{Ref}(last\,(R_j),\, o_j)$
17                     $S_S^j \leftarrow \overrightarrow{Inf}(o_j,\, last\,(R_j))$
18                     $R_j \leftarrow RunRefine\,(R_j,\, S_S^j)$
                       Create a new run for each successor
19                     $R'_j \leftarrow R_j \xrightarrow{o_j} S_S^j$
20                     if $Consistent\,(R'_j)$ and $(S_S^j \notin R_j)$ then
21                         if $S_S^j$ is not a conflict state then
22                             $\mathcal{R} \leftarrow \mathcal{R} \cup R'_j$
23                         else
24                             $\mathcal{U} \leftarrow \mathcal{U} \cup R'_j$
25     return $\mathcal{R}$
26 end
```

synthesis also considers partially applicable ones. We are not only interested in the conflict-free specifications reachable from *every* state represented by S_C, but also in the conflict-free specifications accessible from *any* state represented by S_C.

4.1.3.2 Norm Synthesis

Algorithm 2 details the Norm Synthesis stage where social norms are generated for each complete run. As input this algorithm takes the set of runs $\mathcal{R}$ generated during traversal, and as output it produces a set of prohibitionary norms $\mathcal{N}$.

Algorithm 2: Synthesising Prohibitionary Norms

```
  Input: The set of complete runs $\mathcal{R}$
  Result: A set of prohibitionary social norms $\mathcal{N}$
1 begin
      Initialise the set of prohibitionary norms
2     $\mathcal{N} \leftarrow \{\}$
3     for each run $R \in \mathcal{R}$ do
          Create a norm for this run
4         $\varphi \leftarrow R[0]$
5         $o \leftarrow R[\overset{0}{\rightarrow}]$
          Append the norm to the set
6         $\mathcal{N} \leftarrow \mathcal{N} \cup \langle \varphi, o \rangle$
7     return $\mathcal{N}$
8 end
```

Norm synthesis is the simplest stage in our approach. For each complete run we synthesise a unique social norm, where the components of a norm tuple are extracted from each complete run found during traversal. The condition of the norm is the first conflict-free specification and the prohibited operator is the contributing action in the run. These simple steps, conducted on each complete run, allow us to synthesise a complete set of social norms.

Example Suppose we identify the following complete run in our Parcel Delivery domain:

$$\{\text{Agent at } \texttt{node}_3\} \xrightarrow{\texttt{move to node}_1} \{\text{Agent at } \texttt{node}_1\} \rightarrow \ldots$$

where $\texttt{a}_1$ should not be in $\texttt{node}_1$. One action that leads to conflict is when Agent $\texttt{a}_1$ moves from $\texttt{node}_3$ to $\texttt{node}_1$. The synthesised norm for this run would be $n = \langle \varphi, o \rangle$ where:

- φ is the specification representing all states where Agent $\texttt{a}_1$ is at $\texttt{node}_3$, and
- o is the action where Agent $\texttt{a}_1$ moves from $\texttt{node}_3$ to $\texttt{node}_1$. □

4.1.3.3 Reachability Analysis

The set of synthesised norms is guaranteed to avoid conflict states but we must identify the size of the conflict-free state space that will no longer be reachable. Figure 4.11 presents a graphic representation of this where the state space is split into conflict-free (S_F) and shaded conflict (S_C) regions. Reachability analysis is interested in the portion of the conflict-free state space affected through the prohibition of the conflict state space. This portion of the conflict-free space (bounded in the diagram with a dashed border) should be equal to S_C, symbolising that the norms only remove conflict states.

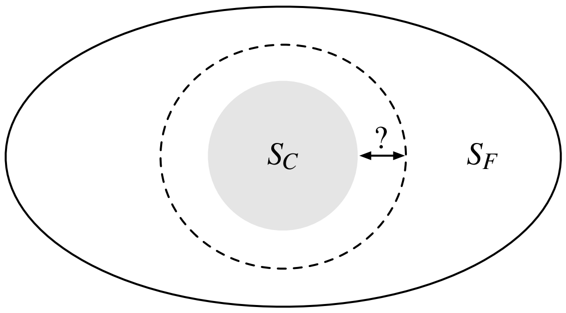


Figure 4.11: Identifying the conflict-free space prohibited under norms

Definition 4.1.14. *Reachability analysis is a function of the form:*

$$Reachability(\mathcal{R}, \mathcal{N}, \Xi) = \begin{cases} \top & \textit{if reachability holds in } \Xi \textit{ under norms } \mathcal{N} \\ \bot & \textit{otherwise} \end{cases}$$

where:

- $\mathcal{R}$ *is the set of complete runs generated during conflict traversal,*
- $\mathcal{N}$ *is the set of prohibitionary norms generated during norm synthesis, and*
- Ξ *is the domain specification.*

The function returns $\top$ *to represent reachability, and* $\bot$ *to represent failure.*

To evaluate reachability we require a means of constructing the normative system with the candidate norms. Using our state transition semantics we can easily construct an alternative domain structure for reachability checking by constructing a new transition function, θ', as follows:

$$\theta'(s,o) = \begin{cases} \theta(s,o) & \text{if } \forall \langle \varphi, o \rangle \in \mathcal{N} \,.\, s \not\models \varphi \\ \perp & \text{otherwise} \end{cases}$$

The new transition function is a restricted form of the original and we call the process of domain restriction *norm application*. When we introduce the bindings to particular planning formalisms we readdress the issue of norm application and introduce techniques that utilise the state and operator abstractions to avoid requiring the complete transition relation.

We reduce the problem of reachability checking to instances of the planning problem where, for each complete run we search for a conflict-free alternative in the normative system. Consider an arbitrary complete run:

$$R = S_1 \xrightarrow{\mathtt{o_1}} S_2 \xrightarrow{\mathtt{o_2}} \ldots \xrightarrow{\mathtt{o_{n-1}}} S_n.$$

For this run we now construct a new planning problem $\Pi_N = \langle \Xi, S_I, S_G \rangle$ where:

- Ξ is a tuple $\langle \Sigma, O \rangle$ where Σ is the set of states in the transition system, and O is the set of operators,
- $S_I = R[1]$, the initial state specification is the first specification of the run, and
- $S_G = R[n]$, the goal state specification is the last specification of the run.

In order to ensure that the alternative conflict-free plan is identical to the complete run we introduce one additional constraint. The accessibility conditions are satisfied by a conflict-free plan $\overline{\Delta}$ and original grounded conflict plan Δ if the effects of $\overline{\Delta}$ are *identical* to the effects of Δ. We define a solution to Π_N that satisfies these additional requirements to be a *valid* solution.

Apart from reducing the development effort of implementing conflict-rooted synthesis, mapping the reachability checking to a planning problem in this way provides two significant benefits:

1. Reachability checking is *more efficient* since the mapping to a standardised problem representation allows us to adopt state of the art planning technologies.
2. This increased *modularity* allows conflict-rooted synthesis to easily use domain specific planning approaches or technologies.

We present the reachability analysis in Algorithm 3. A simple extension of this reachability analysis approach is to return the subset of runs that are identified as not being

Algorithm 3: Reachability Analysis with Candidate Norms

```
Input: The set of complete runs $\mathcal{R}$, candidate norms $\mathcal{N}$ and the original domain structure $\Xi$
Result: TRUE if all runs are reachable and reachability is satisfied, FALSE otherwise.
1  begin
       Create the restricted prohibitionary system
2      $\Xi' \leftarrow NormApplication(\Xi, \mathcal{N})$
       Ensure reachability between conflict-free states of each run
3      for each complete run $R \in \mathcal{R}$ do
           Construct a planning problem to verify reachability
4          $S_I \leftarrow first(R)$
5          $S_G \leftarrow last(R)$
6          $\Pi \leftarrow \langle \Xi', S_I, S_G \rangle$
           Invoke a planner to solve the planning problem
7          $\Delta \leftarrow InvokePlanner(\Pi)$
8          if $\Delta$ is not a valid solution then
9              return FALSE
       All the runs have been verified as reachable
10     return TRUE
11 end
```

reachable, rather than simply a Boolean value. This allows us to quantify exactly which runs are no longer reachable in the restricted system. Note that the algorithmic properties of our reachability analysis process are very dependent on the planner adopted. For the purposes of this discussion we assume a sound and complete planner.

4.2 Propositional Synthesis

Our state transition description of conflict-rooted synthesis avoided associating the abstract algorithm with any particular state space representation, choosing rather to adopt the notions of states and state specifications at an intuitive level. The requirement to enumerate all states in a domain representation is not realistic in practice. A representation that abstracts away from a transition system is required that allows for the real world specification of domains. In this section we provide our first binding between the conflict-rooted synthesis algorithm and a *propositional* set theoretic planning formalism previously presented in Section 3.2. We use the term *binding* since we do not redefine the entire framework but instead extend it where appropriate. We write s to represent a state in this formalism, and S a specification of a set of states. Operators are tuples of the form $\langle name, pre, post \rangle$. We write O to represent the set of all operators.

4.2.1 Propositional Parcel Delivery Domain

We reintroduce the Parcel Delivery domain in this setting encoding state attributes using propositions. For clarity in our representation, propositions take a parenthesised predicate form. This is simply a notational feature, and in implementation each proposition could simply be mapped to an acceptable representation. For example, $\mathtt{at}(\mathtt{a}_1,\mathtt{node}_1)$ could become $\mathtt{at\_a}_1\mathtt{\_node}_1$.

The possible locations of $\mathtt{a}_1$ and $\mathtt{a}_2$ are defined using one of the following propositions:

$$\mathtt{at}(\mathtt{a}_1,\mathtt{node}_1), \mathtt{at}(\mathtt{a}_1,\mathtt{node}_2), \ldots$$
$$\mathtt{at}(\mathtt{a}_2,\mathtt{node}_1), \mathtt{at}(\mathtt{a}_2,\mathtt{node}_2), \ldots$$

The location of a parcel is defined similarly:

$$\mathtt{parcelAt}(\mathtt{parcel}_1,\mathtt{node}_1), \mathtt{parcelAt}(\mathtt{parcel}_1,\mathtt{node}_2), \ldots$$

We use a hold atom to symbolise that a particular agent is currently carrying a parcel:

$$\mathtt{hold}(\mathtt{a}_1,\mathtt{parcel}_1), \mathtt{hold}(\mathtt{a}_2,\mathtt{parcel}_1), \ldots$$

We represent the topology of the graph through directed node connectives as follows:

$$\mathtt{conn}(\mathtt{node}_1,\mathtt{node}_2), \mathtt{conn}(\mathtt{node}_1,\mathtt{node}_3), \ldots$$

Our operator schemata are domain specific. For the purposes of brevity in this overview we present a single instance of the move, drop and pickup operators. We begin with a move operator which moves Agent $\mathtt{a}_1$ from $\mathtt{node}_1$ to $\mathtt{node}_2$:

$$\begin{aligned}\text{OPERATOR:}\quad & \mathtt{move}(\mathtt{a}_1,\mathtt{node}_1,\mathtt{node}_2)\\ \text{PRE:}\quad & \{\mathtt{at}(\mathtt{a}_1,\mathtt{node}_1), \mathtt{conn}(\mathtt{node}_1,\mathtt{node}_2)\}\\ \text{POST:}\quad & \{\neg\mathtt{at}(\mathtt{a}_1,\mathtt{node}_1), \mathtt{at}(\mathtt{a}_1,\mathtt{node}_2)\}\end{aligned}$$

We illustrate the pickup operator with which Agent $\mathtt{a}_1$ picks up a parcel in $\mathtt{node}_1$:

$$\begin{aligned}\text{OPERATOR:}\quad & \mathtt{pickup}(\mathtt{a}_1,\mathtt{parcel}_1,\mathtt{node}_1)\\ \text{PRE:}\quad & \{\mathtt{at}(\mathtt{a}_1,\mathtt{node}_1), \mathtt{parcelAt}(\mathtt{parcel}_1,\mathtt{node}_1)\}\\ \text{POST:}\quad & \{\neg\mathtt{parcelAt}(\mathtt{parcel}_1,\mathtt{node}_1), \mathtt{hold}(\mathtt{a}_1,\mathtt{parcel}_1)\}\end{aligned}$$

Finally, Agent $\mathtt{a}_1$ drops a $\mathtt{parcel}_1$ in $\mathtt{node}_1$ using the following operator:

$$\begin{aligned}\text{OPERATOR:}\quad & \mathtt{drop}(\mathtt{a}_1,\mathtt{parcel}_1,\mathtt{node}_1)\\ \text{PRE:}\quad & \{\mathtt{at}(\mathtt{a}_1,\mathtt{node}_1), \mathtt{hold}(\mathtt{a}_1,\mathtt{parcel}_1)\}\\ \text{POST:}\quad & \{\neg\mathtt{hold}(\mathtt{a}_1,\mathtt{parcel}_1), \mathtt{parcelAt}(\mathtt{parcel}_1,\mathtt{node}_1)\}\end{aligned}$$

Example Consider the specification where we do not wish Agent $\mathtt{a}_1$ to enter $\mathtt{node}_1$:

$$S_C = \{\mathtt{at}(\mathtt{a}_1,\mathtt{node}_1)\}$$

where $\mathtt{at}(\mathtt{a}_1,\mathtt{node}_1)$ is a predicate identifying the location of $\mathtt{a}_1$ as $\mathtt{node}_1$. There is no need to list all states in order to define the set. Similarly, we can define specifications involving multiple agents. The conflict specification to prohibit agents $\mathtt{a}_1$ and $\mathtt{a}_2$ from both being in $\mathtt{node}_1$ simultaneously is:

$$S_C = \{\mathtt{at}(\mathtt{a}_1,\mathtt{node}_1), \mathtt{at}(\mathtt{a}_2,\mathtt{node}_1)\}. \qquad \square$$

4.2.2 Propositional Norms

The definition of a prohibitionary norm in a propositional planning formalism follows directly from before. A prohibitionary norm is a tuple $n_i = \langle \varphi, o \rangle$ where:

- $\varphi \in 2^{\hat{\Sigma}}$, and
- $o \in O$.

Our norm representation changes accordingly with the increased expressiveness of our state representation. Here the representation of the condition of the norm is defined as a set of literals over the atoms in our planning formalism.

4.2.3 Conflict Traversal

An operator o is *applicable* in a state specification S if $S \models pre(o)$. Since we are utilising a set-based representation the satisfiability relation $\models$ follows:

- $S \models pre(o)$ is equivalent to $pre(o) \subseteq S$, and
- $S \not\models pre(o)$ is equivalent to $pre(o) \not\subseteq S$.

Given two state specifications S_1 and S_2, we write $S_1 \models S_2$ if $S_2 \subseteq S_1$. That is, every state represented by S_2 will also be represented by S_1. We now extend our definitions for *contributing* and *partially-applicable* operators in this propositional formalism based on the specification transition function R presented in Section 3.2.3.1.

Definition 4.2.1. *For a state specification $S \subseteq \hat{\Sigma}$ and operator $o \in O$, o* contributes *to S in the propositional planning formalism if:*

1. *$\exists l \in (post(o) \backslash pre(o))$ where $l \in S$, and*

2. $\nexists l \in \big((pre(o) \backslash \neg post(o)) \cup post(o)\big)$ *where* $\neg l \in S$.

Operator o contributes to S if the application of o from some state specification results in at least one literal that occurs in S, so long as none of the effects of o contradict literals in S, and none of the preconditions of o that are not removed by $\neg post(o)$ are inconsistent with S.

Example Let $S_C = \{\texttt{at(a}_1\texttt{,node}_1\texttt{)}\}$. Two operators can contribute to S_C: when $\texttt{a}_1$ moves from $\texttt{node}_2$ to $\texttt{node}_1$, and from $\texttt{node}_3$ to $\texttt{node}_1$. The set of contributing operators follows:

$$O_{cont}(S_C) = \{\ \texttt{move(a}_1\texttt{,node}_2\texttt{,node}_1\texttt{)},\ \texttt{move(a}_1\texttt{,node}_3\texttt{,node}_1\texttt{)}\ \}. \qquad \square$$

Definition 4.2.2. *For a state specification* $S \subseteq \hat{\Sigma}$ *and operator* $o \in O$, o *is* partially applicable *in* S *if:*

$$\forall l \in S \ \nexists l' \in pre(o) \textit{ such that } (l = \neg l' \vee l' = \neg l).$$

An operator o is partially applicable in S if there exists no literal in S that is the negation of a precondition of o. More simply, o is partially applicable if it is not explicitly forbidden in S. A succinct set theoretic representation for this, with equivalent meaning, is $(\neg S) \cap pre(o) \neq \emptyset$. We interchange between the two definitions in this text. We refer to this function as $O_{par} : 2^{\hat{\Sigma}} \rightarrow 2^O$.

Example Let $S_C = \{\texttt{at(a}_1\texttt{,node}_1\texttt{)}\}$. Here, the set of partially applicable operators are any of those that, as one of their preconditions, require $\texttt{a}_1$ to be in $\texttt{node}_1$. For a single parcel in the domain, $\texttt{parcel}_1$, the partially applicable operator set follows:

$$O_{par}(S_C) = \left\{ \begin{array}{c} \texttt{move(a}_1\texttt{,node}_1\texttt{,node}_2\texttt{)},\ \texttt{move(a}_1\texttt{,node}_1\texttt{,node}_3\texttt{)}, \\ \texttt{pickup(a}_1\texttt{,node}_1\texttt{,parcel}_1\texttt{)},\ \texttt{drop(a}_1\texttt{,node}_1\texttt{,parcel}_1\texttt{)} \end{array} \right\}. \qquad \square$$

4.2.3.1 State Inference and Refinement

We now redefine state inference and state refinement in the propositional planning formalism.

Definition 4.2.3. *Given a state specification* S_1 *and applicable operator* o*, we define* forward inference *as the function to construct the state specification* S_2 *where:*

$$S_2 = \overrightarrow{Infer}(S_1, o) = \big(S_1 \backslash \neg post(o)\big) \cup post(o).$$

The forward inference follows from the operator application function $R : 2^{\hat{\Sigma}} \times O \rightarrow 2^{\hat{\Sigma}}$ we defined in Section 3.2.3.1. When an operator o is applied to a specification S_1, forward inference constructs a new specification which is identical to S_1 except that the negated postconditions are removed and the positive postconditions added.

Definition 4.2.4. *Given a state specification S_2 and contributing operator o we define reverse inference as the function to construct the state specification S_1 where:*

$$S_1 = \overleftarrow{Infer}(S_2, o) = (S_2 \backslash post(o)) \cup pre(o).$$

Given a specification S_2 and a contributing operator o we infer the preceding state specification from which o would be applied (S_1). Here, S_1 is S_2 with all postconditions removed, and all preconditions added. The intuition is that S_1 may, or may not contain any of the effects that are added through the application of o, but must satisfy all of o's preconditions. Note that the resulting specification is somewhat broad: it is impossible to know whether any postconditions of o held in the precursor specification, so we assume no knowledge of their state and simply remove them from the specification.

Definition 4.2.5. *Given a state specification S and a partially applicable operator o, a* forward refinement *of the state specification of S is defined as:*

$$\overrightarrow{Refine}(S, o) = S \cup pre(o).$$

Our set based specification representation becomes more restrictive as more literals are added. This forward refinement is stating that o is only applicable from a subset of states represented by S, specifically those that contain all preconditions of o.

Definition 4.2.6. *Given a state specification S and contributing operator o, we define* reverse refinement *as:*

$$\overleftarrow{Refine}(S, o) = S \cup (pre(o) \backslash \neg post(o)) \cup post(o)$$

The refined specification is identical to S but contains the preconditions of o that are not removed by o as well as the postconditions of o. That is, the application of o results in the refined specification that includes all effects of o ($post(o)$) as well as the preconditions of o that have not been removed by an effect ($pre(o) \backslash \neg post(o)$).

We now briefly illustrate how refinement always results in equally, or more specific state specifications. This is a useful result that we discuss in Section 4.2.5.1.

Lemma 4.2.7. *Both forward and reverse refinement of a specification S results in S', a restriction of S, where $Mod(S') \subseteq Mod(S)$.*

Proof. We handle the cases of forward and reverse refinement in turn through an analysis of the transition function R presented in Section 3.2.3.1:

- Let $S' = \overrightarrow{Refine}(S, o)$. Since we only add literals in $pre(o)$ to the specification S to form S' it follows that $S' \supseteq S$.
- Let $S' = \overleftarrow{Refine}(S, o)$. Since we only add the literals $post(o)$ and $pre(o) \setminus \neg post(o)$ to S it follows that that $S' \supseteq S$.

The state specification generated through forward or reverse refinement is therefore at least as specific as the original specification. ■

The process of refinement only ever results in state specifications that are at least as specific as the original, but usually are more restrictive.

Example Consider the specification $S = \{\texttt{at(a}_1\texttt{,node}_1\texttt{)}\}$. We will now illustrate the steps of inference and refinement using example contributing and partially applicable operators. Let $o_p \in O_{cont}(S)$ be the contributing operator $\texttt{move(a}_1\texttt{,node}_3\texttt{,node}_1\texttt{)}$. It follows that o_p can lead to some of the states represented by S. We use reverse refinement to identify this subset S^1 and reverse inference to create the precursor specification S^1_P:

$$S^1 = \overleftarrow{Refine}(S, o_p) = \{\texttt{at(a}_1\texttt{,node}_1\texttt{)}, \texttt{conn(node}_3\texttt{,node}_1\texttt{)}, \neg\texttt{at(a}_1\texttt{,node}_3\texttt{)}\}$$

$$S^1_P = \overleftarrow{Infer}(S^1, o_p) = \{\texttt{at(a}_1\texttt{,node}_3\texttt{)}, \texttt{conn(node}_3\texttt{,node}_1\texttt{)}, \}$$

We can now initialise our consistent run: $S^1_P \xrightarrow{\texttt{o}_\texttt{p}} S^1$. Similarly, it is possible to use forward refinement and inference to construct a run originating in S. Let $o_s \in O_{par}$ be the partially applicable operator $\texttt{drop(a}_1\texttt{,node}_1\texttt{,parcel}_1\texttt{)}$. We again show the refinement of S to S^2, and infer the successor specification S^2_S:

$$S^2 = \overrightarrow{Refine}(S, o_s) = \{\ \texttt{at(a}_1\texttt{,node}_1\texttt{)},\ \texttt{hold(a}_1\texttt{,parcel}_1\texttt{)}\ \}$$

$$S^2_S = \overrightarrow{Infer}(S^2, o_s) = \left\{ \begin{array}{c} \texttt{at(a}_1\texttt{,node}_1\texttt{)},\ \neg\texttt{hold(a}_1\texttt{,parcel}_1\texttt{)}, \\ \texttt{parcelAt(parcel}_1\texttt{,node}_1\texttt{)} \end{array} \right\}$$

The consistent run $S^2 \xrightarrow{\texttt{o}_\texttt{s}} S^2_S$ originating from a subset of the states represented by S can now be initialised. □

4.2.3.2 Run Refinement

We define how refinement can be consistently applied to runs of state specifications using our propositional formalism, first by introducing a definition of consistency, and following this with the run refinement itself.

Definition 4.2.8. *Given a consistent specification S and a consistent set of literals L_+, the resulting specification $S' = S \cup L_+$ is termed* consistent *if:*

$$\forall l \in L_+ : \neg l \notin S.$$

Consider a run $R = S_1 \xrightarrow{\mathtt{o_1}} \ldots \xrightarrow{\mathtt{o_{n-1}}} S_n$. Suppose we refine S_n for some successor operator o. We construct a new successor specification S'_n where $S'_n = \overrightarrow{Refine}(S_n, o)$. Let L_+ be the set of new literals added to the specification S_n such that:

$$L_+ = S'_n \backslash S_n.$$

Under certain conditions the literals we add may conflict with literals added in previous specifications in the run. L_+ is consistent with the run R if:

- L_+ is consistent with each of the state specification $S_1 \ldots S_n$, and
- $\forall l \in L_+, \forall i \leq n-1 : \neg l \notin post(o_i)$.

The key notion here is to consider the set L_+ with respect to each specification in the run. We know that literals in L_+, or their negations, do not appear in S_n since S_n is consistent with L_+ by definition. Since neither the literals in L_+ nor their negations appear in S_n it follows that they are not referenced at any point previously in the run. If a previous operator added a literal that is present in L_+, then either the literal or its negation must appear in S_n since literals cannot be removed from a specification. Once a literal is introduced by an operator, the literal or its negation will appear in every subsequent specification of the run. Since none of the literals in L_+ appear beforehand we know that none of the prior specifications make reference to these atoms, and that the literals must hold in every precursor specification. If we refine S_n and include the literals in L_+ then we must refine every specification in the run accordingly.

Definition 4.2.9. *Given a run $R = S_1 \xrightarrow{\mathtt{o_1}} \ldots \xrightarrow{\mathtt{o_{n-1}}} S_n$ and a successor partially applicable operator o. Let the literals added to S_n during refinement be L_+ as defined above. The* refined run *R' from which o is fully applicable is defined as:*

$$R' = (S_1 \cup L_+) \xrightarrow{\mathtt{o_1}} (S_2 \cup L_+) \ldots (S_{n-1} \cup L_+) \xrightarrow{\mathtt{o_{n-1}}} (S_n \cup L_+).$$

Example Let $S_C = \{\mathtt{at(a_1,node_1)}\}$. A possible partial run of the form $S_P \xrightarrow{\mathtt{o_P}} S_S$ is:

$$\left\{ \begin{array}{c} \mathtt{at(a_1,node_3)}, \\ \mathtt{conn(node_3,node_1)} \end{array} \right\} \xrightarrow{\mathtt{move(a_1,node_3,node_1)}} \left\{ \begin{array}{c} \mathtt{at(a_1,node_1)}, \\ \mathtt{conn(node_3,node_1)}, \\ \neg\mathtt{at(a_1,node_3)} \end{array} \right\}.$$

Now let's consider $\texttt{drop(a}_1\texttt{,node}_1\texttt{,parcel}_1\texttt{)}$ a successor operator o_s to be appended to this run. We refine S_S to S'_S by adding the literal set $L_+ = \{\texttt{hold(a}_1\texttt{,parcel1)}\}$ so that the operator o_s is applicable in all states specified by S'_S. Run refinement states that if the resulting specification (S'_S) is consistent, then the remaining specifications in the run must also be refined. The new, refined run is:

$$\left\{\begin{array}{c}\texttt{at(a}_1\texttt{,node}_3\texttt{),}\\ \texttt{conn(node}_3\texttt{,node}_1\texttt{),}\\ \texttt{hold(a}_1\texttt{,parcel1)}\end{array}\right\}\xrightarrow{\texttt{move(...)}}\left\{\begin{array}{c}\texttt{at(a}_1\texttt{,node}_1\texttt{),}\\ \texttt{conn(node}_3\texttt{,node}_1\texttt{),}\\ \neg\texttt{at(a}_1\texttt{,node}_3\texttt{),}\\ \texttt{hold(a}_1\texttt{,parcel1)}\end{array}\right\}\xrightarrow{\texttt{drop(a}_1\texttt{,node}_1\texttt{,parcel}_1\texttt{)}}$$

□

With these definitions we complete the presentation of propositional conflict traversal. The next step of the synthesis algorithm, Norm Synthesis, follows directly as before. We now conclude with our presentation of the Reachability Analysis stage.

4.2.4 Propositional Reachability Analysis

Reachability analysis is composed of two core steps: norm application transforms the domain so that actions prohibited by the norms are no longer applicable and reachability checking constructs a planning problem to verify that alternative conflict-free plans exist for each complete run. We begin by presenting an extension to the planning formalism that compiles a given set of prohibitionary norms into the operator schemata to accomplish norm application.

4.2.4.1 A Normative Planning Extension

We define what constitutes norm-respecting behaviour within a planning-based multiagent system. Typically, planning agents have no explicit representation of social norms and plans might therefore violate the social norms. In such situations, there is no notion of a norm-respecting plan, or norm-respecting behaviour. With this in mind, we extend the presented propositional planning formalism to incorporate an explicit representation of social norms.

Definition 4.2.10. *A* normative planning problem *is an extended planning problem presented as a tuple:*

$$\Pi_N = \langle \Xi, S_I, S_G, \mathcal{N} \rangle$$

where Ξ, S_I and S_G are the domain structure, initial state specification and goal state specification respectively, and $\mathcal{N}$ are a set of prohibitionary norms.

Any application of operators in this formalism is conditional on the set of prohibitions. In this context, a solution to a normative planning problem must not contain any actions that are prohibited by the set of norms. Formally, we can define a prohibition function over a state specification as $F : 2^{\hat{\Sigma}} \times O \times 2^{\mathcal{N}} \rightarrow \{\top, \bot\}$ such that:

$$F(S,o,\mathcal{N}) = \begin{cases} \top & \text{if } \exists \langle \varphi, o' \rangle \in \mathcal{N} : (\varphi \models S) \wedge (o' = o) \\ \bot & \text{otherwise.} \end{cases}$$

An operator o is forbidden for a state specification S under norms $\mathcal{N}$ if there exists a norm prohibiting o with precondition modelled by S. This prohibition function is used to extend the state transition function presented in Section 3.2.3.1 as follows:

$$R(S,o,\mathcal{N}) = \begin{cases} R(S,o) & \text{if } F(S,o,\mathcal{N}) = \bot \\ \bot & \text{otherwise.} \end{cases}$$

A *solution* to the normative planning problem $\Delta_N = \langle o_1, o_2, \ldots \rangle$ is a set of operators that, if applied to the initial state specification S_I, will result in a state that satisfies the goal state specification S_G without violating any of the social norms in $\mathcal{N}$.

4.2.4.2 Operator Specification Rewriting

A simple, static implementation of the prohibitionary norm set in a given domain can be accomplished through an operator specification rewrite procedure, allowing us to use off the shelf planners to compute a solution. We transform the task of planning with norms into a simple classical planning problem.

We begin by rewriting each operator in turn. For each $o \in O$ we rewrite the preconditions of the operator as a conjunction of the conditions of the norms that reference the operator. The subset of norms that reference the operator o can be written as $\mathcal{N}_o = \{n | n \in \mathcal{N} \text{ and } o(n) = o\}$. The transformed precondition pre' for operator o follows:

$$pre'(o) = pre(o) \wedge \Big(\bigwedge_{n \in N_o} \neg\varphi(n) \Big).$$

Since we use a set notation for $pre(n)$ it should be emphasised that the Boolean expression resulting from this transformation will be formed by taking the conjunction of each of the literals in $pre(n)$.

Example Let $S_C = \{\mathtt{at(a_1,node_1)}, \mathtt{at(a_2,node_1)}\}$ and $n = \langle \varphi, o \rangle$ be a norm that partially enforces this where:

- $\varphi = \{\mathtt{at(a_1,node_3)}, \mathtt{at(a_2,node_1)}, \mathtt{conn(node_3,node_1)}\}$

- $o = \texttt{move}(\texttt{a}_1,\texttt{node}_3,\texttt{node}_1)$.

Here n prohibits Agent $\texttt{a}_1$ from moving from $\texttt{node}_3$ to $\texttt{node}_1$ if Agent $\texttt{a}_2$ is in $\texttt{node}_1$. Using our rewrite procedure, we would reform the preconditions of o as follows:

$$pre'(o) = pre(o) \wedge \Big(\neg\big(\texttt{at}(\texttt{a}_1,\texttt{node}_3) \wedge \texttt{at}(\texttt{a}_2,\texttt{node}_1) \wedge \texttt{conn}(\texttt{node}_3,\texttt{node}_1)\big)\Big)$$

where $pre(o) = \texttt{at}(\texttt{a}_1,\texttt{node}_3) \wedge \texttt{conn}(\texttt{node}_3,\texttt{node}_1)$. Here $\texttt{move}(\texttt{a}_1,\texttt{node}_3,\texttt{node}_1)$ can be applied in a state if its preconditions are satisfied, and if it is not the case that Agent $\texttt{a}_1$ will move into $\texttt{node}_1$ while Agent $\texttt{a}_2$ occupies this location. $\square$

The remainder of the reachability analysis and checking follows directly from the algorithm presented in Section 4.1.3.3.

4.2.5 Algorithm Properties

We present discussions and proofs related to the properties of termination and complexity of the algorithm followed by an argument for completeness and correctness.

4.2.5.1 Termination

Theorem 4.2.11. *The conflict-rooted synthesis bound to the propositional planning formalism is guaranteed to terminate.*

We show termination of the conflict-rooted synthesis algorithm by independently considering termination of the traversal, synthesis and reachability steps.

Lemma 4.2.12. *The traversal procedure of the propositional conflict-rooted synthesis algorithm always terminates.*

Proof. We begin by stating the following assumptions according to definitions in our planning formalism:

1. The set of operators O is finite. As a result, so too are the subsets O_{cont} and O_{par}.
2. The set of atoms Σ is finite. As a result, the set of literals $\hat{\Sigma}$ is also finite.
3. The inference, refinement and run-based operations always terminate.

We present the proof of termination through a loop analysis of Algorithm 1. From assumption (1) any iteration over precursor operators (line 3) and successor operators (line 14) is bound. As a result the algorithm terminates once the set of incomplete runs $\mathcal{U}$ is empty ($\neg|\mathcal{U}| > 0$).

We now show the set of possible incomplete runs to be finite. Let $\mathscr{R}$ be the set of all possible runs composed over the set of all possible state specifications $2^{\hat{\Sigma}}$, using operators from O. By assumption both O and $2^{\hat{\Sigma}}$ are finite. The set $\mathscr{R}$ however is not finite. To illustrate this consider an element of $\mathscr{R}$ to be the run R of arbitrary length:

$$R = S_1 \xrightarrow{o_1} S_2 \xrightarrow{o_2} S_1 \xrightarrow{o_1} S_2 \dots$$

It is possible to construct an infinite sequence of runs simply by extending the sequence, resulting in an infinite member of the set $\mathscr{R}$.

Let the subset of runs $\hat{\mathscr{R}} \subseteq \mathscr{R}$ be a set of runs that do not contain loops. A loop is present if any two specifications in the run are identical. Formally:

$$\hat{\mathscr{R}} = \Big\{ R \mid R \in \mathscr{R} \text{ where } \forall i \nexists j \; (R[i] = R[j] \land i \neq j) \Big\}.$$

The set $\hat{\mathscr{R}}$ is finite since the sets of all specifications and operators are finite.

Since conflict traversal does not consider runs that contain loops (line 20) and each iteration of the traversal (via line 19) adds runs that are strictly longer than their predecessor, we know that the set of all possible incomplete runs is finite and that no run is considered twice. The conflict traversal is therefore guaranteed to terminate. ■

Lemma 4.2.13. *The norm synthesis procedure of the conflict-rooted synthesis algorithm with the propositional planning binding always terminates.*

Proof. From Lemma 4.2.12 it follows that the number of runs produced during traversal is finite, and that the monotonicity of the traversal process guarantees that runs are only considered once. The set $\mathcal{R}_{\llcorner}$ of complete conflict-free runs is finite, and as a result the norm synthesis process terminates. ■

Finally, we show termination of the reachability analysis procedure. Since we outsource the solving of planning problems to an automated planner we emphasise that the termination result is dependent on the ability of the planner to complete. Fortunately, modern day planners that perform heuristic hill climbing search will fall back onto breadth first searches when required, meaning that they too are guaranteed to terminate given sufficient resources. We assume, for this lemma, that such a planner is in use. Importantly, we acknowledge that in practice planning may take an excessive amount of time, or may never complete due to a lack of resources.

Lemma 4.2.14. *The propositional reachability analysis procedure always terminates.*

Proof. The set of complete runs $\mathcal{R}_{\mathcal{C}}$ and the set of candidate norms $\mathcal{N}_{\mathcal{C}}$ are both finite. Termination of the reachability analysis algorithm holds since:

1. the norm application method (*NormApplication* on line 2) implemented via operator transformations (as defined in Section 4.2.4.2) terminates since it is dependent on the number of norms and operators, both of which are finite.

2. the reachability check terminates since the number of complete runs is finite (lines 7 and 8) and by assumption the planner invocation, defined via the method *InvokePlanner* call, will terminate.

Since each of the components of the reachability analysis terminates we show this lemma proved. ■

The proof of Theorem 4.2.11 follows from Lemmas 4.2.12, 4.2.13 and 4.2.14.

4.2.5.2 Complexity

Recall the set of operators to be O and the set of propositional atoms in the domain be Σ. Using closed world semantics, a state is represented by some subset of atoms, leading to a maximum number of $2^{|\Sigma|}$ system states. The state specification space is larger with $3^{|\Sigma|}$ possible specifications, since specifications contain literals rather than atoms. The worst case complexity of each of the conflict-rooted synthesis steps follow:

- **Conflict Traversal**: The length of runs found during traversal are bound above by the number of specifications, $3^{|\Sigma|}$. At each step of the conflict traversal, the algorithm chooses at most $|O|$ operators as successors for each run, producing a worst case computational complexity upper bound of $\mathbb{O}(|O|^{3^{|\Sigma|}})$[1]. If duplicate runs are discarded, the total number of runs produced by traversal is bound above by $|\mathcal{R}_{\mathcal{C}}| = 3^{2|\Sigma|}|O|$, representing that a unique run is composed of a contributing operator ($|O|$) and first and last state specifications ($3^{|\Sigma|}3^{|\Sigma|}$).

- **Norm Synthesis**: The synthesis step is linear in the number of runs produced and is therefore $\mathbb{O}(3^{2|\Sigma|}|O|)$.

- **Reachability Analysis**: For each run, a planner is invoked. In the worst case, the planner will search the same space as performed during traversal for each

[1]We write $\mathbb{O}(\ldots)$ for Big O notation that characterises the upper bound for the growth rate of a function, in order to differentiate it from the set of operators O.

run produced, resulting in a total complexity of $\mathbb{O}(3^{2|\Sigma|}\ |O|^{3^{|\Sigma|}+1})$.

A large portion of the resulting complexity of conflict-rooted synthesis is dependent on the complexity of plan synthesis. In the general case the decision problem of plan existence is PSPACE-complete. Consider that if there are n propositional atoms, then the length of the shortest solution to a planning problem must be less than or equal to 2^n. All longer solutions contain loops, implying that in order to find the solution no more than 2^n nondeterministic operator choices are required. Although the length of a solution plan is exponential, an algorithm need only polynomial space in order to find a solution plan (Bylander, 1994). Since the plan existence problem is in NPSPACE it is also in PSPACE. In time conflict-rooted synthesis is NP-complete, since an exponential amount of computation is required to synthesise runs, and alternative plans for each of the runs, yet the results can be verified in polynomial time. Space complexity is worse, since the number of runs produced during traversal is exponential in the number of atoms in the domain, implying a worst case complexity of EXPSPACE. If runs are not remembered, the complexity is PSPACE.

Much of the complexity is involved in the planner invocations during reachability analysis. While propositional planning is PSPACE-complete, restrictions on the domain are able to simplify the problem. For example, with no negative effects allowed the problem is NP-complete and with no negative preconditions it is in P.

4.2.5.3 Soundness and Completeness

We present our statement of soundness in the following theorem and prove it using the two subsequent lemmas.

Theorem 4.2.15. *The propositional conflict-rooted synthesis algorithm is sound.*

We show soundness in two parts by proving:

1. The norms synthesised are guaranteed to prohibit access to all conflict states from conflict-free states.
2. A successful reachability check guarantees that a conflict-free alternative sequence of actions exists for every complete run, and thereby ensures that previously accessible conflict-free states remain accessible.

Our proof of correctness also shows completeness: the norms prohibit all conflict-states and under no circumstances is a norm not found, if one exists.

Lemma 4.2.16. *The norms synthesised using propositional conflict-rooted synthesis prohibit access to all conflict states.*

Proof. We adopt a proof by contradiction. Let $\mathcal{N}$ represent the set of synthesised norms for a conflict specification S_C. We begin by assuming that there exists a simple incomplete run $R^* = S_P^* \xrightarrow{o^*} S_C^*$ where S_P^* is conflict-free precursor specification, and S_C^* is a conflict specification, and where R^* is not prohibited by any norms in $\mathcal{N}$.

We prove our lemma by showing that the run R^* cannot exist since the operator o^* is always considered as a contributing operator, and that the inferred specification S_P^* must be accounted for by some norm condition.

First, we show that we always consider the operator o^* in our set of contributing operators O_{cont}. For o^* to not be considered as a contributing action at least one of the following two requirements (presented in Section 4.2.3) must hold:

1. $\nexists l \in post(o^*)$ where (i) $l \in S_C$ and (ii) $l \in pre(o^*)$. That is, no effect l of o^* exists that contributes to S_C and is not a precondition of o^* already contained in S_P^*.
2. There exists a literal effect $l \in \big((pre(o^*) \setminus \neg post(o^*)) \cup post(o^*)\big)$ where $\neg l \in S_C^*$. This cannot hold, since if $\neg l \in S_C^*$ and l is an effect of the operator o^* then the resulting set S_C^* is inconsistent.

Therefore, we know that the operator o^* must be in the set of contributing operators. Secondly, if R^* is not accounted for by some norm it must be the case that the inferred specification S_P^* is not modelled by at least one norm condition. Given conflict specification S_C and the identified contributing operator o^* we know that the inferred precursor state is defined as (from Section 4.2.3):

$$S_P = (S_C \setminus post(o^*)) \cup pre(o^*).$$

Since S_P is used as a norm condition, for R^* to not be considered during synthesis it must hold that $S_P \not\models S_P^*$. That is, the condition that we synthesise for our norm to prohibit o^* does not include the provided precursor S_P^*. However, we know:

1. $pre(o^*) \subseteq S_P^*$ holds since the operator o^* must be applicable in S_P^*, and
2. $(S_C \setminus post(o^*)) \subseteq S_P^*$ holds since all literals in S_C that were not added by an effect of o^* must be part of the precursor specification.

It cannot be the case that the norm does not have the correct condition to prohibit the run R^*. We have reached a contradiction. ■

Lemma 4.2.17. *If reachability is shown successfully then a conflict-free alternative sequence of actions exists in the normative system for every conflict sequence in the original.*

Proof. We adopt a proof by contradiction. Assume that we have shown reachability for all conflict runs, but that there exists a complete run R^* which is not reachable in the underlying system. From Lemma 4.2.16 we know that the traversal considered this run (since it is a complete run) and synthesised a sound norm. For reachability to succeed, we know that we found a sound plan Δ^* which, when applied to the specification in $\mathit{first}(R^*)$ resulted in $\mathit{last}(R^*)$ without traversing any conflict states. Yet if a sound plan exists in the normative system then these actions can be used as a conflict-free alternative to R^*, implying that R^* is indeed reachable. We reach a contradiction. ■

From Lemmas 4.2.16 and 4.2.17 we show that Theorem 4.2.15 holds and that conflict-rooted synthesis is sound. A completeness result follows from Lemma 4.2.16 if the planner is complete. There are a number of sound and complete planners for the domain formalism we are using that will guarantee a correct result, such as FF which we utilise in our evaluation. If a complete planner is used then the completeness of conflict-rooted synthesis follows.

4.2.6 Limitations

There are three core limitations to a propositional approach to norm synthesis:

1. The expressivity of conflict state specifications is limited by the planning formalism. It is not possible to use variables in conflict state specifications to quantify over states in the underlying propositional domain.

2. Limitations in the applicability of the approach are inherited from the limited expressiveness of the planning formalism. Since propositional domains are verbose, it is common that they are not utilised in practice. This provides a limitation to the applicability of our norm synthesis approach.

3. We cannot generate independent norms. Instead, the norms are specific to the input domain due to the lack of operator abstractions.

Example We have been considering conflict specifications where Agents $\mathtt{a}_1$ and $\mathtt{a}_2$ should not occupy $\mathtt{node}_1$ simultaneously. Consider the conflict specification where the two agents are to not occupy *any* nodes simultaneously. An enumeration is required to describe collisions in each node:

$$\{\mathtt{at}(\mathtt{a}_1,\mathtt{node}_1),\mathtt{at}(\mathtt{a}_2,\mathtt{node}_1)\}$$
$$\{\mathtt{at}(\mathtt{a}_1,\mathtt{node}_2),\mathtt{at}(\mathtt{a}_2,\mathtt{node}_2)\}$$
$$\{\mathtt{at}(\mathtt{a}_1,\mathtt{node}_3),\mathtt{at}(\mathtt{a}_2,\mathtt{node}_3)\}$$
$$\ldots$$

This enumeration is tied to the topology of the grid and is not feasible for large systems or complicated social objectives. An unbounded quantification over the nodes in the graph results in a single specification that is more expressive, and that applies independent of the topology of the grid. In this way we not only generate more succinct norms, but also norms that can be applied to different instances of the domain. □

Increases in expressivity often introduce additional complexity. A classical planning formalism uses parameterisation to generalise actions, resulting in schemata that are independent of fine details of the domain. While the propositional domain contains a large amount of knowledge encoded implicitly in the operator schemata, a classical representation does not. Synthesising norms in this more expressive formalism introduces new challenges. We extend the conflict-rooted synthesis algorithm to the classical planning formalism next.

4.3 Classical Planning Synthesis

A classical planning representation affords a significant increase in the expressivity of the planning domain, and we have argued previously that this extension is fundamental to the usability of conflict-rooted synthesis. A benefit of a first order state formalism is an increase in expressivity of conflict specifications, where variable symbols can be included with ground literals, allowing for specifications quantified over all values of these variables.

Unground specifications of states and actions expose a limitation in our existing propositional conflict traversal algorithm. When applying parameterised operators in a state specification planners require the state specification to be *ground* and that all operator parameters should be unified with a ground literal in the specification. The grounded nature of the state specifications is preserved as operators are applied since operators never introduce variables into the state representation. By allowing unground specifications we deviate from this traditional notion of a ground state specification.

Consider the process of refining a specification as described before. In a classical domain, instead of adding ground literals to the specification, we may introduce unground literals too. The ground nature of a specification is not necessarily preserved during refinement, and an appropriate state representation is required to embody this.

Example Let $S_C = \{\mathtt{at(a_1,node_1)}\}$. We show that, even if we enforce S_C to be ground, a formalism that allows unground specifications is required. Consider the unground operator $o = \mathtt{pickup(a_1,X,node_1)}$ where we write the unbound variable $\mathtt{X}$ to represent an arbitrary parcel. Here S'_C, the refined instance of S_C, contains an unground literal:

$$S'_C = \{\ \mathtt{at(a_1,node_1)},\ \mathtt{parcelAt(X,node_1)}\ \}$$

The ground nature of our state specification is not preserved during refinement due to the abstract nature of conflict specifications. □

We present classical conflict-rooted synthesis as an extension of our propositional approach with more expressive, unground state specifications. This requires an extension of the classical planning representation presented in Section 3.3. Our synthesis approach still assumes a classical planning domain representation as input and is still sound for these domains. We are not proposing a new representation for the input domains. Instead, to process these domains we require a more expressive representation to use within our algorithm. This allows for unground conflict specifications as input to the algorithm but does not preclude synthesis on standard classical planning domains.

4.3.1 Classical Parcel Delivery Domain

A classical planning representation allows us to introduce the notion of predicates and parameterised operators into our Parcel Delivery notation. For consistency, we use a notation that is very similar to the propositional case. We write $\mathtt{at(Agent,Node)}$ to represent the location of an agent, where $\mathtt{Agent}$ and $\mathtt{Node}$ are variables. Similarly, we write $\mathtt{parcelAt(Parcel,Node)}$ to represent the location of a parcel. The predicate $\mathtt{hold(Agent,Parcel)}$ indicates that $\mathtt{Agent}$ is holding a $\mathtt{Parcel}$. Finally, the adjacency of nodes is represented by a connectivity predicate $\mathtt{conn(Node_1,Node_2)}$. We now present the parameterised operator schemata that we will use in our examples:

OPERATOR:	$\mathtt{move(Agent,From,To)}$
PRE:	$\{\mathtt{at(Agent,From)},\mathtt{conn(From,To)}\}$
POST:	$\{\neg\mathtt{at(Agent,From)},\mathtt{at(Agent,To)}\}$

OPERATOR:	drop(Agent,Node,Parcel)
PRE:	{at(Agent,Node),hold(Agent,Parcel)}
POST:	{¬hold(Agent,Parcel),parcelAt(Parcel,Node)}

OPERATOR:	pickup(Agent,Node,Parcel)
PRE:	{at(Agent,Node),parcelAt(Parcel,Node)}
POST:	{¬parcelAt(Parcel,Node),hold(Agent,Parcel)}

4.3.2 An Unground Planning Extension

Recall from Section 3.3.1 that a specification in the classical sense is defined over the set of ground literals $L_{\overline{\mathcal{A}}}$. We extend this definition by generalising the definition of a state specification as a subset of the general literal set $\mathcal{A}$ as follows:

$$S \subseteq L_{\mathcal{A}} \text{ , where } L_{\mathcal{A}} = \{a, \neg a | a \in \mathcal{A}\} \cup \top \cup \bot.$$

This differs from the previous definition even though we have switched only from grounded to ungrounded literals. A *ground* specification $\overline{S}$ is defined as:

$$\overline{S} \subseteq L_{\overline{\mathcal{A}}} \text{ , where } L_{\overline{\mathcal{A}}} = \{a, \neg a | a \in \overline{\mathcal{A}}\} \cup \top \cup \bot.$$

Now consider whether an instance of an operator o is applicable in a specification S. Since S is unground we require a means of unifying unground literals. Our typical substitution method σ only allows for constants to be substituted in for variables. If o operates on one of the unground literals in S a unification between the variable parameter and variable literal is required. We illustrate this in the following example.

Example Consider the specification $S = \{\mathtt{at(A,node_1)}, \mathtt{conn(node_1,node_3)}\}$, depicting a Parcel World where an unspecified agent, represented by variable A, is in $\mathtt{node_1}$. In order to reason about A moving to $\mathtt{node_3}$ we must unify the parameters of the operator move(Agent,From,To) with S. In this example, a substitution is not sufficient, since the unification (Agent $\leftarrow$ A) is required along with the standard substitutions (From $\leftarrow$ $\mathtt{node_1}$) and (To $\leftarrow$ $\mathtt{node_3}$). □

We require a more expressive substitution function that not only allows constant substitutions for variables, but also variable unifications for variables. We call this functional superset a *substitution binding* and define the function ς as an extension of σ:

$$\varsigma \subseteq \{(v \leftarrow c) \mid v \in \mathcal{L}_v, c \in \mathcal{L}_c\} \cup \{(v \leftarrow v') \mid v, v' \in \mathcal{L}_v\}.$$

Constant terms c can be substituted for variables v, as in $(v \leftarrow c)$, and also variables v' to be unified with v, as in $(v \leftarrow v')$. As before we denote the bound instance of an operator o as $\varsigma[o]$.

Given a state s and an unground specification S we define $s \models S$ to represent that s is included in the set of states that model S. Since S now may contain variable terms we extend satisfiability for this more expressive state representation as:

- Given an unground specification S and a ground state s we say $s \models S$ if $\exists\sigma$ where $s \models \sigma[S]$. If a substituted instance of S exists that is modelled by s, then s models S.
- Given unground specifications S_1 and S_2 we say that $S_1 \models S_2$ if $\exists\varsigma$ where $\varsigma[S_2] \subseteq S_1$. That is, if a bound instance of S_2 is modelled by S_1, then S_1 models S_2.

Operator applicability in an unground specification follows virtually identically from the classical definition of applicability, except that instead of identifying a substitution we search for any binding.

Definition 4.3.1. *Given an unground specification S and an operator o, the applicability function App which determines whether an instance of o exists that is applicable in S is defined as:*

$$App(S,o) = \begin{cases} \top & \textit{if } \exists\varsigma \textit{ where } pre(\varsigma[o]) \subseteq S \\ \bot & \textit{otherwise} \end{cases}$$

An operator is applicable in S if a bound instance of the operator is applicable in S. Note that this notion of applicability can easily be mapped to the standard classical definition, and also to our propositional definition. In a classical domain we know that S would be ground, so instead of ς we use σ to define substitutions of constants that ground o. The propositional case is even simpler: since o is effectively ground no substitution is required, and applicability is determined simply if $pre(o) \subseteq S$. The final component of our planning formalism extension is to define the transition function.

Definition 4.3.2. *Given an unground specification S, an operator schema o and an extended substitution ς, we define the transition between state specifications as:*

$$R(S,o,\varsigma) = \begin{cases} \big(S \setminus \neg post(\varsigma[o])\big) \cup post(\varsigma[o]) & \textit{if } (S \not\models \bot) \wedge (S \models pre(\varsigma[o])) \wedge \\ & post(\varsigma[o]) \not\models \bot \\ \bot & \textit{otherwise.} \end{cases}$$

This transition function is very similar to the propositional definition in Section 3.2.3.1 where we consider a bound instance of o rather than a substituted. The bindings here are key to reasoning about the effect of an operator as they map the abstract schema o to the variables and predicates present in S. Importantly, the sets of literal postconditions that are removed and added can be ungrounded. As shown above, we can migrate from this more general definition to standard classical and propositional definitions.

We are now able to represent specifications of sets of states that contain variable symbols and reason about operator applicability and operator effects with these sets. Classical conflict-rooted synthesis follows next.

4.3.3 Unground Norms

Our norms can now utilise unground specifications in their representation. A prohibitionary norm is still a tuple $n_i = \langle \varphi, o \rangle$ where:

- $\varphi \subseteq L_{\mathfrak{A}}$, and
- $o \in O$.

The norm condition is no longer a set of propositional atoms but rather a specification over the unground atoms.

4.3.4 Conflict Traversal

For brevity we will not reintroduce each definition but only the components that have changed from the propositional approach. Synthesising in our classical representation follows almost identically from the propositional with one key difference: instead of considering propositional, grounded operator schema we consider ungrounded operator representations. In general we cannot process abstract operator representations directly so we utilise substitution bindings to instantiate the operator schemata into a representation that can be applied to the state specifications directly. As such, many of the definitions remain as before but refer to bound instance ($\varsigma[o]$) of operators.

An operator o is applicable in a propositional state specification S if $S \models pre(o)$. Now, if S is unground and o is parameterised, then simply checking whether the operator's preconditions appear in the specification is insufficient: the operator must be instantiated with substitutions for the operator parameters before applicability is checked. Notice that if $\overline{o} = \varsigma[o]$ is ground then applicability follows almost identically as before ($S \models pre(\overline{o})$). We summarise each of the critical operator definitions below:

- An operator o is *applicable* in a specification $S \subseteq L_{\mathfrak{A}}$ under binding ς if

$$S \models pre(\varsigma[o]).$$

- An operator o is *partially applicable* in a specification $S \subseteq L_{\mathfrak{A}}$ under binding ς if

$$\forall l \in S : \nexists l' \in pre(\varsigma[o]) \text{ such that } (l = \neg l' \vee l' = \neg l).$$

- An operator *o contributes* to a specification $S \subseteq L_{\mathfrak{A}}$ under binding ς if

$$\exists l \in post(\varsigma[o]) \text{ where } \big(l \in S \wedge l \notin pre(\varsigma[o])\big)$$
$$\text{and}$$
$$\nexists l \in \big((pre(\varsigma[o]) \setminus \neg post(\varsigma[o])) \cup post(\varsigma[o])\big) \text{ where } \neg l \in S.$$

Computing the operator sets $O_{cont}(S)$ and $O_{par}(S)$ in this extended representation requires some discussion since operators are now parameterised, abstractions of action. Instead of identifying a subset of O we are now interested in the subset of the *instantiated* operators: operators with bindings for their parameters. There are two problems that require solving before we can apply our definitions of $O_{cont}(S)$ and $O_{par}(S)$ in the context of our classical planning formalism:

1. **Binding Predicates**: We require a means to generate the set of all operator bindings, since given a binding we are able to classify operators appropriately. This is an extension of the propositional approach since we have not yet dealt with variable assignments in parameterised operators.

2. **Constraints**: Given all possible bindings we can then generate sets of instantiated operators, yet these operators may contain unground variables if the binding is incomplete. A means to differentiate unground operators from more complete grounded instances is required.

4.3.4.1 Binding Predicates

We adopt a simple approach to binding two sets of predicates and do not consider this a contribution of this work, yet we present the approach for completeness. Consider the specifications S_F and S_T where we wish to identify a binding from S_F to S_T: a ς such that $S_T = \varsigma[S_F]$. We present a naive algorithm to perform binding next and subsequently consider it the implementation of a function *BindingEnumeration*.

The algorithm recursively performs a full search of all possible bindings between predicates in S_F and S_T. The base case on line 2 continues until S_F is the empty set. On each

Algorithm 4: Creating the set of all bindings between two state specifications

```
Input: The specifications $S_F$ and $S_T$, the current binding $\varsigma = \emptyset$ and the set of all constructed
       binding $2^\varsigma = \emptyset$
Result: The set of all bindings $2^\varsigma$ is populated.
1  begin
       The recursion base case:  all literals have been considered
2      if $S_F$ is empty then
           Add the constructed binding to the set
3          $2^\varsigma \leftarrow 2^\varsigma \cup \varsigma$
4      else
           Consider each predicate literal in the set $S_F$
5          for each $l_F \in S_F$ do
6              $S'_F \leftarrow S_F \setminus l_F$
               Consider each predicate literal in the set $S_T$
7              for each $l_T \in S_T$ do
                   Check whether a valid mapping exists
8                  $\varsigma' \leftarrow BindLiterals(l_F, l_T)$
9                  if $\varsigma'$ is not empty then
                       $l_F$ and $l_T$ have been bound.  Remove $l_T$ from consideration
10                     $S'_T \leftarrow S_T \setminus l_T$
11                     call recursively with $(S'_F, S'_T, \varsigma \cup \varsigma', 2^\varsigma)$
               Consider the case where no binding exists for $l_F$
12             call recursively with $(S'_F, S_T, \varsigma, 2^\varsigma)$
13 end
```

recursive call we pass a subset of S_F on with the predicate that has been considered removed, thereby guaranteeing termination. If a binding is found between two literals then both literals are removed from consideration and the corresponding subsets are now searched for subsequent bindings. We recurse regardless of whether a match is found to return partial as well as maximal bindings, and the algorithm terminates with the set of all possible bindings.

The function *BindLiterals* is responsible for producing a binding if one exists between two predicates. Consider two predicates $p_1(x_1^1, \ldots, x_n^1)$ and $p_2(x_1^2, \ldots, x_m^2)$. We define the function *BindLiterals* as:

$$BindLiterals(p_1, p_2) = \begin{cases} \left\{(x_i^1 \leftarrow x_i^2) \mid \forall 1 \le i \le n\right\} & \text{if } p_1 = p_2 \text{ and } m = n \\ \emptyset & \text{otherwise.} \end{cases}$$

A binding exists between two predicates if they have the same predicate symbols, and if a binding exists between each of the parameters. We assume binding sets are checked for consistency where sets that bind constant symbols to conflicting constant symbols are considered inconsistent.

Our reworked implementations of $O_{cont}(S)$ and $O_{par}(S)$ follow in Section 4.3.4.3. By first generating all possible bindings, we can in turn instantiate the abstract operator schemata and use these sets of instantiated operators to compute the contributing and partially applicable sets as before.

Example Consider two state specifications:

$$\begin{aligned} S_F &= \{\ \mathtt{at}(\mathtt{a}_1,\mathtt{X}),\ \mathtt{conn}(\mathtt{X},\mathtt{node}_3)\ \} \\ S_T &= \{\ \mathtt{at}(\mathtt{A}_1,\mathtt{node}_2),\ \mathtt{conn}(\mathtt{node}_2,\mathtt{Y})\ \} \end{aligned}$$

The resulting binding that satisfies $S_T = \varsigma[S_F]$ is:

$$\varsigma = \{\ (\mathtt{A}_1 \leftarrow \mathtt{a}_1),\ (\mathtt{X} \leftarrow \mathtt{node}_2),\ (\mathtt{Y} \leftarrow \mathtt{node}_3)\ \}$$

For this example there is only one such binding, but in practice any number of bindings may occur. □

To summarise, our statements of contributing and partially applicable operators are identical to those in the propositional formalism if a binding ς is considered for each operator o. Here we have shown how the set of all possible bindings can be calculated. With this complete set of bindings we can construct the complete set of instantiated operators and can use our existing definitions to calculate the subsets of contributing and partially applicable operators.

4.3.4.2 Variable Constraints

During refinement in the classical formalism unground literals may be introduced into a state specification. We introduce *constraints* as explicit relationships between variable literals contained in a state specification.

Example Consider the specification $S = \{\mathtt{at}(\mathtt{a}_1,\mathtt{node}_1),\mathtt{conn}(\mathtt{node}_1,\mathtt{node}_2)\}$, and the operator schema $\mathtt{move}(\mathtt{Agent},\mathtt{From},\mathtt{To})$. One binding for this operator is:

$$\varsigma = \{\mathtt{Agent} \leftarrow \mathtt{a}_1, \mathtt{From} \leftarrow \mathtt{node}_1, \mathtt{To} \leftarrow \mathtt{node}_3\}.$$

Here $\mathtt{a}_1$ moves to $\mathtt{node}_3$, yet a side effect of our notion of partial applicability is that S might additionally represent states from where $\mathtt{a}_1$ moves from $\mathtt{node}_1$ to $\mathtt{node}_2$, $\mathtt{node}_4$

or any other node. This is possible since there is no literal in S that forbids this action, and we have no knowledge of the graph topology. Consider the reduced binding:

$$\varsigma' = \{\texttt{Agent} \leftarrow \texttt{a}_1, \texttt{From} \leftarrow \texttt{node}_1\}.$$

where we have intentionally left the To parameter unbound. If we consider $\varsigma[o]$ and $\varsigma'[o]$ as two different successor operators when constructing runs then we must ensure that the state spaces do not overlap. To distinguish between these two a constraint is introduced for ς' where the unbound variable To cannot be $\texttt{node}_3$. □

More generally, consider an operator o. Suppose we identify some binding ς where $\varsigma[o]$ is fully applicable in a specification S: $pre(\varsigma[o]) \subseteq S$. No refinement is required here since all preconditions have been satisfied. However, consider a new binding $\varsigma' \subset \varsigma$ where $pre(\varsigma[o]) \nsubseteq S$ and refinement is required. Without loss of generality, let $\varsigma = \varsigma' \cup (v_1 \leftarrow v_2)$. The set of operators represented by $\varsigma'[o]$ is a superset of those represented by $\varsigma[o]$. If we wish to create sets that are mutually exclusive then the constraint $(v_1 \not\leftarrow v_2)$ is required for ς'. That is, v_1 can take any value other than v_2, for if it were to take v_2 then ς' would be identical to ς. We define a constraint set κ as:

$$\kappa \subseteq \big\{(v \not\leftarrow c) \mid v \in \mathcal{L}_v, c \in \mathcal{L}_c\big\} \cup \big\{(v \not\leftarrow v') \mid v, v' \in \mathcal{L}_v\big\}.$$

A constraint set contains pairs of symbols that cannot be bound together. Constraints are important to conflict traversal for two reasons:

1. Constraints ensure that unground successor operators are mutually exclusive, since constraints over the variable symbols preclude different unground operators from representing the same ground operator.

2. Constraints restrict which underlying states are represented by a state specification, which simplifies the process of grounding the specification as the space of possible variable assignments is smaller.

4.3.4.3 Conflict Traversal in a Classical Formalism

The core traversal algorithm remains unchanged from Algorithm 1. Instead, we redefine the functions O_{cont} and O_{par} to return substituted instances of the abstract operators contained in O. Recall that the results of these functions are sets of operators in O that have been substituted with some binding. We investigate all possible bindings of parameters for each operator in O. Since not all bindings are maximal, it is the case that certain bindings are supersets of others. To ensure that the operators modelled with these bindings are mutually exclusive we introduce constraints.

Example Let $S = \{\texttt{parcelAt}(\texttt{p}_1,\texttt{node}_1), \texttt{agentAt}(\texttt{a}_2,\texttt{node}_1)\}$. Using Algorithm 4 we compute the set of bindings for Agent $\texttt{a}_2$'s operator $o = \texttt{pickup}(\texttt{a}_2,\texttt{P},\texttt{L})$ to be:

Substituted Operator - $\varsigma[o]$	Bindings - ς	Constraints -κ
$\texttt{pickup}(\texttt{a}_2,\texttt{p}_1,\texttt{node}_1)$	$\{(\texttt{P} \leftarrow \texttt{p}_1), (\texttt{L} \leftarrow \texttt{node}_1)\}$	$\emptyset$
$\texttt{pickup}(\texttt{a}_2,\texttt{p}_1,\texttt{L})$	$\{(\texttt{L} \leftarrow \texttt{node}_1)\}$	$\{(\texttt{L} \not\leftarrow \texttt{node}_1)\}$
$\texttt{pickup}(\texttt{a}_2,\texttt{P},\texttt{node}_1)$	$\{(\texttt{P} \leftarrow \texttt{p}_1)\}$	$\{(\texttt{P} \not\leftarrow \texttt{p}_1)\}$
$\texttt{pickup}(\texttt{a}_2,\texttt{P},\texttt{L})$	$\emptyset$	$\{(\texttt{L} \not\leftarrow \texttt{node}_1), (\texttt{P} \not\leftarrow \texttt{p}_1)\}$

□

Given a related binding ς and constraint set κ we write ς_κ to symbolise that the variables in ς are governed by constraints in κ. Algorithm 5 details our implementation of O_{cont} given abstract operator specifications, taking as input a conflict specification and the set of operator schemata, and returning the set of contributing operators.

Algorithm 5: O_{cont} – Enumerating Contributing Operators in a Classical Formalism

```
Input: Conflict specification $S_C$, and list of operator schemata $O$
Result: The set $O_P$ of contributing operators.
1 begin
      Initialise $O_P$ to the empty set
2     $O_P \leftarrow \emptyset$
3     for each $o \in O$ do
          Identify the set of all bindings of the effects of $o$ to $S_C$
4         $2^\varsigma \leftarrow BindingEnumeration(post(o), S_C)$
          For each binding add the substituted operator
5         for each $\varsigma \in 2^\varsigma$ do
              Create the set of constraints
6             $\kappa \leftarrow ConstraintsFromBinding(\varsigma)$
7             $O_P \leftarrow O_P \cup \varsigma_\kappa[o]$
8     return $O_P$
9 end
```

Similarly, Algorithm 6 implements O_{par} with abstract operator specifications. The key difference here is that the operator's *preconditions* are bound to the source state, as opposed to the operator effects in Algorithm 5. The remainder of the conflict traversal procedure follows as with the propositional domain formalism.

Algorithm 6: O_{par} – Enumerating Partially Applicable Operators in a Classical Formalism

```
  Input: Specification $S$, and list of operator schemata $O$
  Result: The set $O_S$ of partially applicable operators.
1 begin
      Initialise $O_S$ to the empty set
2     $O_S \leftarrow \emptyset$
3     for each $o \in O$ do
          Identify the set of all bindings of the preconditions of $o$ to $S$
4         $2^{\varsigma} \leftarrow BindingEnumeration(pre(o), S)$
          For each binding add the substituted operator
5         for each $\varsigma \in 2^{\varsigma}$ do
              Create the set of constraints
6             $\kappa \leftarrow ConstraintsFromBinding(\varsigma)$
7             $O_S \leftarrow O_S \cup \varsigma_{\kappa}[o]$
8     return $O_S$
9 end
```

4.3.5 Norm Synthesis

The process of synthesising appropriate norms from the set of complete runs produced by the conflict traversal remains unchanged from the propositional approach defined in Algorithm 2. Although the algorithm is identical to before, the resulting set of norms are far more expressive since they are synthesised over unground runs.

Example The conflict specification $S_C = \{\mathtt{at}(\mathtt{a}_1,\mathtt{Y}), \mathtt{at}(\mathtt{a}_2,\mathtt{Y})\}$ prohibits collisions in any node in the Parcel Delivery domain. One of the resulting norms from this specification might be $n = \langle \varphi, o \rangle$ where:

- $\varphi = \{\ \mathtt{at}(\mathtt{a}_1,\mathtt{X}),\ \mathtt{at}(\mathtt{a}_2,\mathtt{Y}),\ \mathtt{conn}(\mathtt{X},\mathtt{Y})\ \}$
- $o = \mathtt{move}(\mathtt{a}_1,\mathtt{X},\mathtt{Y})$.

That is, Agent $\mathtt{a}_1$ must not move into any adjacent node if Agent $\mathtt{a}_2$ is occupying the node. A similar norm is generated for Agent $\mathtt{a}_2$. In this representation variables with the same name are identical across terms. Our norms are therefore more expressive since we are able to quantify over unbound variables in the norm specification. □

4.3.6 Classical Reachability Analysis

We have previously presented our reachability analysis algorithm composed of two core steps:

1. the *norm application* procedure that regiments the domain representation so that our candidate norms are adhered to, and
2. the *reachability checking* procedure that solves planning problems in the normative system.

The unground nature of complete runs is problematic for both the steps above. Firstly, when rewriting operators to include synthesised norms we require a means of quantifying over variables in the norms. Secondly, we previously constructed the initial and goal state specifications of each planning problem from the first and last state specifications of each complete run, yet classical planners are unable to plan using unground specifications. In order to continue to use classical planners these specifications must be ground prior to planning.

4.3.6.1 Normative Planning with Variables

Consider the norm $n = \langle S_1, o_1 \rangle$ where S_1 and o_1 contain variable symbols. Suppose an agent, currently in a ground specification S^*, wishes to check whether the condition of n holds in S^*. It is not immediately clear whether or not n applies in S^*, since S_1 contains variables that require binding.

The agent wishes to reason whether an action a is forbidden in S^* under a prohibitionary norm set $\mathcal{N}$. We define whether or not an action is forbidden by extending the prohibition function defined in our normative planning extension in Section 4.2.4.1. Formally, we redefine the prohibition function as:

$$F(S, o, \mathcal{N}, \sigma) = \begin{cases} \top & \text{if } \exists \langle \varphi, o' \rangle \in \mathcal{N} \text{ where } (S \models \sigma[\varphi]) \wedge (\sigma[o'] = o) \\ \bot & \text{otherwise.} \end{cases}$$

The only difference between this prohibitionary function and the propositional one is the introduction of the substitution σ. The requirement to ground variable symbols is a common addition in this classical planning extension, and it continues here.

When rewriting our operator schemata to incorporate the norms we have synthesised we again must account for variables. Consider previously that we redefined the preconditions of an operator by creating a conjunction composed of the negations of each norm condition:

$$pre'(o) = pre(o) \wedge \Big[\bigwedge_{\langle \varphi, o \rangle \in N_o} \neg \varphi \Big].$$

This rewriting procedure is not directly applicable for two reasons:

1. The operator referenced by a norm may contain a partial assignment of the parameters of the operator. For example, a norm prohibiting the partially ground operator $\texttt{move}(\texttt{a}_1, \texttt{From}, \texttt{To})$ will have a precondition that contains the variables To and From. We require a mechanism whereby these variables will be ground appropriately.

2. The condition specification of a norm may contain variable references that do not form part of the operator parameters. The resulting operator schema may refer to variables that are not bound when assignments are made for all parameters of the operator. For example, suppose φ contains an unbound variable v that is not a parameter of o. The written precondition $pre'(o)$ will still contain the variable v as it cannot be bound. We require a means of binding v appropriately so that we can utilise standard planners to check reachability.

We deal with each of these shortcomings in turn and present a new rewrite procedure to conclude. First we introduce two key concepts:

- Consider an operator written in a parameterised form $o(p_1 \dots p_n)$ where we write $p_i(o)$ to refer to the ith parameter of o, and we write $\mathcal{P}(o)$ to represent the set of parameters of o. Now consider o^* to be an instance of o with a subset of its parameters ground. We represent these assignments for operator o^* as $\alpha(o^*)$, a set composed of assignments $(p_i \leftarrow c_i)$ where p_i is a parameter of o and $c_i \in \mathcal{L}_c$ is a constant symbol.

  We wish to rewrite the specifications of the original operator schema. If a norm prohibits a more specific version of an operator, then it is essentially prohibiting the abstract operator in the presence of some existing parameter assignments. The condition $C(o,n)$ that must hold for a norm n, with partially assigned operator $o(n)$, to prohibit o are:

$$C(o,n) = \bigwedge_{(p_i \leftarrow c_i) \in \alpha(o(n))} p_i(o) = c_i.$$

  That is, for each parameter assignment in $\alpha(o(n))$, the set of assignments for the operator prohibited by n, we stipulate that this condition only holds if a corresponding assignment exists for the original operator o.

  Example Consider $o = \texttt{move}(\texttt{Agent}, \texttt{From}, \texttt{To})$. Let o^* be an instance of o with some of the parameters assigned: $o^* = \texttt{move}(\texttt{a}_1, \texttt{node}_1, \texttt{To})$. Here, the assigned parameters are $\alpha(o^*) = \{(\texttt{Agent} \leftarrow \texttt{a}_1), (\texttt{From} \leftarrow \texttt{node}_1)\}$. If a norm n prohibits

o^* then we write the condition under which n is active for the operator o as follows:

$$C(o,n) = (\texttt{Agent} = \texttt{a}_1) \wedge (\texttt{From} = \texttt{node}_1).$$

This captures the fact that n prohibits o, but only when parameter assignments match those in o^*. $\square$

- We also require a means of identifying which variable symbols appear in a state specification. Let $\mathit{Vars} : 2^{L_{\mathcal{A}}} \to 2^{\mathcal{L}_v}$ be a function that, when given a specification S, returns a set of variable symbols that appear in S. Now consider that an operator o with variable parameters $\mathcal{P}(o)$ is to be applied in state specification S. We define $V(S,o)$ to be a function that returns the set of variable symbols in S that are not parameters of o:

  $$V(S,o) = \mathit{Vars}(S) \backslash \mathcal{P}(o).$$

  We write $\sigma_{V(S,o)}$ to represent a substitution over these variables.

  Example Let $S = \{\texttt{at}(\texttt{Agent}, \texttt{node}_1), \texttt{parcelAt}(\texttt{Parcel}, \texttt{node}_1)\}$ and the operator $o = \texttt{move}(\texttt{Agent}, \texttt{From}, \texttt{To})$. Here, $\mathit{Vars}(S) = \{\texttt{Agent}, \texttt{Parcel}\}$ identifies the variable symbols in S. The symbols that are not parameters of o are $V(S,o) = \{\texttt{Parcel}\}$. $\square$

We now present a reworked rewrite procedure and comment on it below:

$$\mathit{pre}'(o) = \mathit{pre}(o) \wedge \bigwedge_{\langle\varphi, o_n\rangle \in N_o} \neg \Big[C(o, \langle\varphi, o_n\rangle) \wedge \big(\exists \sigma_{V(S,o)}.\sigma[\varphi]\big) \Big].$$

The rewrite procedure can be described as follows. An operator o is permitted to be applied in a state specification if its preconditions are satisfied ($\mathit{pre}(o)$) and no norms are satisfied. A norm n is satisfied if it prohibits the operator o, and is active under the following conditions:

- each of the parameter assignments of the operator prohibited by the norm are present ($C(o,n)$), and
- there exists some substitution σ for all the unbound variables where the grounded precondition of the norm holds.

This rewrite rule satisfies all the shortcomings of the propositional operator rewriting. We ensure that existing parameter assignments form part of the rewrite rule. As a result, the condition of a norm is only considered applicable if the particular instance of the operator matches these assignments. Furthermore, by quantifying over all substitutions of the remaining unbound variables we are guaranteed to have a fully ground specification with which to check for applicability.

4.3.6.2 Grounding Runs

In order to guarantee reachability under the new synthesised prohibitions we must show that a conflict-free plan exists for any state represented by the initial state specification $\mathit{first}(R)$, that results in a state represented by $\mathit{last}(R)$, for all $R \in \mathcal{R}$. Furthermore, we must consider that the state specifications $\mathit{first}(R)$ and $\mathit{last}(R)$ could contain variables. For this, we extend the reachability check to show that a conflict-free alternative plan exists for any grounding of these specifications.

Consider reachability checking for a run R containing specifications that are not grounded. In order to always find an alternative, conflict-free plan to R additional domain-specific knowledge is required.

Example Consider the conflict specification $S_C = \{\mathtt{at(a_1,node_1)}, \mathtt{at(a_2,node_1)}\}$. One example of a complete run found during traversal where Agent $\mathtt{a_2}$ moves from some location (X), into $\mathtt{node_1}$, and then again to some other location (Y).

$$\left\{\begin{array}{c}\mathtt{at(a_2,X)},\\ \mathtt{at(a_1,node_1)},\\ \mathtt{conn(X,node_1)},\\ \mathtt{conn(node_1,Y)}\end{array}\right\} \xrightarrow{\mathtt{move(a_2,X,node_1)}} \left\{\begin{array}{c}\mathtt{at(a_2,node_1)},\\ \mathtt{at(a_1,node_1)},\\ \mathtt{conn(X,node_1)},\\ \mathtt{conn(node_1,Y)},\\ \neg\mathtt{at(a_2,X)}\end{array}\right\} \xrightarrow{\mathtt{move(a_2,node_1,Y)}} \left\{\begin{array}{c}\mathtt{at(a_2,Y)},\\ \mathtt{at(a_1,node_1)},\\ \mathtt{conn(X,node_1)},\\ \mathtt{conn(node_1,Y)},\\ \neg\mathtt{at(a_2,node_1)},\\ \neg\mathtt{at(a_2,X)},\end{array}\right\}$$

Finding an alternative, conflict-free plan for this run is dependent on the topology of the underlying world. Consider two instances of such a grid world in Figure 4.12. In world (i) reachability does not hold since there is no sequence of actions that will result in Agent $\mathtt{a_2}$ reaching $\mathtt{node_3}$ without colliding with Agent $\mathtt{a_1}$. In (ii) the alternative, conflict-free run is a trivial move from $\mathtt{node_2}$ to $\mathtt{node_3}$. $\square$

Our reachability algorithm changes appropriately to incorporate the additional domain knowledge prior to solving the corresponding planning problem. If a run R is unground, then it is modelled by a set of ground runs in the underlying system. We identify each of these ground runs through the substitution that, when applied to R, results in a grounded instance $\overline{R}$. Consider a set of such substitutions $\{\sigma_1, \sigma_2 \ldots\}$ and

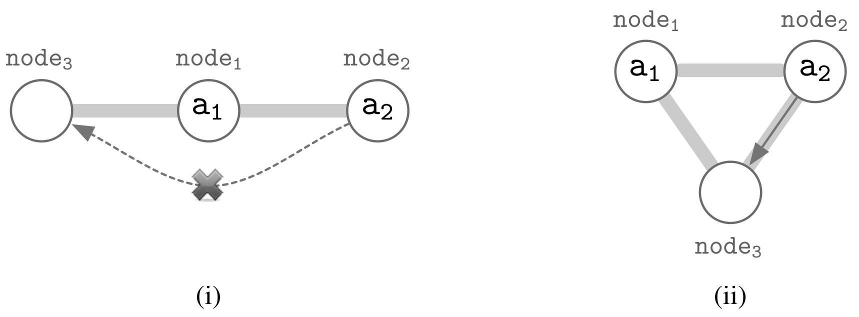


Figure 4.12: Two Parcel Delivery worlds illustrating how the state reachability of unground runs is dependent on the topology of the world.

a run R. To show reachability of conflict-free states in the unground run R we must show reachability between states for each of the grounded runs $\sigma_1[R]$, $\sigma_2[R]$ .... If no conflict-free alternative is found for a single grounded run, then no guarantees exists for the general reachability of the ungrounded run. We adjust the reachability analysis as follows:

1. For the run R, find the set $\{\sigma_1, \sigma_2 \ldots\}$ of consistent constant-only substitutions that result in unique groundings of *first*(R) and *last*(R), given the set of possible atoms $\overline{\mathcal{A}}$.
2. For each substitution σ, solve the planning problem Π for $S_I = \sigma[first(R)]$ and $S_G = \sigma[last(R)]$ under the synthesised prohibitions.

There are advantages to grounding the conflict-free pairs at the reachability analysis stage of the synthesis. Firstly, the traversal runs are ungrounded and therefore common to all problem instances of the domain specification. Secondly, the refinement of runs during traversal can be seen as a process whereby constraints are placed on the possible variable groundings for any specific state: the grounding of variables need only consider those atoms that satisfy the prohibition conditions in the problem instance. This, coupled with object typing, reduces the number of unique grounded runs that are considered for reachability. Finally, if we consider the operator rewrite process under the synthesised prohibitions it becomes clear that grounding prior to, or during the traversal produces many prohibitions conditional on each variable grounding, and does not take advantage of the generality of the planning formalism and action schemata. We present the revised reachability analysis in Algorithm 7.

Algorithm 7: Modified Reachability Analysis in the Classical Planning Formalism

```
Input: Set of complete runs $\mathcal{R}$, candidate norms $\mathcal{N}$ and domain structure $\Xi$
Result: TRUE if reachability is satisfied, FALSE otherwise.
1  begin
       Create the restricted prohibitionary system
2      $\Xi' \leftarrow NormApplication(\Xi, \mathcal{N})$
       Ensure reachability between conflict-free states for each conflict run
3      for each complete run $R \in \mathcal{R}$ do
           Identify all substitution groundings for the unground $R$
4          for each grounding $\sigma_i$ of $R$ in $\Xi$ do
               Construct a planning problem to verify reachability
5              $S_I \leftarrow first(\sigma[R])$
6              $S_G \leftarrow last(\sigma[R])$
7              $\Pi \leftarrow \langle \Xi', S_I, S_G \rangle$
               Invoke a planner to solve the planning problem
8              $\Delta \leftarrow InvokePlanner(\Pi)$
9              if $\Delta$ is not a valid solution then
10                 return FALSE
       All the runs have been verified as reachable
11     return TRUE
12 end
```

4.3.7 Algorithm Properties

We now reassess our arguments for termination, complexity, soundness and completeness for conflict-rooted synthesis in a classical planning domain.

4.3.7.1 Termination

If we consider propositional planning, then the traversal process is guaranteed to terminate since the set of literals is bounded and each successor operator only adds literals to specifications in the run. Furthermore, the reachability check for these runs is also grounded. While our first order extension provides the advantage of generally applicable, variable operators, it results in the loss of any implicit problem specific information. As a result, when successor operators are considered they can introduce new, unbounded variables into runs: the process might repeat infinitely and never terminate.

Theorem 4.3.3. *Conflict-rooted synthesis bound to a classical planning formalism is not guaranteed to terminate.*

Proof. We prove non-termination by showing that conflict traversal is not guaranteed to terminate. Consider a simplified Parcel Delivery world where agents can only move. Assume Agent $\texttt{a}_1$ begins in $\texttt{node}_1$. Through refinement and inference we generate the run:

$$R = \left\{ \begin{array}{c} \texttt{at}(\texttt{a}_1,\texttt{node}_1), \\ \texttt{conn}(\texttt{node}_1,\texttt{Node}) \end{array} \right\} \xrightarrow{\texttt{move}(\texttt{a}_1,\texttt{node}_1,\texttt{Node})} \left\{ \begin{array}{c} \texttt{at}(\texttt{a}_1,\texttt{Node}), \\ \neg\texttt{at}(\texttt{a}_1,\texttt{node}_1), \\ \texttt{conn}(\texttt{node}_1,\texttt{Node}) \end{array} \right\}$$

with the constraint that $\texttt{Node} \neq \texttt{node}_1$. We can continue to pick new move operators in the sequence:

$$\begin{array}{rcl} \texttt{move}(\texttt{a}_1,\texttt{Node},\texttt{Node}_1) & \text{where} & (\texttt{Node}_1 \neq \texttt{Node}) \\ \texttt{move}(\texttt{a}_1,\texttt{Node}_1,\texttt{Node}_2) & \text{where} & (\texttt{Node}_2 \neq \texttt{Node}_1) \\ \texttt{move}(\texttt{a}_1,\texttt{Node}_2,\texttt{Node}_3) & \text{where} & (\texttt{Node}_3 \neq \texttt{Node}_2) \\ & \dots & \end{array}$$

Using this process we are always able to create a new run that has not been considered to date. Conflict traversal is not guaranteed to terminate. ■

We propose two techniques to improve the termination properties of our algorithm:

1. Optimisations, such as those described in the following Chapter, reduce the number of infinite runs considered by discarding sequences that are known to be reachable.

2. Termination can be guaranteed by bounding the algorithm. We investigate possible bounding strategies next.

Bounding the Traversal The simplest means of bounding the traversal is to limit either the number of runs produced during traversal, or the maximum length of any run investigated. These strategies are similar: once the run limit is exceeded the traversal process can be queried to identify the run length of the last iteration. Limiting traversal in this way negatively affects the guarantees that conflict-rooted synthesis produces during reachability analysis. Since a subset of the complete runs are analysed, ensuring reachability only provides guarantees that the same subset is achievable in the normative system. In our evaluation we utilise the bounding of the traversal process to ensure that traversal is consistently limited while analysing the properties of our algorithm. For a full comparison a superior strategy is required that can provide guarantees over all runs in a domain.

Domain knowledge (initial state knowledge from a planning perspective) is required to bound the traversal. Theorem 4.3.3 highlights that, without domain knowledge, traversal is not guaranteed to terminate. The remaining question then is, given this domain knowledge, how can we effectively bound the traversal process? To this end we introduce a bound based on limiting the number of predicates in state specifications investigated during traversal.

Our strategy is simple to implement. Given initial state knowledge we count the number of predicates, and each time refinement increases the size of a state specification we ensure that none of the predicate limits have been breached.

Example In Theorem 4.3.3 we illustrated how the core synthesis process does not terminate by illustrating that an arbitrary number of `move` operators could be sequenced. Each move operator introduced a new predicate into our state specification during refinement: $\texttt{conn}(\texttt{Node}_1,\texttt{Node}_2)$, $\texttt{conn}(\texttt{Node}_2,\texttt{Node}_3)$, ... It is clear that, given a particular problem instance, we are able to deduce the maximum number of `conn` predicates allowed. For example, in a 2x2 grid this equates to 8 bidirectional links. By ensuring no specification references more than 8 `conn` predicates we can effectively bound the traversal. □

What is particularly appealing about this approach is that the norms synthesised are independent of the initial state knowledge, and the Reachability Analysis results guarantee reachability for the given problem instance.

4.3.7.2 Complexity

The classical planning formalism adopted in this thesis does not include functional symbols, implying that the systems represented by these domain are finite. Bylander (1994) showed that the plan existence decision problem in these domains is PSPACE-complete. Conflict-rooted synthesis in classical domains differs from the propositional variant in that unground predicates are introduced during state refinement. This, coupled with the lack of problem knowledge during traversal implies that the search space is infinite, as new unique predicates can be introduced arbitrarily. As a result, we bound the traversal process by incorporating problem-specific knowledge, thereby ensuring that a limit exists for the maximum run length, and subsequently that the algorithm is guaranteed to terminate. Under these assumptions, the worst case complexity remains unchanged from the propositional.

4.3.7.3 Soundness and Completeness

Our soundness and completeness argument follows the propositional result in Theorem 4.2.15. We assume the traversal to be bound, and that the planner used for reachability analysis is sound and complete.

Theorem 4.3.4. *Bound conflict-rooted synthesis in classical domains is sound and complete.*

Proposition 4.3.5. *The bound conflict traversal algorithm is sound and complete for classical planning domains.*

Proof. We adopt a proof by contradiction. Assume $R = S_1 \xrightarrow{\texttt{o}_1} \dots \xrightarrow{\texttt{o}_{n-1}} S_n$ is a complete conflict run that is not identified during traversal. Since the core traversal is sound by Theorem 4.2.15 it follows that there are two reasons why R is not identified:

1. The set of contributing operators does not include the operator o_1.
2. At least one of $o_2 \dots o_{n-1}$ was not found in the set of partially applicable operators.

Completeness and soundness therefore follows from the completeness and soundness properties of O_{cont} and O_{par}. The operators in O_{cont} and O_{par} are all created through bindings generated by the function *BindingEnumeration*: for an operator to be missing, a corresponding binding must be erroneously left out. Yet this cannot be the case, since *BindingEnumeration* enumerates all possible bindings between all subsets of predicates in each of the given state specifications. Since all operators are considered correctly, every valid conflict run will be found, and a contradiction is reached. ■

Our proof for the Norm Synthesis algorithm follows identically as before. Next, we investigate reachability analysis.

Proposition 4.3.6. *In a classical domain, if reachability is ensured then a conflict-free alternative sequence of actions exists in the normative system for every conflict sequence in the original.*

Proof. We refrain from a complete proof of reachability, choosing rather to focus on the core change from the propositional approach. Since conflict runs are unground, reachability analysis performs a grounding step to identify every run represented. If reachability checking is successful for every ground run, then we deduce that the reachability requirements are satisfied for the unground run.

The soundness of reachability analysis in a classical domain is dependent on two factors:

- Soundness and completeness of the adopted planner.
- Soundness and completeness of the grounding algorithm used.

In our work we assume the planner to be both sound and complete while the grounding approach simply enumerates all possible bindings of each variable in the run, incorporating those that do not violate the run's constraints. Since every possible assignment is considered, no possible binding is ignored, and every binding returned is valid. It therefore follows that reachability is sound in classical domains. ■

From Propositions 4.3.5 and 4.3.6 it follows that Theorem 4.3.4 holds. The inherent value in keeping variable constraints becomes clear through our grounding algorithm discussion, since with no constraints all combinations of bindings must be considered during grounding. Constraints help to reduce the number of runs produced, allowing only runs that are consistent and valid in the problem domain.

4.4 Conflict-Rooted Synthesis Summary

The conflict-rooted synthesis algorithm is separated into three components:

1. **Conflict Traversal**: Inference and refinement operators facilitate a localised search of the conflict specification space. The main output of this process is a set of complete runs that represent everything achievable through conflict.
2. **Norm Synthesis**: Given the set of complete runs, norm synthesis extracts appropriate prohibitionary norms where the contributing operators are prohibited conditional on the first specification of each run.
3. **Reachability Analysis**: With the set of complete conflict runs and synthesised norms, reachability analysis invokes an external planner to verify that each of the complete runs is achievable using a conflict-free plan in the normative system.

The semantics of our approach were detailed in a state transition system. The algorithm was then extended to utilise state and operator abstractions in propositional and classical planning domains. In the propositional setting we showed conflict-rooted synthesis to always terminate, while always being sound and complete. Since less information is contained in classical domains we introduced a set of approaches to bounding the traversal, thereby ensuring that the approach continues to terminate and to be sound and complete.

In the worst case conflict-rooted synthesis is intractable, yet in Chapter 7 we show that it is still applicable in a number of benchmark domains. In order to reduce the need for unnecessary computation we introduce a set of domain independent optimisations of the algorithm next.

Chapter 5

Synthesis Optimisations

Even though conflict-rooted synthesis utilises state and operator abstractions a sizeable number of runs are generated during traversal for even the simplest domains. In the Parcel Delivery domain a full traversal considers over 350000 complete runs of length 5 or less. There is clearly incentive to reduce the number of runs investigated during synthesis, especially when each resulting run produces a set of planning problems to be solved. In order to reduce the computation required to synthesise norms we focus on two classes of optimisations:

1. **Traversal Optimisations**: are performed during, or directly after conflict traversal. They prune the traversal space allowing for a reduction in the number of subsequent planner invocations.

2. **Reachability Optimisations**: are performed in the reachability analysis phase of the algorithm. These reduce the number of planning iterations required to ensure reachability.

In this work we intentionally use the term "optimisation" rather than "heuristic". Recall that we perform an exhaustive traversal search and reachability check to determine the effectiveness of our candidate norms without any assumptions or approximations. It is important that every run is considered since we require every outcome to be achievable in the normative system. We use the term optimisation since we are concerned with changes to the algorithm that will produce identical results to the original, but with some computation or space reduction. None of the approaches we detail in this chapter alter the soundness or completeness properties of the conflict-rooted synthesis algorithm. The main contributions of this chapter were previously published by Christelis et al. (2010).

5.1 Traversal Optimisations

We begin by illustrating the intuition behind our traversal optimisations. The search adopted by conflict traversal produces a set $\mathcal{R}_C$ containing every complete conflict run. Traversal optimisations reduce the size of $\mathcal{R}_C$.

Proposition 5.1.1. *Let R_1 and R_2 be runs contained in $\mathcal{R}_C$. R_1 can be removed from $\mathcal{R}_C$ if it holds that the reachability of states in R_1 is* dependent *on the reachability of R_2.*

This dependency relationship implies that, during reachability analysis, if a conflict-free run exists for R_2 then a conflict-free run must exist for R_1.

Example Consider the trivial case where R_2 is identical to R_1. Let Δ be an alternative, conflict-free plan found during reachability analysis for R_2. Since R_1 is identical to R_2, then we know that Δ is also an alternative for R_1. There is no reason to consider R_1 during reachability analysis: R_1 can be removed from $\mathcal{R}_C$. □

In Section 4.1.3.3 we detailed how planning problems are constructed from runs in order to ensure goal reachability in the normative system. Importantly, for a given run R a planner searches for plans with initial state $\mathit{first}(R)$ and goal state $\mathit{last}(R)$. All other intermediate specifications do not form part of the planning problem. We now refine our notion of reachability dependence, writing $S_1 \equiv S_2$ to represent set equivalence.

Definition 5.1.2. *Let R_1 and R_2 be runs contained in $\mathcal{R}_C$. The reachability of states in R_1 is* dependent *on the reachability of those in R_2 if both of the following hold:*

- $\mathit{first}(R_1) \equiv \mathit{first}(R_2)$.
- $\mathit{last}(R_1) \equiv \mathit{last}(R_2)$.

Proof. The proof follows directly from the fact that both R_1 and R_2 will produce identical planning problems during reachability analysis if the above conditions hold. As such, any conflict-free alternative plan for R_1 will also hold for R_2. ■

The traversal optimisations in this chapter remove runs from consideration using the following two guidelines:

1. **Redundancy**: If an optimisation guarantees a run to be reachable in the normative system then it is excluded.
2. **Dependency**: If an optimisation shows that the reachability of conflict-free states in a run is dependent on the reachability of states in an already investigated run, then the run is removed.

Figure 5.1 provides a visual representation of the set of conflict runs. Dependencies between runs are illustrated using directed edges. Our optimisations exclude runs known to be reachable, or runs dependent on others. Now we need only check the remaining subset of runs during reachability analysis.

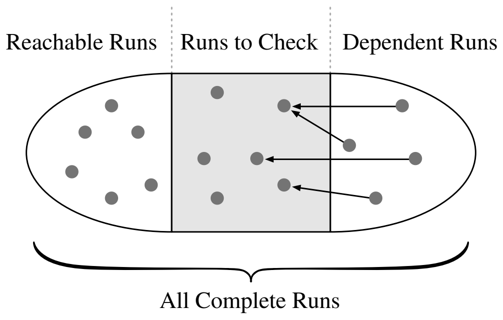


Figure 5.1: Identifying reachable and dependent runs in the set of complete runs.

Dependency removal also allows us to remove *incomplete* runs during the traversal process too. Let $\mathcal{U}$ be the set of incomplete runs after some iterations of the conflict traversal algorithm. The conditions in Definition 5.1.2 can be used to remove incomplete runs from consideration. Let R_1 and R_2 be two runs in $\mathcal{U}$. Since the final specifications of R_1 and R_2 are identical, we know that the subsequent searches during traversal will be identical too, resulting in a duplication of search effort. In this situation we can simply remove one of R_1 or R_2 from $\mathcal{U}$.

A common technique we adopt in our optimisations to expose dependency relations is *operator reordering*. Consider an arbitrary, complete run of the form:

$$R_1 = S_1 \xrightarrow{\mathtt{o_1}} \dots \xrightarrow{\mathtt{o_{n-1}}} S_n.$$

Let R_2 be a run constructed by reordering o_k to the beginning of R_1. We assume R_2 to be consistent and that the reordering does not affect the first and last specifications of the run. We present the reordered R_2 below:

$$R_2 = S_1 \xrightarrow{\mathtt{o_k}} S_2' \xrightarrow{\mathtt{o_1}} \dots \xrightarrow{\mathtt{o_{k-1}}} S_k' \xrightarrow{\mathtt{o_{k+1}}} S_{k+1}' \cdots \xrightarrow{\mathtt{o_{n-1}}} S_n.$$

Notice that $\mathit{first}(R_1) \equiv \mathit{first}(R_2)$ and $\mathit{last}(R_1) \equiv \mathit{last}(R_2)$, while all other specifications are possibly different. We illustrate R_1 and R_2 in Figure 5.2. In both cases the contributing operator o_1 has been emphasised. Notice that if o_k does not lead to conflict then the sequence of conflict specifications has been shortened. We shade conflict specification nodes appropriately.

The two runs are now dependent by Definition 5.1.2 since the shorter conflict sequence in R_2 is considered prior to R_1 and R_1 can therefore be removed from consid-

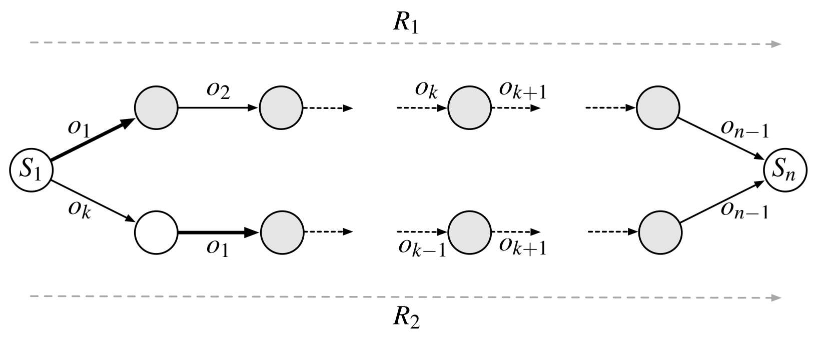


Figure 5.2: Comparing runs in which an operator has been reordered.

eration.

Each of the optimisations presented in this section can be applied independently of the domain, and they generically exploit implicit constraints and dependencies between operators. The effectiveness of these optimisations is dependent on what characteristics any particular domain has, but exploiting these characteristics can be performed in a domain independent fashion. Importantly, the optimisations we present never increase the number of traversal runs.

5.1.1 Traversal Optimisations Overview

We begin our presentation of traversal optimisations by providing an overview of each optimisation in the context of an example run. Consider the complete conflict run produced for the simple conflict specification $S_C = \{\texttt{at}(\texttt{a}_1,\texttt{node}_1)\}$:

$$S_1 \xrightarrow{\texttt{o}_1} S_2 \xrightarrow{\texttt{o}_2} S_3 \xrightarrow{\texttt{o}_3} S_4 \xrightarrow{\texttt{o}_4} S_5 \xrightarrow{\texttt{o}_5} S_6 \xrightarrow{\texttt{o}_6} S_7$$

where the operators o_i are listed in the following table:

Index	Operator
o_1	move(a₁,X,node₁)
o_2	destroy(a₁,parcel₁)
o_3	pickup(a₁,parcel₂,node₁)
o_4	idle(a₁)
o_5	pickup(a₁,parcel₃,node₁)
o_6	move(a₁,node₁,Y)

We include two additional operators: an agent can `destroy` a held parcel, resulting in the parcel no longer existing, or can remain `idle` with no effects. This run forms part of the set of complete runs generated for the specification S_C, alongside many other complete runs. We number and detail each optimisation in the context of this example.

1. **A Priori Operator Filtering**

   Through operator dependency analysis identify which operators depend on, or contribute to, the conflict specification. The set of independent operators that remain need not be considered during traversal, resulting in a smaller operator set and subsequently reducing the number of runs produced.

   The operator `idle`(a_1) in the above run is discarded since it is not dependent on, and does not contribute to the literals in S_C. Removing the operator results in a shorter run with conflict-free state reachability dependent on the longer.

2. **Traversal Pruning**

   Exploit operator reordering during each iteration of conflict traversal to identify shorter dependent runs. If a shorter run is identified, the longer is discarded.

   In the example the agent can invoke the `destroy`(a_1,$\texttt{parcel}_1$) action first, prior to moving into $\texttt{node}_1$ while maintaining the effects of the `destroy` action. The remaining operators form a shorter run, and the longer run is discarded.

3. **Duplicate Run Removal**

   We need not consider complete runs that are duplicates of each other as already highlighted in Definition 5.1.2.

4. **Partial Order Sequencing**

   Identify operators in the run that can be applied in any order since their preconditions and effects are independent. Arbitrary orders of the operators results in a set of identical runs, all of which are dependent on one another. We select one of these for analysis and discard the remainder.

   The two actions `pickup`(a_1,$\texttt{parcel}_2$,$\texttt{node}_1$) and `pickup`(a_1,$\texttt{parcel}_3$,$\texttt{node}_1$) in our example can be executed in any order. We need only consider a single ordering of these actions.

5. **Repetitive Operators**

   Runs that model the repeated application of an operator can share common reachability plans. More specifically, in certain situations altering a run that has been shown to be reachable by repeating a particular operator in the run will result in similar repetitions in the alternative conflict-free run. Inductively, if we can show that the reachability of states in the modified run is still satisfied by the modified conflict-free run, then an arbitrary repetition is catered for as well.

In our example run above, if a conflict-free plan exists for $\mathtt{a}_1$ to pick up $\mathtt{parcel}_2$, then a similar plan exists for $\mathtt{a}_1$ to pick up $\mathtt{parcel}_3$, or any other parcel in $\mathtt{node}_1$. We consider these repeated actions as a single instance, thereby reducing the number of runs considered. Furthermore, this allows us to reason about the reachability of states incurred through sequences of actions of arbitrary length.

These optimisations never increase the search space size, make no assumptions regarding agent goals and preserve soundness by disregarding complete runs that are known to be reachable. We formally present each of these optimisations in turn.

5.1.2 Traversal Optimisation 1: A Priori Operator Filtering

A priori operator filtering is an optimisation that reduces the set of operator schemata that are considered prior to beginning traversal. It is not sufficient to only consider operators that are partially applicable in the conflict specification. We also must consider operators that are independent of the conflict specification, since applying independent operators may result in a dependent operator becoming applicable. We begin by proving this fact via Proposition 5.1.4 below which shows that operators that are entirely independent of the conflict specification S_C must still be considered during traversal.

Definition 5.1.3. *An operator $o \in O$ is* independent *of a state specification S if the operator is not dependent on, and does not affect any of the literals in S. Formally, o is* independent *of S if and only if:*

$$\nexists \varsigma \textit{ where } \forall l \in \big(pre(\varsigma[o]) \cup post(\varsigma[o])\big) \;.\; l \notin S \wedge \neg l \notin S.$$

We write $lit(o) = pre(o) \cup post(o)$ to be the set of literals in o.

Proposition 5.1.4. *Let $O_I \subseteq O$ be the subset of operators that are independent of a conflict-state specification S_C. Operators in O_I cannot be removed from consideration during conflict traversal a priori.*

Proof. We present a proof by counterexample. Consider the following three abstract operators operating over literals $\{\mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{q}\}$:

$$\{\mathtt{z}\} \xrightarrow{\mathtt{o_1}} \{\mathtt{x}, \mathtt{y}, \neg\mathtt{z}\} \qquad \{\mathtt{y}\} \xrightarrow{\mathtt{o_2}} \{\mathtt{q}\} \qquad \{\mathtt{q}\} \xrightarrow{\mathtt{o_3}} \{\neg\mathtt{x}\}$$

Let $S_C = \{\mathtt{x}\}$. Operator o_1 is a contributing operator that results in a conflict state, while o_3 is guaranteed to leave a conflict state. Let the set of independent operators $O_I = \{o_2\}$ since neither $\mathtt{y}$ nor $\mathtt{q}$ are atoms that exist in S_C. By not considering o_2

during traversal we acknowledge that no run containing o_2 exists that is not reachable in the resulting normative system. Our counterexample must show that a run exists that contains o_2 that is not reachable when access to S_C is prohibited. Consider the complete run:

$$\{\mathtt{z}\} \xrightarrow{\mathtt{o_1}} \{\mathtt{x},\mathtt{y}\} \xrightarrow{\mathtt{o_2}} \{\mathtt{x},\mathtt{y},\mathtt{q}\} \xrightarrow{\mathtt{o_3}} \{\mathtt{y},\mathtt{q}\}$$

which achieves q through the application of o_2. Even though o_2 makes no reference to any atoms in S_C, it is enabled as a side effect of performing o_1: o_2 is not dependent on S_C but is only applicable in a conflict state. There is no conflict-free way to achieve q since the above run cannot be applied in the normative system. If the independent operators in O_I are ignored, the above run is not considered and is assumed reachable. All other conflict runs are shown to be reachable, and we deduce that the synthesised norms preserve reachability in the general case. A contradiction is reached. ■

A stronger notion of operator independence is required. Instead of showing independence between o and S we also show independence between o and any refined subset of the specification S: if a sequence of operators can be applied in S that leads to a state specification S' still represented by S, then o must be independent of S' as well.

Let O_D be the set of operators dependent on each other or on S_C, and O_I for all other operators. Algorithm 8 introduces a simple approach to create the set of dependent operators. We initialise Λ with the literals in S_C and the set of dependent operators, O_D, to be empty. The algorithm repeats, continually adding all operators dependent on the literals in Λ to O_D until no new operators are added.

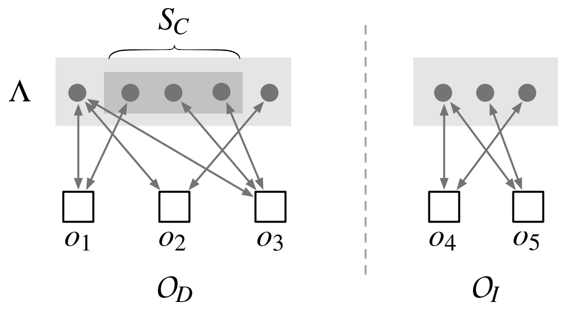


Figure 5.3: Splitting the operator set into those dependent on S_C (O_D) , and those independent (O_I).

We illustrate operator filtering in Figure 5.3 with the set of atoms on the top and the set of operator schemata $o_1 \ldots o_5$ on the bottom. Initially, operators are linked to the atoms in their preconditions and effects producing a two-level graph structure. Operators contained in the same graph as atoms in the conflict specification are considered to be

Algorithm 8: Computing the Independent Operator Set

```
   Input: Conflict specification $S_C$, and list of operator schemata $O$
   Result: The set of dependent operators $O_D$
1  begin
       Initialise the set of dependent literals $\Lambda$ to $S_C$
2      $\Lambda \leftarrow S_C$
       Begin with no dependent operators
3      $O_D \leftarrow \emptyset$
4      repeat
5          continue $\leftarrow$ false
6          for each $o \in O$ do
               If we have not considered the operator already, and if the
               operator is dependent on some literal in $\Lambda$
7              if $o \notin O_D$ and $(lit(o) \cap \Lambda) \neq \emptyset$ then
                   Add to the dependent set
8                  $O_D \leftarrow O_D \cup o$
                   Add the literals to the set of dependent literals
9                  $\Lambda \leftarrow \Lambda \cup lit(o)$
                   Ensure we repeat with the new literals added
10                 continue $\leftarrow$ true
11     until continue is false
12     return $O_D$
13 end
```

dependent and all other operators are independent. Here, the effects of operators in O_D never conflict with those in O_I. We term this independence over all possible action sequences *universal independence*.

Example We revisit the example in Proposition 5.1.4. None of o_1, o_2 or o_3 are universally independent, and therefore none can be excluded during traversal. We present the resulting dependency graph.

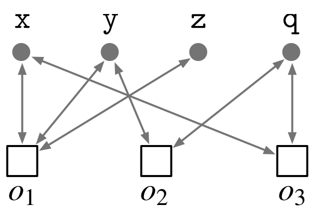


Figure 5.4: Graph identifying universal independence for operators in Proposition 5.1.4.

Consider a further action $\{\mathtt{a},\mathtt{b}\} \xrightarrow{\mathtt{o_4}} \{\mathtt{c}\}$. Here, $\Lambda = \{\mathtt{q},\mathtt{x},\mathtt{y},\mathtt{z}\}$, and since no literal in Λ is referenced by $\mathtt{D}$ we know $O_D = \{o_1, o_2, o_3\}$ and $O_I = \{o_4\}$. Therefore, operator

o_4 can be ignored during traversal. □

Proposition 5.1.5. *Let O_I be the operators universally independent of S_C, and O_D be the remaining operators. If operators O_I are excluded from traversal the norm synthesis algorithm remains sound.*

Proof. Let the operators in a complete conflict run be the plan:

$$\Delta = \langle o_1, o_2, \ldots, o_n \rangle .$$

We split the operator sequence Δ into two subsequences, Δ_I which contains the operators independent of S_C, and Δ_D which contains the dependent operators:

$$\Delta_I = \Delta \setminus O_D \qquad\qquad \Delta_D = \Delta \setminus O_I .$$

Let the result of applying Δ in a state specification S_1 result in S_2 ($Res(S_1, \Delta) = S_2$). If the sequence of independent operators can be applied prior to the dependent operators without altering the effects of the plan then it holds that we can apply all Δ_I and subsequently all Δ_D in S_1 and result in the same specification S_2. We write this as a nested set of functions, $Res\Big[Res(S_1, \Delta_1), \Delta_2\Big]$ and state the equality:

$$Res\Big[Res(S_1, \Delta_1), \Delta_2\Big] = Res(S_1, \Delta).$$

We now show that any independent operator can be reordered before a dependent one. Consider the two operator plan $\langle o_i, o_d \rangle$ where $o_d \in O_D$ and $o_i \in O_I$. Since o_i cannot alter any literals referenced in o_d, the operators can be switched to form the equivalent sequence $\langle o_d, o_i \rangle$. It follows that the sequence Δ_I can always be reordered before Δ_D, and since operators in Δ_I cannot contribute to S_C a dependency exists where reachability holds for Δ if it holds for the subsequence Δ_D. ■

5.1.2.1 Complexity of A Priori Filtering

Let the operator count be $n_o = |O|$ and the number of atoms be $n_a = |\mathcal{A}|$. The computational complexity of constructing the dependency graph is proportional to the number of atoms considered for each operator, where in the worst case each operator references every atom, resulting in the complexity $O(n_o n_a)$. A dependency graph requires nodes for every operator and every atom, with space complexity $O(n_a + n_o)$. Once the preprocessing is complete and the set of operators has been filtered this data structure can be discarded prior to synthesis beginning.

5.1.3 Traversal Optimisation 2: Traversal Pruning

The *traversal pruning* optimisation exploits reachability dependencies of operator sequences at runtime. If a candidate successor operator of a run can be applied at the beginning of the sequence, then the reachability of states in the partial run is dependent on a shorter run that has already been considered. If traversal pruning determines that an operator need not be considered then the search is pruned, removing the need to search all successor operators.

Consider the following incomplete run, $R = S_1 \xrightarrow{\mathtt{o_1}} \ldots \xrightarrow{\mathtt{o_{n-1}}} S_n$. During traversal we identify the successor operators that are partially applicable in S_n. Operator reordering reduces the size of the successor set by discarding operators that are applicable in S_1 so long as the reordering does not alter the effects of the run. A successor operator o for run R can be removed from consideration under the following *pruning conditions*:

1. **Applicable**: The successor operator o is applicable in the first state specification of R. Formally, o is applicable in S_1 if and only if:

   $$pre(o) \cap S_n \subseteq S_1.$$

   All the preconditions of o that exist in S_n must also exist in S_1.

2. **Consistent**: No intermediate operator in the run is dependent on a literal that o affects. If o affects a literal required by a subsequent operator then this operator may no longer be applicable and the run is inconsistent. Formally, consistency is guaranteed if and only if:

   $$S_1 \cap \neg post(o) = \emptyset.$$

   Notice that we determine whether a subsequent operator is dependent purely based on the literals in S_1 rather than investigating each intermediate specification. Consider a subsequent operator dependent on a literal l. If l is not in S_1, then some intermediate operator has l as an effect and l will therefore exist after reordering. If l is in S_1, then it has been added during refinement and some future operator is dependent on l. We consider the second case only and determine consistency by analysing S_1 only.

3. **Preserved Effects**: Assume that o is reordered from some position k to the beginning of the run. For R to be consistent we must preserve the effects of o at k, even though o has been reordered to the beginning of the run. Formally, we

know effects are preserved if:

$$S_n \cap \neg post(o) = \emptyset.$$

That is, no negated effect of o exists in the final state specification of R. Again, we only consider the final specification rather than each intermediate one. If no intermediate operator contradicts the effects of o, then these effects will be preserved in S_n. Our approach allows for an intermediate operator to negate the effects of o if a subsequent operator restores each of the negated effects.

Proposition 5.1.6. *Only runs that are guaranteed to be reachable are excluded if the pruning conditions are satisfied.*

Proof. We write runs simply as a sequence of operators for brevity. Consider an incomplete run that would be generated during the traversal process:

$$R_k = \langle o_1 \dots o_k \rangle.$$

Assume R to be an extension of R_k that is complete, but that is not reachable:

$$R = \langle o_1 \dots o_{k-1}, o_k, o_{k+1} \dots o_n \rangle.$$

No conflict-free alternative exists to R. All other complete runs are shown to be reachable. We now remove o_k from R to construct a new, shorter complete run R':

$$R' = \langle o_1 \dots o_{k-1}, o_{k+1} \dots o_n \rangle.$$

We now present our proof by contradiction. If R' is consistent and reachable, then so is R, and since R' is shorter than R we know that R' will be considered first. Importantly, we terminate the traversal if a run is found to not be reachable. Since R' is considered prior to R, we know that for R to be considered R' must be consistent and reachable.

Let $\overline{R_k} = \langle o_k, o_1 \dots o_{k-1} \rangle$ be the reordered instance of R_k where the operator o_k is moved to the beginning of the run. We show that the conditions and effects of R_k are identical to $\overline{R_k}$ if the operator pruning conditions are met.

- Let $S_1 = first(R_k)$ and $S_n = last(R_k)$. We begin by showing that the reordered operator o_k is applicable in S_1. When considering o_k as a successor operator we refine the preceding specification by adding the literals $L_+ = pre(o) \backslash S_n$ to each specification in the run. Let the refined initial specification be $S_1' = S_1 \cup L_+$. According to pruning condition (1) the remaining literals $pre(o) \cap S_n$ are already present in S_1. It therefore holds that o_k is applicable in S_1'.

S_1	S_n	Consistent	Reason
$\emptyset$	$\emptyset$	Yes	No subsequent operators affect l.
	l	Yes	Reordering o_k has no effect on l since l is already present.
	$\neg l$	No	Inconsistent since a subsequent operator has $\neg l$ as one of its effects. Reordering o_k does not preserve effects.
$\neg l$	-	No	A subsequent operator is dependent on $\neg l$. Reordering o_k results in an inconsistent run.
l	l	Yes	No conflicting effects since l is preserved.
	$\neg l$	No	A subsequent operator negates l by having $\neg l$ as an effect. Reordering results in an inconsistent run.

Table 5.1: Conditions under which operator reordering results in an inconsistent run.

- Next we show that the effects of o_k are preserved after reordering. If reordering o_k has no effect on the run then $last(R_k) = last(\overline{R_k})$. We now show that this holds. Consider each effect literal $l \in post(o_k)$. Table 5.1 details the conditions where inconsistencies occur due to effects not being preserved, depending on whether l or $\neg l$ appear in S_1 and S_n. If neither l nor $\neg l$ are present, we write $\emptyset$.

  There are three conditions (highlighted in grey above) under which an effect is not preserved. Each of these three conditions is shown not to hold since they are eliminated by the pruning conditions (2) and (3). Under these conditions, reordering o_k to the beginning of the sequence of actions does not alter the net effects of the run: $last(R_k) = last(\overline{R_k})$.

Since o_k does not contribute to conflict the reachability of states in $\overline{R}$ and R follow from the reachability of those in R'. Since we know the reachability of this shorter run has already been checked, then the states in R are reachable, and a contradiction is reached. ■

Example Let $S_C = \{\texttt{hold}(\texttt{a}_1,\texttt{Parcel}_1),\texttt{hold}(\texttt{a}_2,\texttt{Parcel}_2)\}$ be the conflict specification where Agents $\texttt{a}_1$ and $\texttt{a}_2$ are not permitted to hold parcels concurrently, where $\texttt{Parcel}_1$ and $\texttt{Parcel}_2$ are unbound and represent *any* parcel in the domain. Suppose that during conflict traversal we initialise the partial run R:

$$\left\{\begin{array}{c}\neg\texttt{hold}(\texttt{a}_1,\texttt{Parcel}_1),\\ \texttt{hold}(\texttt{a}_2,\texttt{Parcel}_2),\\ \texttt{at}(\texttt{a}_1,\texttt{X}),\\ \texttt{parcelAt}(\texttt{Parcel}_1,\texttt{X})\end{array}\right\} \xrightarrow{\texttt{pickup}(\texttt{a}_1,\texttt{Parcel}_1,\texttt{X})} \left\{\begin{array}{c}\texttt{hold}(\texttt{a}_1,\texttt{Parcel}_1),\\ \texttt{hold}(\texttt{a}_2,\texttt{Parcel}_2),\\ \texttt{at}(\texttt{a}_1,\texttt{X}),\\ \neg\texttt{parcelAt}(\texttt{Parcel}_1,\texttt{X})\end{array}\right\}.$$

In this run, Agent $\texttt{a}_2$ is holding $\texttt{Parcel}_2$ and $\texttt{a}_1$ enters conflict by picking up $\texttt{Parcel}_1$ in location `X`. Now consider a successor operator o to be the action $\texttt{move}(\texttt{a}_2,\texttt{Y},\texttt{Z})$ where Agent $\texttt{a}_2$ moves from `Y` to `Z`. We are now interested in the effects that o has on our partial run. During traversal, the pruning optimisation will result in o not being considered with the following filtering conditions:

1. Applicability holds since $pre(o) \cap last(R) = \emptyset \subseteq first(R)$. Here, refinement adds all of $pre(o)$ to $first(R)$ so o is certainly applicable in $first(R)$.

2. Reordering o to the beginning of the run does not affect the applicability of the `pickup` action in the run, since $first(R) \cap \neg post(o) = \emptyset$.

3. Since $last(R) \cap \neg post(o) = \emptyset$ we know that if o is reordered prior to `pickup` then the effects of o are preserved since `pickup` does not negate any effects of o.

Since the effects of o are preserved the operator is ignored for this run. This implies that $\texttt{move}(\texttt{a}_2,\texttt{Y},\texttt{Z})$'s contributions to the conflict run are achievable out of conflict and reachability for any descendent runs need not be checked. $\square$

Traversal pruning significantly improves the traversal process, especially in classical domains where we consider all possible bindings of operators. Traversal pruning removes unrelated operators from consideration, reducing the number of operators considered and the number of runs produced. One side effect of traversal pruning is that operators filtered using the a priori optimisation are also excluded by the traversal pruning optimisation. We still performed the a priori optimisation since it is computationally inexpensive, and reduces the number of operators that are analysed during each iteration of the traversal process.

5.1.3.1 Complexity of Traversal Pruning

Traversal pruning occurs every time a new successor operator is considered for a run. For each run, the complexity of performing the optimisation is proportional to the number of literals in the specification sets. Let $n_l = |L_{\mathcal{A}}|$ be the total number of literals in the domain. The worst computational complexity of traversal pruning is $O(n_l)$ where each specification contains all literals and the complexity of set intersection is linear. The traversal pruning optimisation does not increase the computational complexity of the algorithm, as we already perform set intersections to compute successor specifications. It is therefore an attractive optimisation, since it is effective at reducing the number of runs considered, requires no additional space, and is feasible to implement.

5.1.3.2 Reverse Traversal Pruning

We have only considered reordering operators to the beginning of the run since, for incomplete runs, it makes little sense to reorder elsewhere: the only conflict-free specification in the run is the first. Once a run is complete it is possible to reorder after the final specification which is conflict-free, in which case the run can be discarded.

The benefits of reverse traversal pruning are not as significant as forward pruning since it only removes a single, complete run at runtime. We argue it is still useful, especially when considering ungrounded runs that may be grounded into many instances prior to reachability analysis.

5.1.4 Traversal Optimisation 3: Partial Order Sequencing

Since runs are constructed in a breadth first fashion all sequences of operators are considered independently. Post processing the runs ensures that we do not consider two runs that are identical but it is in our interests to also reduce the number of times we consider different permutations of operators. Consider the sequences of operators in a run as partially ordered where subsets of operators can be applied in any order without altering the effects of the run. The sequencing of these groups still applies in the run, specifying that all runs in one set must be performed before another.

Conflict traversal produces a unique run for every combination of operator orderings, even though the effects of partially ordered operators are identical for all combinations. Since norm synthesis and reachability analysis is not concerned with the actual contents of a run we pick a single possible ordering from the set of all permutations of the operators.

Example Let $S_C = \{\texttt{at}(\texttt{a}_1, \texttt{node}_1), \texttt{at}(\texttt{a}_2, \texttt{node}_1)\}$ be the conflict specification where Agents $\texttt{a}_1$ and $\texttt{a}_2$ occupy $\texttt{node}_1$ simultaneously. Let R_1 and R_2 be two incomplete runs:

$$\begin{aligned} R_1 &= \bigcirc \xrightarrow{\texttt{move}(\texttt{a}_1,\texttt{node}_2,\texttt{node}_1)} \bigcirc \xrightarrow{\texttt{pickup}(\texttt{a}_1,\texttt{parcel}_1,\texttt{node}_1)} \bigcirc \xrightarrow{\texttt{drop}(\texttt{a}_2,\texttt{parcel}_2,\texttt{node}_1)} \bigcirc \\ R_2 &= \bigcirc \xrightarrow{\texttt{move}(\texttt{a}_1,\texttt{node}_2,\texttt{node}_1)} \bigcirc \xrightarrow{\texttt{drop}(\texttt{a}_2,\texttt{parcel}_2,\texttt{node}_1)} \bigcirc \xrightarrow{\texttt{pickup}(\texttt{a}_1,\texttt{parcel}_1,\texttt{node}_1)} \bigcirc \end{aligned}$$

Since the operators `pickup` and `drop` in the runs are independent, both R_1 and R_2 have identical net effects. There is no reason to consider both of these permutations. In this optimisation we identify these independent operators and pick only a single sequential, unique ordering to investigate. □

Identifying order independent operators a priori is not possible in a classical planning representation as we cannot enumerate all possible parameter bindings for each operator schema. Instead, we adopt an online approach that scans operators at runtime to identify partial orderings, and then discards duplicated orderings, selecting a single one to investigate further. In our implementation we select a simple ascending lexicographic ordering of the operator names as the unique ordering to be kept.

Definition 5.1.7. *The operators $o_1 \dots o_n$ are* order independent *if $\forall i, j \in \{1 \dots n\}$ it holds that $\nexists \varsigma$ where:*

$$i \neq j \quad \wedge \quad pre(\varsigma[o_i]) \cap \neg post(\varsigma[o_j]) = \emptyset \quad \wedge \quad post(\varsigma[o_i]) \cap \neg post(\varsigma[o_j]) = \emptyset.$$

Proposition 5.1.8. *An incomplete run $R = S_1 \xrightarrow{\mathtt{o_1}} \dots \xrightarrow{\mathtt{o_{n-1}}} S_n$ constructed during traversal can be removed from consideration if $\exists k < n$ where operators in $\{o_{n-k} \dots o_{n-1}\}$ are order independent, and not lexicographically sorted.*

Proof. Let $\overline{R}$ be the set of runs constructed by considering all permutations of the last k operators in R. Since the last k operators are order independent, we know that each run in $\overline{R}$ is consistent, and has the same effects as all other runs in the set when applied in identical start specifications.

During the traversal process each incomplete run in $\overline{R}$ is iteratively expanded into a set of complete runs. Each of these complete runs begins with the same initial incomplete portion. Since runs in $\overline{R}$ have the same effects it is possible to substitute this initial portion of each complete run with any other run in $\overline{R}$. Using this mechanism we show the state reachability of runs derived from all runs in $\overline{R}$ simply by performing reachability analysis on complete runs derived from a *single* element of $\overline{R}$. Without loss of generality, we select this single incomplete run to be that in which the operators are ordered in an ascending lexicographic fashion. ■

5.1.4.1 Complexity of Partial Order Sequencing

We invoke the partial order sequencing optimisation after every iteration of the conflict traversal produces a new incomplete run R. In the worst case the set of candidate independent runs is proportional to the length of R. Furthermore, since each operator is checked against every other operator in the sequence the computational complexity to evaluate partial order sequencing on a single run is $O(|R|^2)$.

5.1.5 Traversal Optimisation 4: Duplicate Runs

This optimisation identifies and removes duplicate incomplete runs found during traversal, or complete runs once traversal completes. We wish to avoid a specification and operator comparison that ensures that each element of one runs is exactly the same as another, since:

1. During synthesis and reachability analysis we only utilise the first state specification, the last state specification, and the contributing operator.
2. Specifications and operators in runs may contain variables that can be unified to create matching specifications.

Definition 5.1.9. *Two complete runs R_1 and R_2 are* duplicates *if* $\exists\varsigma$ *where:*

1. $\varsigma[\mathit{first}(R_1)] = \varsigma[\mathit{first}(R_2)]$*, the bound first state specifications are equal, and*
2. $\varsigma[\mathit{last}(R_1)] = \varsigma[\mathit{last}(R_2)]$*, the bound last state specifications are equal, and*
3. $\varsigma[R_1[\xrightarrow{1}]] = \varsigma[R_2[\xrightarrow{1}]]$*, the contributing operators are equal.*

The three conditions presented above are based on the components of the runs that are checked during reachability analysis. We plan from the first to the last state specification and synthesise norms using the contributing operators.

Example Consider the two complete runs R_1 and R_2. We abbreviate $\mathtt{parcel_i}$ as $\mathtt{p_i}$ and $\mathtt{node_i}$ as $\mathtt{n_i}$.

$$R_1 = \Big\langle \mathtt{move(a_1,n_2,n_1)},\ \mathtt{pickup(a_1,p_1,n_1)},\ \mathtt{move(a_2,n_1,n_2)} \Big\rangle$$
$$R_2 = \Big\langle \mathtt{move(a_1,n_2,n_1)},\ \mathtt{pickup(a_2,p_1,n_1)},\ \mathtt{drop(a_2,p_1,n_1)},\ \mathtt{pickup(a_1,p_1,n_1)},\ \mathtt{move(a_2,n_1,n_2)} \Big\rangle$$

Even though these two runs contain different operators they are considered equivalent since the first and last specifications of each run are the same, and the contributing operator, $\mathtt{move(a_1,n_2,n_1)}$, is the same in each case. □

5.1.5.1 Complexity of Duplicate Run Removal

The duplicate run optimisation is expensive in terms of space required, since all incomplete runs found to date must be kept for comparison. Let $\mathcal{R}$ be the set of runs kept. In practice, we can reduce the size of $|\mathcal{R}|$ by keeping only the relevant components of each run: the first and final state specifications, and the contributing operator. If space is still a concern we keep only complete runs, trading a reduction in space required for an increase in time required to complete the conflict traversal. Importantly, the resulting set of complete runs will be identical.

Given a set of operators O and literals L we estimate an upper bound to the number of runs. Let $n_l = |L|$ and $n_o = |O|$. From n_l literals we can construct 3^{n_l} possible specifications. A run is composed of a unique first and last specification pair and contributing operator: the maximum number of possible runs is $3^{n_l+1}n_o$, and the corresponding space complexity to store all runs is $O(3^{n_l+1}n_o)$.

Duplicate runs is also the most computationally complex traversal optimisation. Given a candidate run R' we wish to know whether a binding ς exists for any $R \in \mathcal{R}$ such that $R' = \varsigma[R]$. Constructing ς requires us to investigate all possible pairings of literals, which is $O(n_l!)$ for every run in $\mathcal{R}$. The resulting computational complexity of the duplicate run removal algorithm is $O(n_l!3^{n_l+1}n_o)$. Due to the space and time requirements of this optimisation it is invoked after all other optimisations, once the set of runs to operate on is as small as possible.

5.1.6 Traversal Optimisation 5: Repetitive Operators

The repetitive operators optimisation is a means of reasoning about the state reachability of runs containing (possibly infinite) sequences of repeatedly applied operators.

Example Consider the following incomplete runs created by repeatedly applying `pickup` operators in the Parcel Delivery domain:

$$\begin{tikzcd}
\bigcirc \arrow[r, "\texttt{pickup(a}_1\texttt{,P1,node}_1\texttt{)}"] & \bigcirc & & \\
\bigcirc \arrow[r, "\texttt{pickup(a}_1\texttt{,P1,node}_1\texttt{)}"] & \bigcirc \arrow[r, "\texttt{pickup(a}_1\texttt{,P2,node}_1\texttt{)}"] & \bigcirc & \\
\bigcirc \arrow[r, "\texttt{pickup(a}_1\texttt{,P1,node}_1\texttt{)}"] & \bigcirc \arrow[r, "\texttt{pickup(a}_1\texttt{,P2,node}_1\texttt{)}"] & \bigcirc \arrow[r, "\texttt{pickup(a}_1\texttt{,P3,node}_1\texttt{)}"] & \bigcirc
\end{tikzcd}$$

Agent $\texttt{a}_1$ repetitively applies the `pickup` operator, each time with a new unbound variable representing the parcel to be picked up. If $\texttt{a}_1$ is able to pickup parcel `P1`, then they should be able to pick up `P2` and `P3` as well. We justify this intuition by considering the preconditions of the operator $\texttt{pickup(a}_1\texttt{,P3,node}_1\texttt{)}$ in turn:

- $\texttt{at(a}_1\texttt{,node}_1\texttt{)}$ - this holds since Agent $\texttt{a}_1$ is already in $\texttt{node}_1$.
- $\texttt{parcelAt(P3,node}_1\texttt{)}$ - this holds as it is introduced during refinement since parcel `P3` is not mentioned previously in the run.

Consider now the subsequent reachability analysis of this run. If the agent is able to find an alternative, conflict-free plan to pick up a parcel `P1` in $\texttt{node}_1$, then a conflict-free alternative plan exists to pick up `P2` and `P3` as well. In fact, the agent is able to pick up any number of parcels in $\texttt{node}_1$ in a conflict-free manner. □

5.1.6.1 Definitions and Notation

We adopt grammar-like $^+$ symbols to markup repetitively applied operators and literals affected by these operators according to the following definitions:

- A *repetitive operator* is an operator that can be repeatedly applied. We write $\texttt{pickup}^+$ to denote that the operator `pickup` can be applied repetitively.
- A *repetitive parameter* is an unbound parameter of a repetitive operator. Every repetitive operator has at least one repetitive parameter. For example, the operator $\texttt{pickup}^+(\texttt{a}_1,\texttt{P}^+,\texttt{node}_1)$ includes the repetitive parameter $\texttt{P}^+$. We can substitute any unique variable symbol or parcel for $\texttt{P}^+$ to create unique instances of the `pickup` operator that are all applicable.
- A *repetitive literal* is a predicate literal that has at least one repetitive parameter. For example, the repetitive parameter $\texttt{P}^+$ may be referenced in the repetitive literal $\texttt{parcelAt}^+(\texttt{P}^+,\texttt{node}_1)$.

We have avoided implicitly defining repetitive operators based on whether they contain a repetitive parameter. We have chosen to be explicit about the repetitive nature of operators, parameters and literals for clarity.

5.1.6.2 Repetition in Operators

A successor operator o for an incomplete run R is *repetitive* if it does not affect a literal present in $last(R)$. If o alters a literal in $last(R)$ then we say that o *consumes* this literal, since another instance of o cannot certainly be applied immediately after. Literals specifically added to a run during refinement in order to make o applicable *can* be affected by o, since repeated applications of the operator introduces a literal for each consumed. We are interested in the literals already present in R that are affected by o only.

Definition 5.1.10. *Let o be the successor operator considered for a run R, and $L = last(R) \cap pre(o)$ be the subset of o's preconditions already in $last(R)$. Operator o is* repetitive *if either of the following hold:*

1. $L \cap \neg post(o) = \emptyset$, *or*

2. *$\forall l \in (L \cap \neg post(o))$. l is repetitive.*

That is, no literals are affected, or every affected literal is introduced by a previous repetitive operator. The intuition is that if an operator consumes no literals in the last specification then it can be applied repetitively (1), or if it does consume a literal then this literal should be repetitive (2).

Definition 5.1.11. *Let o^+ be a repetitive successor operator for run R with parameters $p_1 \dots p_n$. We define p_i as a* repetitive parameter *if:*

1. *$p_i \in \mathcal{L}_v$. That is, p_i is a variable symbol, and*
2. *$\nexists l \in last(R)$ where l references p_i. That is, the only occurrences of variable p_i are introduced by o^+.*

We identify repetitive operators by considering the binding ς used to instantiate o^+ from its source schema. Every parameter of o^+ that is not contained in the operator's binding ς is considered to be repetitive, since it does not already exist in the run.

Definition 5.1.12. *Every literal that references a repetitive parameter is repetitive.*

Identifying repetitive operators is now fully defined, but it is unclear what the semantics are of a non-repetitive operator that is dependent on a repetitive literal in a run. To this end we introduce repetitive operator instantiation next.

5.1.6.3 Instantiating Repetitive Operators

A repetitive literal is added as an effect of a repetitive operator, hence an arbitrary number of literals can be introduced. Consider the situation where a successor operator is not repetitive but is dependent on a literal introduced by some previous repetitive operator. In this situation we employ an *instantiation* procedure to create non-repetitive instances for each operator that contributes a literal, thereby ensuring that non-repetitive operators are dependent only on literals introduced by other non-repetitive operators.

Example Consider the following incomplete run:

$$\begin{tikzcd} \bigcirc \arrow[r, "\texttt{move}(\texttt{a}_1\texttt{,node}_1\texttt{,node}_2)"] & \bigcirc \end{tikzcd}$$

where Agent $\texttt{a}_1$ moves from $\texttt{node}_1$ to $\texttt{node}_2$. Let $o = \texttt{pickup}(\texttt{a}_1\texttt{,P1,node}_2)$ be the successor operator. The set of existing precondition literals is $L = \{\texttt{at}(\texttt{a}_1\texttt{,node}_2)\}$ and since $L \cap \neg post(o) = \emptyset$ we consider pickup to be repetitive:

$$\begin{tikzcd} \bigcirc \arrow[r, "\texttt{move}(\texttt{a}_1\texttt{,node}_1\texttt{,node}_2)"] & \bigcirc \arrow[r, "\texttt{pickup}^+(\texttt{a}_1\texttt{,P1}^+\texttt{,node}_2)"] & \bigcirc \end{tikzcd}$$

Next, consider successor o to be $\texttt{drop}(\texttt{P1}^+,\texttt{node}_2)$. Since $L = \{\texttt{hold}^+(\texttt{a}_1,\texttt{P1}^+)\}$ then condition (1) of Definition 5.1.10 does not hold. Since $\texttt{hold}(\texttt{a}_1,\texttt{P1}^+)$ is introduced through repetitive operator $\texttt{pickup}^+(\texttt{a}_1,\texttt{P1}^+,\texttt{node}_2)$ then $\texttt{drop}(\texttt{a}_1,\texttt{P1}^+,\texttt{node}_2)$ too is repetitive, and the resulting run is:

$$\begin{tikzcd} \bigcirc \arrow[r, "\texttt{move}(\texttt{a}_1,\texttt{node}_1,\texttt{node}_2)"] & \bigcirc \arrow[r, "\texttt{pickup}^+(\texttt{a}_1,\texttt{P1}^+,\texttt{node}_2)"] & \bigcirc \arrow[r, "\texttt{drop}^+(\texttt{a}_1,\texttt{P1}^+,\texttt{node}_2)"] & \bigcirc \end{tikzcd}$$

Finally, consider operator o to be $\texttt{destroy}(\texttt{a}_1,\texttt{P1}^+)$ that affects a single instance of $\texttt{P1}^+$. Since $\texttt{P1}^+$ is introduced by a repetitive operator it is possible that when this run is implemented in practice that a number of parcels are introduced. In order to invoke the `destroy` operator on just one of these parcels we *instantiate* the repetitive operator `pickup`+ as follows:

$$\begin{tikzcd} \bigcirc \arrow[r, "\texttt{move}(\ldots)"] & \bigcirc \arrow[r, "\texttt{pickup}^+(\texttt{a}_1,\texttt{P1}^+,\texttt{node}_2)"] & \bigcirc \arrow[r, "\texttt{pickup}(\texttt{a}_1,\texttt{P3},\texttt{node}_2)"] & \bigcirc \arrow[r, "\texttt{drop}^+(\texttt{a}_1,\texttt{P1}^+,\texttt{node}_2)"] & \bigcirc \end{tikzcd}$$

We have introduced a new variable P3, of which there is a single instance by inserting the non-repetitive operator $\texttt{pickup}(\texttt{a}_1,\texttt{P3},\texttt{node}_2)$. We can now apply the `destroy` operator to this particular parcel:

$$\begin{tikzcd} \ldots \arrow[r, "\texttt{pickup}(\texttt{a}_1,\texttt{P3},\texttt{node}_2)"] & \bigcirc \arrow[r, "\texttt{drop}^+(\texttt{a}_1,\texttt{P1}^+,\texttt{node}_2)"] & \bigcirc \arrow[r, "\texttt{destroy}(\texttt{a}_1,\texttt{P3})"] & \bigcirc \end{tikzcd}$$

□

Definition 5.1.13. *A non-repetitive operator o is an* instance *of a repetitive operator* o^+ *if:*

1. *o and* o^+ *are produced from the same operator schema,*
2. *every non-repetitive parameter of o is identical to that in* o^+*, and*
3. *every repetitive parameter of* o^+ *is replaced by a unique non-repetitive variable parameter in o.*

We write *non-optimised* to describe a run without any repetitive operators and *optimised* a run with repetitive operators.

Example Consider the repetitive pickup operator from the previous example:

$$\texttt{pickup}^+(\texttt{a}_1,\texttt{P1}^+,\texttt{node}_2).$$

An instance of this operator must also be a `pickup` action involving Agent $\texttt{a}_1$ and $\texttt{node}_2$, but with $P1^+$ replaced with a non-repetitive instance. For example, the subset of possible instances of this operator are:

$$\begin{gathered} \texttt{pickup}(\texttt{a}_1,\texttt{P4},\texttt{node}_2) \\ \texttt{pickup}(\texttt{a}_1,\texttt{Parcel},\texttt{node}_2) \\ \ldots \end{gathered}$$

□

We write *Instantiate*(R, o) as the function that instantiates the repetitive operator o in R. In practice when we instantiate an operator we seek to introduce non-repetitive parameters for each repetitive parameter. Since all variable symbols are represented as parameters in our operators we can be sure that an operator with no repetitive parameters is itself not repetitive.

In this model, repetitive operators can be conditional on literals added by previous repetitive operators. We say that a *dependency* exists between the operators, and if the later operator is instantiated then the prior must be too.

5.1.6.4 Managing Dependencies Between Repetitive Operators

Dependencies between operators imply that runs should be interpreted right to left with dependencies followed during instantiation. For example, consider the run with repetitive `pickup` and `drop` operators presented previously. Each instantiated `drop` action must be preceded by a matching `pickup` action. By processing a run from right to left we ensure that a sufficient number of precursor operators are created for each dependent successor. If we produce 5 `drop` actions, then we know to also produce 5 `pickup` actions. We have modularised our approach into two Algorithms:

1. **Algorithm 9**: Given a run R with non-repetitive operator o, return a new run R' where all repetitive operators that o is dependent on are instantiated. The resulting run R' ensures that o is dependent on no repetitive operators.

2. **Algorithm 10**: Given a run R and a candidate successor operator o, Algorithm 10 returns new instances of R and o where o is appropriately marked to be repetitive and all repetitive dependencies of o are managed through Algorithm 9.

Line	Explanation and Comments for Algorithm 9
2	Identify the set of repetitive literals that o is conditional on.
4	Initialise R' a modified version of R that is subsequently modified and returned.
5	We wish to instantiate all operators that as an effect provide a repetitive literal in L_+. Since this must be performed for every literal we repeat until the set is empty
6	Identify the set of operators O that bring about the literals in L_+. Each of the operators in O are already in the run R. Since they contribute repetitive literals they are in turn repetitive and must be instantiated.
7	O contains all operators that bring about L_+, so we clear L_+ appropriately.

Line	Explanation and Comments for Algorithm 9
8–9	For each of the operators in O we modify the run R' by instantiating a version of o' in the run R'. The instantiated operator is inserted into the run directly after its repetitive ancestor.
10	It may be the case that the instantiated repetitive operator is dependent on other repetitive operators. We repeat the instantiation process by adding all remaining repetitive literals to L_+.
13	If o is dependent on no repetitive literals then R and o can be used for traversal as is.

Algorithm 9 takes as input a non-repetitive operator. It is possible that a successor operator may be classified as repetitive. In this case no manipulation of the run is required since repetitive operators are permitted to depend on other repetitive operators. We therefore append the repetitive operator and continue as usual. The algorithm to determine whether an operator is repetitive is presented in Algorithm 10.

Line	Explanation and Comments for Algorithm 10
2	First, we identify whether o can be applied repetitively, according to Definition 5.1.10.
3–4	If we can apply o repetitively, then we create an annotated version $o+$ of o using the $^+$ notation introduced above to identify all repetitive literals in o. The run R and o^+ can then be used for traversal.
6–7	We have identified as o being a non-repetitive operator, although it may still be dependent on repetitive literals. Here we make a call to Algorithm 9 that returns a modified instance of R where all dependent repetitive operators have been instantiated. We then return the modified run and original operator for further traversal.

5.1.6.5 Generating Non-Optimised Runs

We have already detailed under what conditions operators are considered repetitive, and have provided an algorithm that can be followed to manage the dependencies between these operators. Once conflict traversal concludes a set of optimised complete runs is produced. We require a means of producing non-optimised runs from these optimised runs so that we can perform reachability analysis.

We present *unfolding* as a process to compute a set $\mathcal{R}$ of non-optimised runs from an optimised run $\hat{\mathcal{R}}$. Each of the repetitive operators in $\hat{\mathcal{R}}$ is instantiated a number of times to form a sequence of non-repetitive operators. The set of all runs is infinite since repetitive operators can be unfolded an arbitrary number of times. We adopt a

Algorithm 9: Instantiating Dependent Repetitive Operators

```
Input: The run $R$ and non-repetitive operator $o$, where $o \in R$.
Result: The new run $R'$ with all repetitive operators that $o$ depends on instantiated.
1  begin
       Check whether $o$ is dependent on a repetitive literal.
2      $L_+ \leftarrow RepetitiveLiterals(pre(o))$
3      if $L_+ \neq \emptyset$ then
           $R$ is modified and repetitive operators instantiated
4          $R' \leftarrow R$
5          while $L_+ \neq \emptyset$ do
               Identify operators that create literals in $L_+$
6              $O \leftarrow Operators(L_+)$
7              $L_+ \leftarrow \emptyset$
8              for $o' \in O$ do
                   Instantiate the repetitive operator $o'$
9                  $R' \leftarrow Instantiate(R', o')$
                   Add each of $o'$s repetitive dependencies to $L_+$
10                 $L_+ \leftarrow L_+ \cup RepetitiveLiterals(pre(o'))$
11         return $R'$
12     else
           No dependency on repetitive literals.
13         return $R$
14 end
```

Algorithm 10: Managing Repetitive Successor Operators

```
Input: An incomplete run $R$ with repetitive operators, and successor operator $o$.
Result: The tuple $\langle R', o' \rangle$ where all repetitive dependencies are instantiated in $R'$, and $o'$ is the
        altered operator to be used for traversal.
1  begin
       Begin by identifying if $o$ is repetitive
2      if $o$ is repetitive then
           Annotate $o$ with $^+$ and perform run refinement as usual
3          $o^+ \leftarrow MarkAsRepetitive(o)$
4          return $\langle R, o^+ \rangle$
5      else
           Instantiate all dependencies on other repetitive operators by
           calling Algorithm 9
6          $R' \leftarrow$ call $Algorithm9(R, o)$
7          return $\langle R', o \rangle$
8  end
```

specified bound to limit the length of the unfolded runs as detailed in Section 4.3.7.1. A non-optimised run is *represented* by an optimised run if the non-optimised run can be generated through the unfolding of repetitive operators.

Example Consider the run where Agent $\texttt{a}_1$ moves into $\texttt{node}_2$, performs at least one `pickup` action, and then moves to $\texttt{node}_1$:

$$\begin{tikzcd} \bigcirc \arrow[r, "\texttt{move}(\texttt{a}_1\texttt{,node}_1\texttt{,node}_2)"] & \bigcirc \arrow[r, "\texttt{pickup}^+(\texttt{a}_1\texttt{,P1}^+\texttt{,node}_2)"] & \bigcirc \arrow[r, "\texttt{move}(\texttt{a}_1\texttt{,node}_2\texttt{,node}_1)"] & \bigcirc \end{tikzcd}$$

The set of non-optimised runs produced from this run through repetitive operator unfolding includes:

$$\begin{tikzcd} \bigcirc \arrow[r, "\texttt{move(...)}"] & \bigcirc \arrow[r, "\texttt{pickup}(\texttt{a}_1\texttt{,P1,node}_2)"] & \bigcirc \arrow[r, "\texttt{move(...)}"] & \bigcirc \\ \bigcirc \arrow[r, "\texttt{move(...)}"] & \bigcirc \arrow[r, "\texttt{pickup}(\texttt{a}_1\texttt{,P1,node}_2)"] & \bigcirc \arrow[r, "\texttt{pickup}(\texttt{a}_1\texttt{,P2,node}_2)"] & \bigcirc \arrow[r, "\texttt{move(...)}"] & \bigcirc \\ \bigcirc \arrow[r, "\texttt{move(...)}"] & \bigcirc \arrow[r, "\texttt{pickup}(\texttt{a}_1\texttt{,P1,node}_2)"] & \bigcirc \arrow[r, "\texttt{pickup}(\texttt{a}_1\texttt{,P2,node}_2)"] & \bigcirc \arrow[r, "\texttt{pickup}(\texttt{a}_1\texttt{,P3,node}_2)"] & \bigcirc \ldots \end{tikzcd}$$

□

Unfolding is straightforward so we avoid a full presentation. Given an optimised run $\hat{\mathcal{R}}$ we continually identify repetitive operators to instantiate. For each of these operators we create an instantiated instance by calling the *Instantiate* method detailed in Section 5.1.6.3, and subsequently call Algorithm 9 to ensure that the repetitive dependencies are also instantiated in the run. The process continues until the bound is exceeded.

5.1.6.6 Properties of Repetitive Operators

We present two propositions specifying properties of the repetitive operator optimisation regarding soundness and expressivity of optimised runs.

Proposition 5.1.14. *Using the repetitive operator optimisation results in a set of complete traversal runs from which every non-optimised run is represented.*

Proof. Let $\mathcal{R}$ be the set of complete runs returned by the traversal when the repetitive optimisation is not used, and write $\overline{\mathcal{R}}$ to be the set when the repetitive optimisation is used. If no repetitive operators are found during traversal then $\mathcal{R} \equiv \overline{\mathcal{R}}$ since Algorithm 9 performs no modification of the run, and Algorithm 10 similarly returns the run and successor operator unchanged. Conflict traversal continues as usual and the resulting set of runs contains every non-optimised run as before.

Let us assume now that some runs in $\overline{\mathcal{R}}$ contain repetitive operators. Let $\mathcal{R}^* = \mathcal{R} \cap \overline{\mathcal{R}}$ be the set of non-optimised runs that appear in both traversal sets. We wish to show that every run in $(\mathcal{R} \backslash \mathcal{R}^*)$ is represented by a run in $\overline{\mathcal{R}}$.

We present a proof by contradiction. Let $R \in (\mathcal{R} \backslash \mathcal{R}^*)$ be a run that cannot be represented by a run in $\overline{\mathcal{R}}$. By definition, R must contain repeated operators, otherwise it would be in $\overline{\mathcal{R}}$. Without loss of generality let $O' = \{o_k, o_{k+1}, o_{k+2} \ldots\}$ be the sequence of repetitively applied operators in R. By Definition 5.1.10 we know that none of the operators in O' consume literals in the run and that during traversal we will have constructed an alternative run R' by replacing all O' operators in R with a single repetitive operator o^+. All subsequent operators remain the same. Our proof holds if R is represented by R'. Since the operators in O' do not consume any preceding literals we know we can instantiate instances of o^+ in R', continually refining the run with the additional literals added. By completing this process for each operator in O', and by subsequently removing o^+ from R' we are left with a run that is equivalent to R. A contradiction is reached. ■

An intuitive way to think about repetitive operators is to consider optimised runs as partially expanded. During conflict traversal repetitive operators are identified but not unfolded immediately since we know that they can be unfolded in the future. In fact, the process of unfolding is very similar to run refinement since required preconditions are added to the run for every unfolded operator.

We now show that every optimised run represents an infinite set of non-optimised runs, implying that repetitive operator runs are strictly more expressive than traditional runs and allowing us to reason about infinite sequences of operators.

Proposition 5.1.15. *The repetitive operator optimisation results in more expressive runs since each optimised run represents an infinite set of non-optimised runs.*

Proof. Consider the run $R = S_1 \xrightarrow{\mathtt{o_1}} \ldots \xrightarrow{\mathtt{o_k^+}} \ldots \xrightarrow{\mathtt{o_{n-1}}} S_n$ with a single repetitive operator o_k^+. Furthermore, let O^+ be the set of instantiated, non-repetitive operators derived from o_k^+, where each instance of O^+ is unique. The size of O^+ is infinite, since for every repetitive parameter we can construct a unique, non-repetitive parameter and substitute this during instantiation.

By the definition of repetitive operators specified in Definition 5.1.10, we know that every operator in O^+ can be applied directly after o_k^+ in R. Furthermore, we know that the instances of the repetitive operators do not consume any literals in R. As a result, every operator in O^+ can be inserted into R sequentially after o_k^+, resulting in different runs of arbitrary length. The finite length optimised run *R* therefore represents an infinite set of non-optimised runs. ■

We have introduced a simple means of identifying and tracking which operators in a run can be applied repetitively. There are many benefits to this optimisation:

1. Repetitive operators increase the effectiveness of traversal pruning since an arbitrary number of repeated actions are considered for removal.
2. Successor operators that are instances of previous repetitive operators need not be considered if the results of the repetitive operator is preserved, since the repetitive operator can always be instantiated to provide the successor operator.
3. This succinct representation conserves space since we do not represent arbitrary sequences of operators if they can be represented by a single repetitive operator.
4. In Section 5.2.2 we detail a reachability optimisation that utilises this more succinct representation to produce similarly expressive alternative plans that show state reachability for every run represented by optimised runs.

5.1.6.7 Repeating Compound Actions

One limitation of our repetitive operator approach is we are only able to represent repetitions of single operators rather than sequences of operators. Successor operators that do not consume any literal are considered to be repetitive, however the same can be said for sequences of non-repetitive operators.

Example In the Parcel Delivery domain we can represent an arbitrary number of parcel pickups followed by an arbitrary number of drops using the following run:

$$\begin{tikzcd} \bigcirc \arrow[r, "\mathtt{pickup^{+}(a_1{,}P1^{+}{,}node_1)}"] & \bigcirc \arrow[r, "\mathtt{drop^{+}(a_1{,}P1^{+}{,}node_1)}"] & \bigcirc \end{tikzcd}$$

We change the `pickup` and `drop` operators so that an agent can carry only a single parcel at a time through the introduction of a `holding` atom:

OPERATOR:	`drop(Agent,Node,Parcel)`
PRE:	{`at(Agent,Node)`,`hold(Agent,Parcel)`,**holding**}
POST:	{¬`hold(Agent,Parcel)`,`parcelAt(Parcel,Node)`,¬**holding**}

OPERATOR: `pickup(Agent,Node,Parcel)`
PRE: $\{$`at(Agent,Node),parcelAt(Parcel,Node),`$\neg$**holding**$\}$
POST: $\{\neg$`parcelAt(Parcel,Node),hold(Agent,Parcel),`**holding**$\}$

The previous run is no longer valid, since the `pickup` operator now consumes the atom $\neg$`holding` which is required by any subsequent `pickup`. While $\langle \texttt{pickup}^{+}, \texttt{drop}^{+} \rangle$ is not valid, the sequence $\langle \texttt{pickup}, \texttt{drop} \rangle^{+}$ is. Here, since the `drop` operator reverses the effect of the `pickup` operator on the `holding` predicate the sequence can be applied repetitively. □

We refrain from reproducing all the theory presented above for sequences of actions since the fundamental process is identical to when we consider single actions. Instead of considering single successor operators we consider single *compound* actions constructed at runtime from sequences of operators in the run. A compound action is a single schema representation of a sequence of operators. We construct these compound operators by considering the *net preconditions* and *net postconditions* of a sequence of operators as defined by:

$$\begin{aligned} post(\langle o_1 \rangle) &= post(o_1) \\ post(\langle o_1 \dots o_n \rangle) &= \left[\ post(\langle o_1 \dots o_{n-1} \rangle) \ \setminus \ \neg post(o_n)\ \right] \cup post(o_n) \end{aligned}$$

$$\begin{aligned} pre(\langle o_1 \rangle) &= pre(o_1) \\ pre(\langle o_1 \dots o_n \rangle) &= pre(\langle o_1 \dots o_{n-1} \rangle) \cup \left[\ pre(o_n) \ \setminus \ post(\langle o_1 \dots o_{n-1} \rangle)\ \right] \end{aligned}$$

For example, consider a sequence of `pickup`, `move` and `drop` operators that include the `holding` predicate as detailed in the example above:

$$\begin{tikzcd} \bigcirc \arrow[r, "\texttt{pickup}^{+}(\texttt{a}_1\texttt{,P1}^{+}\texttt{,N1}^{+})"] & \bigcirc \arrow[r, "\texttt{move}^{+}(\texttt{a}_1\texttt{,N1}^{+}\texttt{,N2}^{+})"] & \bigcirc \arrow[r, "\texttt{drop}^{+}(\texttt{a}_1\texttt{,P1}^{+}\texttt{,N2}^{+})"] & \bigcirc \end{tikzcd}$$

A single compound operator that embodies the effects of this sequence is:

OPERATOR: `pickup_move_drop(Agent,`$\texttt{Node}_1$`,`$\texttt{Node}_2$`,Parcel)`
PRE: $\{\texttt{at(Agent,Node}_1\texttt{),parcelAt(Parcel,Node}_1\texttt{),conn(Node}_1\texttt{,Node}_2\texttt{),}\neg\texttt{holding}\}$
POST: $\{\texttt{at(Agent,Node}_2\texttt{),}\neg\texttt{at(Agent,Node}_1\texttt{)}$
$\texttt{parcelAt(Parcel,Node}_2\texttt{),}\neg\texttt{parcelAt(Parcel,Node}_1\texttt{)}\}$

Notice that the above compound operator is repetitive since it no longer affects `holding`. The means by which we incorporate the compound action into a run is identical to before, instead we repeat each operator in the sequence so that we do not need to keep an explicit representation for every possible compound operator.

5.1.6.8 Complexity of Repetitive Operators

Consider a run R with successor operator o. By Definition 5.1.10 the cost of deciding whether an operator is repetitive is independent of the length of the run. If $|L|$ is the number of literals in our domain then the complexity of evaluating whether or not an operator is repetitive is $O(|L|)$. Importantly, this result does not increase the complexity of our algorithm, since the cost of incorporating the successor operator into the run is linear in L too.

The worst case cost of instantiating dependent repetitive operators is $O(|R|)$, since it is possible that for every successor operator every preceding operator must be instantiated. This computational complexity is in line with the complexity of the partial operator ordering optimisation.

5.1.7 Traversal Optimisation 6: Loopback

For completeness we term that the loop detection built into the conflict traversal process as the Loopback optimisation. If any specification appears more than once in a given run, traversal can be terminated.

5.1.8 Traversal Optimisations Summary

We rank traversal optimisations in Figure 5.5 according to their computational complexity and their effectiveness at reducing the state space, and invoke more effective optimisations first in our implementation.

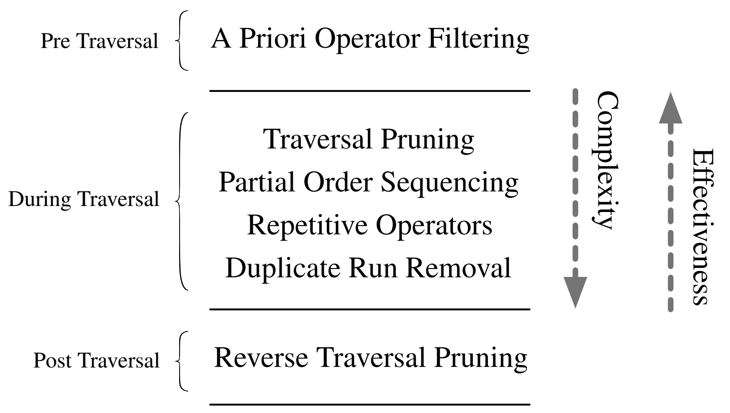


Figure 5.5: Overview of Conflict Traversal Optimisations

Each of these optimisations reduces the number of runs produced during traversal. Next we investigate optimisations invoked during reachability analysis.

5.2 Reachability Optimisations

We present two optimisations that reduce the number of reachability checks performed during reachability analysis, starting with a high level overview of each followed by a more in depth formal presentation.

1. **Planning with Variables**
   Conflict traversal may produce ungrounded conflict runs that are subsequently ground prior to reachability analysis. This optimisation avoids this grounding by assuming variables to be atoms and searching for a single, alternative plan for the unground run.
2. **Reachability with Repetitive Operators**
   Repetitive operators allow us to reason about the reachability of states visited through infinite sequences of operators without enumerating all runs. This optimisation allows us to determine whether every possible non-optimised run represented by an optimised run is reachable based on a simple analysis of only the optimised run.

5.2.1 Reachability Optimisation 1: Planning with Variables

Determining the state reachability of ungrounded runs in a classical domain involves grounding each run. Enumerating groundings can be an expensive process: in the Parcel Delivery domain groundings of runs might enumerate all combinations of nodes in the underlying graph topology. This optimisation makes assumptions about the unground run in order to find a solution plan that applies to all possible groundings, thereby skipping the grounding and planning process entirely.

Example Let $S_C = \{\texttt{at}(\texttt{a}_1,\texttt{Node}_\texttt{Y}),\texttt{at}(\texttt{a}_2,\texttt{Node}_\texttt{Y})\}$ be a conflict specification prohibiting agents $\texttt{a}_1$ and $\texttt{a}_2$ from occupying any node simultaneously. Consider the following complete run:

$$\begin{tikzcd}\bigcirc \arrow[r, "\texttt{move}(\texttt{a}_1{,}\texttt{Node}_\texttt{X}{,}\texttt{Node}_\texttt{Y})"] & \bigcirc \arrow[r, "\texttt{move}(\texttt{a}_2{,}\texttt{Node}_\texttt{Y}{,}\texttt{Node}_\texttt{X})"] & \bigcirc\end{tikzcd}$$

with unbound variables $\texttt{Node}_\texttt{X}$ and $\texttt{Node}_\texttt{Y}$ (and the constraint that $\texttt{Node}_\texttt{X} \neq \texttt{Node}_\texttt{Y}$). An alternative plan can be constructed by reordering the sequence of existing operators:

$$\begin{tikzcd}\bigcirc \arrow[r, "\texttt{move}(\texttt{a}_2{,}\texttt{Node}_\texttt{Y}{,}\texttt{Node}_\texttt{X})"] & \bigcirc \arrow[r, "\texttt{move}(\texttt{a}_1{,}\texttt{Node}_\texttt{X}{,}\texttt{Node}_\texttt{Y})"] & \bigcirc.\end{tikzcd}$$

The information about the topology of the graph is implicit in the conflict run: the fact that $\texttt{Node}_\texttt{X}$ and $\texttt{Node}_\texttt{Y}$ are adjacent allows us to construct an alternative plan without requiring any more domain specific information. □

Proposition 5.2.1. *Let R be a conflict run containing variable symbols and σ being a substitution containing mappings for each variable symbol in R to a unique fresh constant symbol. If state reachability can be shown for the grounded run $\sigma[R]$ then it holds for all possible groundings of R.*

Proof. Let $\overline{\Delta}$ be the alternative plan for $\sigma[R]$. We prove that $\overline{\Delta}$ is a viable alternative plan for any possible grounding of R by mapping each unique constant symbol present in $\overline{\Delta}$ and σ back to its variable symbol. In a sense we are reversing the σ mapping.

Let Δ represent the unground instance of $\overline{\Delta}$ where Δ is identical to $\overline{\Delta}$ except that for all substitution pairs $(v \leftarrow c) \in \sigma$, every occurrence of c is replaced by v. Since the constant symbols are unique we are guaranteed that this reverse mapping exists, and that no non-substituted constant symbols will be changed inadvertently. Δ is a reachable plan for R containing unground actions. For any possible substitution σ' the ground plan $\sigma'[\Delta]$ is a reachable alternative for $\sigma'[R]$. ■

The simplicity of this optimisation should not detract from its effectiveness. By avoiding the grounding of runs in a domain we save a substantial amount of computation. Naturally, this optimisation is not guaranteed to work in all cases, particularly when additional domain knowledge is required to find an alternative plan. In these fail cases we continue with reachability analysis as before.

5.2.1.1 Complexity of Planning with Variables

Planning with variables requires an additional planning step for every complete run generated, prior to the run being grounded. The classical planning problem is PSPACE complete. While planning in general is expensive we point out that the complexity of the initial and goal state specifications for the complete run is typically minimal, so we expect the resulting planning process to fail abruptly if no viable solution is found. Furthermore, while we are invoking an additional planning step, we are potentially saving many more subsequent invocations.

5.2.2 Reachability Optimisation 2: Repetitive Operators

The final reachability optimisation takes advantage of the succinct repetitive operator representation introduced in Section 5.1.6. By producing runs with repetitive operators we are able to synthesise conflict-free alternatives that utilise repetitive operators to reduce the number of reachability checks performed.

Example Consider the conflict specification $S_C = \{\texttt{at}(\texttt{a}_1,\texttt{node}_1),\texttt{at}(\texttt{a}_2,\texttt{node}_1)\}$ in the Parcel Delivery domain. One conflict run produced during traversal is:

$$R = \bigcirc \xrightarrow{\texttt{move}(\texttt{a}_1,\texttt{node}_2,\texttt{node}_1)} \bigcirc \xrightarrow{\texttt{pickup}^+(\texttt{a}_1,\texttt{P}^+,\texttt{node}_1)} \bigcirc \xrightarrow{\texttt{move}(\texttt{a}_2,\texttt{node}_1,\texttt{node}_2)} .$$

In this run Agent $\texttt{a}_1$ moves into location $\texttt{node}_1$, picks up a number of parcels (represented by repetitive parameter $\texttt{P}^+$) and subsequently Agent $\texttt{a}_2$ moves to $\texttt{node}_2$. The run below is an alternative plan for every run represented by R:

$$R = \bigcirc \xrightarrow{\texttt{move}(\texttt{a}_2,\texttt{node}_1,\texttt{node}_2)} \bigcirc \xrightarrow{\texttt{move}(\texttt{a}_1,\texttt{node}_2,\texttt{node}_1)} \bigcirc \xrightarrow{\texttt{pickup}^+(\texttt{a}_1,\texttt{P}^+,\texttt{node}_1)} .$$

Agent $\texttt{a}_2$ moves out of conflict prior to Agent $\texttt{a}_1$ entering $\texttt{node}_1$. Reachability with repetitive operators constructs optimised conflict-free alternative runs as above so that no unfolding of repetitive runs is required. □

For each optimised complete run we construct an unoptimised version by unfolding all repetitive operators out a minimum number of times. This produces the shortest run that is represented by the optimised run. Next, we plan for a conflict-free alternative to this run, and analyse this plan to determine whether the same set of repetitive operators exist. If the operators exist and can still be repetitively applied then state reachability for all represented runs is guaranteed and the process terminates. If this is not the case, then reachability analysis continues as per usual and no improvements are made.

Proposition 5.2.2. *Let R be a complete run with repetitive operators and $\overline{R}$ be a modified version of R with each repetitive operator replaced by one of its instances. Let $\overline{\Delta}$ be a conflict-free alternative plan for the run $\overline{R}$.*

Furthermore, let O^+ be the set of repetitive operators in R, and $\overline{O}$ be the set of instances of these operators in $\overline{R}$. Let the following conditions hold $\forall o_i \in \overline{O}$:

1. *$o_i \in \overline{\Delta}$.*
2. *o_i can be repetitively applied in $\overline{\Delta}$ (by Definition 5.1.10).*
3. *the repetitive instance o_i^+ of o_i is in O^+.*

Finally, let Δ be the modified instance of $\overline{\Delta}$ where each $o_i \in \overline{O}$ is marked repetitive. The plan Δ guarantees state reachability for every run represented by R.

Proof. We present a proof by contradiction. Assume that Δ does not guarantee the state reachability of runs represented by R. This can only be the case if $\mathit{first}(\Delta) \neq \mathit{first}(R)$ or $\mathit{last}(\Delta) \neq \mathit{last}(R)$.

We know that $\mathit{first}(\overline{\Delta}) = \mathit{first}(\overline{R})$ and $\mathit{last}(\overline{\Delta}) = \mathit{last}(\overline{R})$. Furthermore, we know that by substituting every instantiated operator in $\overline{O}$ with its corresponding repetitive operator from O^+ in $\overline{R}$ results in R (since this is the process that generates $\overline{R}$). We show that by performing a similar substitution in $\overline{\Delta}$ we produce Δ where the first and last specification are equal.

We know by (1) above that an instance of every repetitive operator appears in $\overline{\Delta}$, and by (2) that each instance in $\overline{\Delta}$ can be applied repetitively. We can therefore construct Δ and substitute appropriate repetitive operators for instances. Condition (3) above guarantees that the repetitive operators match exactly, implying that the exact same literals are added to the first and final specifications during refinement.

From (1) and (2) we know that Δ must have the same set of repetitive operators as R. From (3) we have shown that the net preconditions and net postconditions of these repetitive operators are also identical to those in R. That is, since reachability holds for states visited by $\overline{\Delta}$, and since the literals added when creating Δ are identical to those added to $\overline{R}$ to create R we know that both $\mathit{first}(\Delta) \neq \mathit{first}(R)$ and $\mathit{last}(\Delta) \neq \mathit{last}(R)$ must hold, and a contradiction is reached. ■

5.2.2.1 Complexity of Reachability with Repetitive Operators

The worst case complexity of checking reachability with repetitive operators is PSPACE complete, due to the invocation of the planner on the grounded run $\overline{R}$. However, considering that the planner would be invoked on all possible instantiations of R if the optimisation fails, we consider the additional planning step for this optimisation as minimal extra effort for a potentially large benefit.

5.2.3 Reachability Optimisation Summary

Both of the reachability optimisations presented are invoked prior to grounding, and both reduce the number of reachability checks performed, but since Planning with Variables is computationally simpler we invoke this first.

The most important consideration when adopting reachability optimisations is the complexity of planning in the underlying domain. While the pre-grounding optimisations are beneficial in systems where planning can be performed very quickly, post-grounding optimisations are not. The fundamental question is therefore whether it is more efficient to employ an optimisation to avoid the reachability check for a single run, or simply to perform the reachability check regardless.

5.3 Optimisations Summary

We presented two classes of optimisations that can be used to improve conflict-rooted synthesis performance:

1. **Traversal Optimisations**: Reduce the number of runs produced during traversal, and the resulting complete runs produced as output, by taking advantage of dependencies between operators to ensure that the state space searched is reduced.

2. **Reachability Optimisations**: Reduce the number of reachability checks required by attempting to show that reachability holds for the unground runs produced, prior to grounding and subsequent planning.

While the optimisations are sound, the question remaining is how effective they are in practice. To this end we present essential details of our implementation of conflict-rooted synthesis next, and follow this with an in depth investigation of its performance when applied to a set of benchmark planning domains.

Chapter 6

CRS Architecture and Design

We have presented conflict-rooted synthesis as a theoretically sound approach to norm synthesis, and we detail the benefits of our approach in Section 7.5.1. We are interested not only in theoretical advantages, but also in how significant these benefits are in practice. In this chapter we present details regarding the design and implementation of our approach, split into three sections:

1. Details regarding the conflict-rooted synthesis approach and the associated optimisations.
2. Details on the integration of the model checking approach into the evaluation framework.
3. Details of the evaluation framework.

The CRS source code, as well as the integration and testing framework are provided with the archived copy of this thesis. Additionally, all code is available upon request from the University of Edinburgh's Agents Group web site (Agents Group, 2011).

6.1 CRS: A Conflict-Rooted Synthesis Implementation

Our implementation of the conflict-rooted synthesis algorithm is called CRS, and is broadly composed of a PDDL parser and processor, and an implementation of the conflict traversal, norm synthesis and reachability analysis procedures. For efficiency purposes our traversal process utilises a succinct data structure called traversal graphs. We present the implementation details of this data structure next, followed by a discussion of how we produce randomly generated conflict specifications, and additionally comment on mechanisms employed to ensure consistent execution results. Finally, we document the changes made to FF to support the batch processing of planning problems.

CRS is implemented in the Java language, apart from the modifications made to the native FF planner. CRS represents a proof of concept implementation sufficient to detail the benefits of our approach, yet significant gains could be made through an improved reimplementation. Currently, CRS totals over 40000 lines of Java code and 15000 lines of modified C code. It ensures accuracy in repeated empirical tests by managing the order in which traversal runs are created in order to produce identical results. This simplifies our empirical evaluation, since the only performance deviation in the computation of CRS is attributed to the underlying operating system.

6.1.1 Architecture and Design

The architecture of CRS closely follows the presentation of the theory detailed in Chapters 4 and 5 and is illustrated in Figure 6.1. Control flows between four core modules that implement parsing, conflict traversal, norm synthesis and reachability analysis. Each module is composed of a set of components, and produces artefacts that follow as input to subsequent components. As input, CRS takes textual representations of the PDDL domain and conflict specification, and as output produces a positive result if norms are synthesised than ensure goal reachability, and a negative result if goal reachability is not guaranteed.

6.1.1.1 Parsing Module

The CRS parsing module is responsible for parsing textual input into a model that can subsequently be used programmatically. The module is composed of three components: a PDDL domain file parser, a PDDL problem file parser, and a conflict specification parser, and produces a model of the planning domain and a model of the conflict specification. All three parsing components are generated from a grammar that specifies the PDDL language and generates Java source code capable of recognising individual language fragments. Actions are associated with the produced recogniser in order to populate the models that are eventually produced as output.

CRS's parsers are programmatically produced using the ANTLR parser generation tool (Parr, 2007) and a modified version of Zeyn Saigol's PDDL grammar (Saigol, 2011). The grammar was extended to allow for the parsing of state specifications containing variable terms. CRS extends Zeyn Saigol's PDDL model to allow the mod-

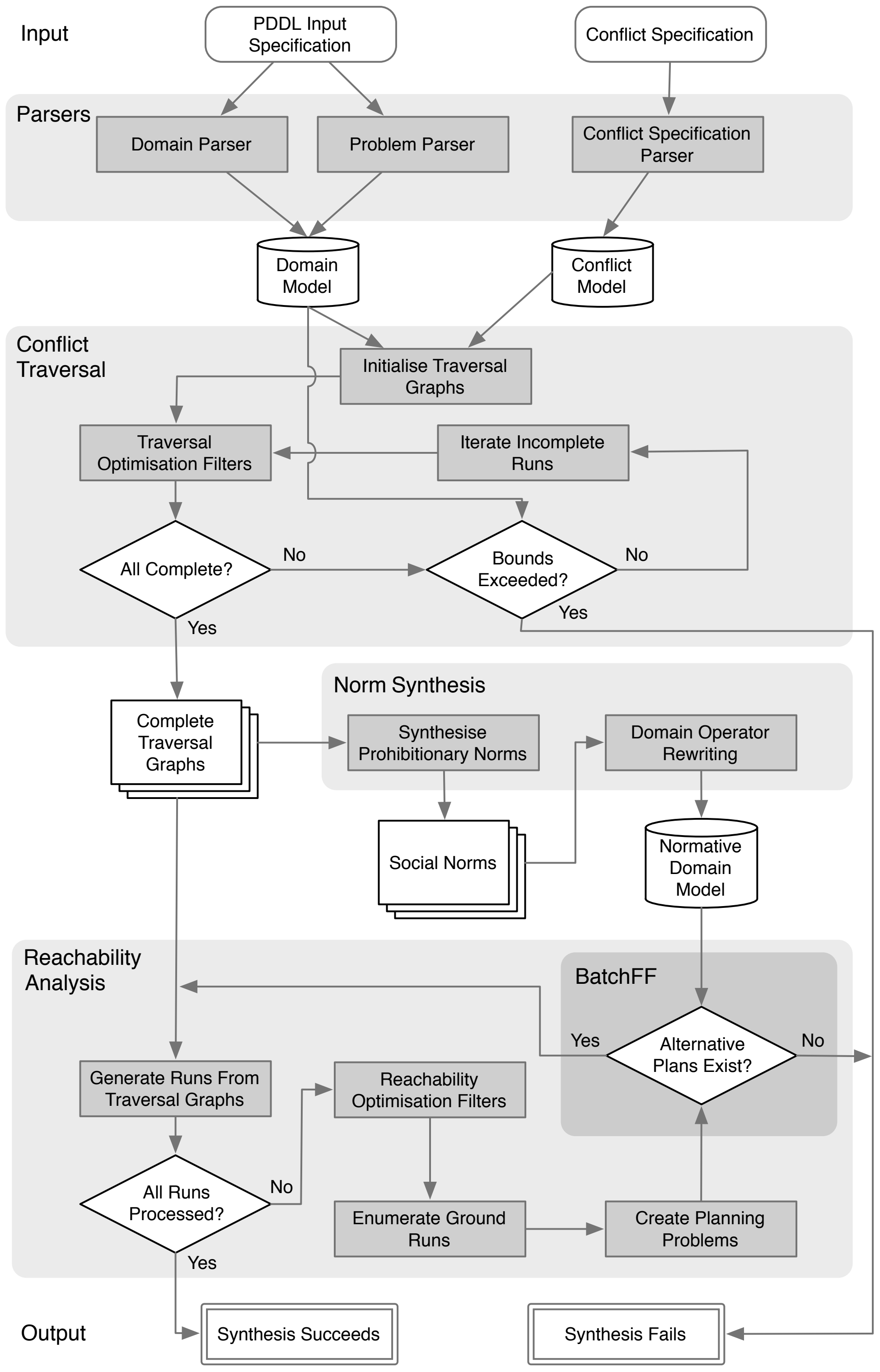


Figure 6.1: The flow of control between modularised components in CRS.

elling of unground state specifications, as well as the application of partially applicable operators. Furthermore, the model was extended to provide support for the representation of constraint sets, substitutions and bindings as detailed in Section 4.3.2. The resulting models provide a structured, programmatic domain representation upon which our abstract search algorithm is implemented.

6.1.1.2 Conflict Traversal Module

Conflict traversal takes the parsed domain and conflict specification models as input, and performs the abstract search of the specification space. Although central to the theoretical presentation of this work, the conflict traversal module does not utilise a set of runs as the core data structure due to the redundancy involved in persisting many, very similar runs. Instead, *traversal graphs* are equivalent graph-based representations that reduce the redundancy significantly, resulting into less space required to represent the same traversal search. We present details of this data structure in Section 6.1.2.

Conflict traversal begins with an initialisation component that takes the parsed models as input and produces a set of traversal graphs as output. The traversal graphs represents all runs of length 2, where all operators are contributing operators. An iterative process then follows. First the traversal graphs are pruned by invoking each of the traversal optimisations in order to discard represented runs. Next, traversal completes successfully if no incomplete runs are represented by the traversal graphs. If incomplete runs exist, a second check is performed in order to identify whether the traversal bounds (as proposed in Section 4.3.7.1) have been exceeded. If the bounds are exceeded traversal terminates in failure, otherwise the traversal graphs are iterated and the process continues. The domain model is taken as input into the bound checking as the specified limits may be proportional to the size of the domain.

A key requirement for the efficient implementation of the conflict traversal optimisations is the ability to identify duplicate runs, according to the conditions outlined in Section 5.1.5. Runs are characterised by their first and last state specifications, and first contributing operators. If a binding exists from a new run to a stored run then the runs are considered duplicates. However, storing and enumerating all runs generated during traversal is expensive, since identifying the existence of a binding for every member requires expensive search. In order to efficiently identify duplicates we adopt signatures for runs in order to identify smaller sets of candidate matching runs. Candidate runs must contain the same number of literals in the first and last specifications, identical counts for each predicate symbol and have contributing operators with the same operator symbol.

At every point in our implementation bindings and constraint sets are kept consistent. We utilise a graph structure called a *constraint graph* to simplify this, with nodes representing variable and constant symbols. Nodes connected by a binding edge are unified, while those connected by a constraint edge cannot be. New bindings or constraints must comply with the restrictions depicted by the graph: bindings cannot be created between two nodes where constraint edges exist, and constraints cannot be added if a binding exists.

6.1.1.3 Norm Synthesis Module

Conflict traversal produces a set of traversal graphs that represent all complete runs found. Norm synthesis takes these graphs as input and produces a set of prohibitionary social norms as output. For the purpose of norm synthesis the runs represented by traversal graphs are enumerated, and the corresponding norms produced for each run. Once complete, the norms are again utilised to produce a model of the normative domain: the domain where norm-abiding behaviour is regimented. Operators contained within the domain model are rewritten as described in Section 4.3.6.1, and the resulting domain model is produced as input for reachability analysis checking.

6.1.1.4 Reachability Analysis Module

Reachability analysis takes as input the set of traversal graphs representing all complete runs, and the model of the normative domain. As with the conflict traversal module, an iterative approach is taken. Given the traversal graphs a set of complete runs are generated from the graphs. For the purposes of this discussion note that this set could contain a single run, but for efficiency purposes in practice it may contain multiple runs. If a complete run exists that has not been checked it is passed to the reachability optimisation component that decides whether the given run can be safely discarded. Discarded runs are removed from the set runs.

A grounding step is then required. Recall that traversal graphs may represent unground runs, and the reachability optimisations typically utilise this more abstract representation to discard sets of complete, grounded runs. Should the reachability optimisations not discard a run, it must be grounded to produce a set of runs to be checked. As discussed in Section 4.1.3.3, for each grounded run a unique planning problem is constructed and the external planner is invoked to find an alternative conflict-free plan for each run. Should any of these checks fail, the synthesis procedure terminates in failure. Additionally, the process fails if planning time limits are exceeded. If all runs are shown to have conflict-free alternatives, the process continues until no more complete runs can be generated from the traversal graphs. If this is the case, CRS terminates and is successful and the social norms can be utilised safely.

When grounding, simplified representations for the runs are created according to the run's constraint graph. All variable symbols bound to a constant symbol are replaced with the constant symbol, while all variables bound to variable symbols are replaced with an arbitrary variable symbol. The result is a run representation that contains the minimal number of unground variable symbols to be ground.

Reachability analysis utilises a modified version of the FF planner called BatchFF which allows for the more efficient batch processing of multiple planning problems. The motivation and implementation details behind BatchFF are presented in Section 6.1.3.

6.1.2 Traversal Graphs

Throughout the formal presentation of our norm synthesis algorithm we have utilised sets of runs to quantify what agents might achieve in conflict. For each successor operator considered during traversal we create a run by appending the successor operator to the original run. The copying of the original run to an independent version is required since modifications made through refinement to one run should not alter any other runs. A run-based representation is intuitive for discussion purposes, yet has two downsides in practice:

1. Runs are *not space efficient* since a complete copy of the original run is created for each successor operator. A more efficient representation would allow runs to share their common components, to reduce the space required to represent the set of complete runs.

2. Copying entire runs is *computationally expensive* since the complexity is proportional to the length of the run and is invoked on every iteration of traversal.

We require a data structure that succinctly represents the space of possible runs. Adopting standard planning graphs, where nodes represent state specifications and edges between nodes represent operator applications, is not sufficient for our purposes. Consider how a planning graph changes under run refinement where literals are added to existing state specifications in the graph, which in turn affect the representation of every run specified.

We present *traversal graphs* as a means of achieving a succinct representation for generated runs without unnecessary duplication of the state or operator representations. These graphs are simple to modify, and crucially, many of our optimisations can be applied directly to the graph. In practice, the benefits of traversal graphs are significant: the space saved by reducing redundancy and the computational effects of less structure duplication is presented next, after the theoretical outline.

6.1.2.1 Definition of Traversal Graphs

An acyclic graph is a tuple $\mathcal{G} = \left\langle \mathcal{V}, \mathcal{E} \right\rangle$ composed of a set of vertices $\mathcal{V}$ and set of pairs $\mathcal{E}$ representing directed edges between nodes. We write $v \in \mathcal{V}$ to refer to a specific vertex v, and $e = (v_1, v_2)$ to refer to the directed edge $e \in \mathcal{E}$ originating from vertex v_1 and terminating in vertex v_2. The graphs we construct have tree-like structures.

Traversal graphs are an annotated form of acyclic graph, where vertices are labelled as state specifications and edges continue to represent action-based transitions. We incorporate the notion of partial operator applicability by defining that an edge is labelled with an operator that is partially applicable in the specification modelled by the vertex it originates from. Importantly, we annotate the edge not only with the operator, but also the literals added to the precursor specification during refinement. A traversal graph is a tuple composed of an acyclic graph and two labelling functions:

$$\mathcal{TG} = \left\langle \mathcal{V}, \mathcal{E}, l_v, l_e \right\rangle$$

where:

- $\mathcal{V}$ and $\mathcal{E}$ are the vertices and edges as above,
- l_v is a labelling function linking each vertex in $\mathcal{V}$ with a state specification composed over the atoms in our state representation language, and
- l_e is an edge labelling function linking each edge in $\mathcal{E}$ with a tuple $\langle a, L_+ \rangle$ where a is an action, and L_+ is is a set of literals.

An edge in a traversal graph is annotated using an action a, a ground instance of an operator in O, and by L_+, the set of literals that must be added to the precursor state specification during refinement so that a is fully applicable. We write $S(v)$ as a shorthand to represent the specification referenced by vertex v, and we write $a(e)$ and $L_+(e)$

to represent the action of edge e and the literals to be added during refinement respectively. Finally, when illustrating these edges we write the action name above the edge, and the set $L_+(e)$ below, as in Figure 6.2, meaning that a transition exists from *a subset* of the states represented by v_1 to all states represented by v_2.

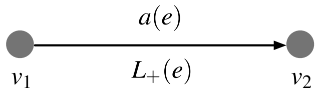


Figure 6.2: An annotated edge $e = (v_1, v_2)$ with action $a(e)$ and refined literals $L_+(e)$.

Example Suppose we wish to represent the following set of runs using a traversal graph:

$$R_1 = \bigcirc \xrightarrow{\texttt{move}(\texttt{a}_1,\texttt{node}_1,\texttt{node}_2)} \bigcirc$$
$$R_2 = \bigcirc \xrightarrow{\texttt{move}(\texttt{a}_1,\texttt{node}_1,\texttt{node}_3)} \bigcirc$$
$$R_3 = \bigcirc \xrightarrow{\texttt{pickup}(\texttt{a}_1,\texttt{parcel}_1,\texttt{node}_1)} \bigcirc$$

All three of the above runs are partially applicable in states represented by the specification $S = \{\texttt{at}(\texttt{a}_1,\texttt{node}_1)\}$, where each run is applicable from a mutually exclusive subset of S. Conflict traversal produces the following initial states for each of the runs:

$$\begin{aligned} R_1[1] &= \{\texttt{at}(\texttt{a}_1,\texttt{node}_1),\texttt{conn}(\texttt{node}_1,\texttt{node}_2)\} \\ R_2[1] &= \{\texttt{at}(\texttt{a}_1,\texttt{node}_1),\texttt{conn}(\texttt{node}_1,\texttt{node}_3)\} \\ R_3[1] &= \{\texttt{at}(\texttt{a}_1,\texttt{node}_1),\texttt{parcelAt}(\texttt{parcel}_1,\texttt{node}_1)\} \end{aligned}$$

Conflict traversal would traditionally consider each of these runs to be entirely different, and continue traversal independently of each. We represent this diagrammatically in Figure 6.3, where each run originates from a subset of the states represented by S.

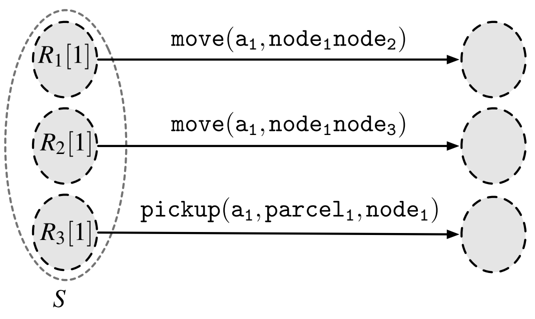

$R_1[1]$
move(a$_1$,node$_1$node$_2$)
$R_2[1]$
move(a$_1$,node$_1$node$_3$)
$R_3[1]$
pickup(a$_1$,parcel$_1$,node$_1$)
S

Figure 6.3: Three runs originating from subsets of a common specification S.

Figure 6.4 presents a traversal graph that represent these runs. Here, the successor state specification S_1, S_2 and S_3 are equal to the final specifications in each of the runs, $last(R_1)$, $last(R_2)$ and $last(R_3)$ respectively. Even though S is not equivalent to

$\textit{first}(R_1), \textit{first}(R_2)$ or $\textit{first}(R_3)$, these first specifications can be constructed if required.

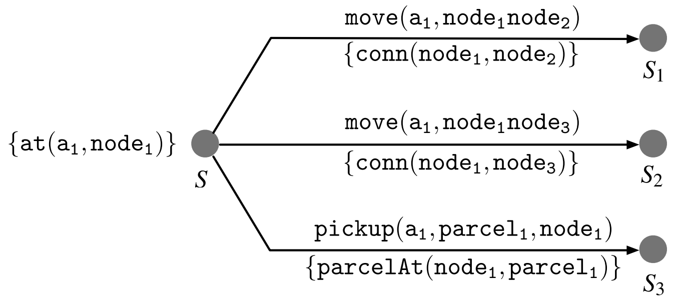


Figure 6.4: A traversal graph representing runs R_1, R_2 and R_3. □

6.1.2.2 Constructing Traversal Graphs

Adapting conflict traversal to utilise traversal graphs requires minimal modifications to our existing algorithms. We present a high level description of the graph creation process in the terms of conflict traversal initialisation and iteration.

6.1.2.2.1 Initialisation We require traversal graphs that represent the initialised runs $S_P \xrightarrow{o_c} S'_C$, where S_C is the conflict specification, S'_C the refined specification, o_c a contributing operator and S_P the precursor specification. We construct a graph with vertices $\mathcal{V} = \{v_1, v_2\}$ connected with edges $\mathcal{E} = \{e\}$. We define the labelling functions as follows:

$$\begin{array}{ll} a(e) = o_c & S(v_1) = S_P \\ L_+(e) = \emptyset & S(v_2) = S'_C \end{array}$$

Similarly, we construct an initialised traversal graph for every contributing operator.

6.1.2.2.2 Iteration We identify the set of successor operators for each conflict specification represented as a leaf node of a traversal graph. Formally, given a traversal graph $\mathcal{TG}$ we define the set of incomplete vertices $\mathcal{V}_I$ as:

$$\mathcal{V}_I = \{v \in \mathcal{V} \mid S(v) \models S_C \wedge \nexists v_2.(v, v_2) \in \mathcal{E}\}$$

An incomplete vertex represents a conflict specification and has no edges traversing from it to other vertices. For each incomplete vertex $v_I \in \mathcal{V}_I$ we identify the set of partially applicable operators as before using the function $O_{par}(S(v_I))$: we are interested in all partially applicable operators from the specification represented by the vertex v_I. For each operator o we identify the successor state S_S and the set of literals L_+ added during refinement, however we do not refine $S(v_I)$ with the literals in L_+. Instead, we create a vertex v_S and the corresponding edge $e = (v_I, v_S)$ with the labels:

$$S(v_S) = S_S \qquad L_+(e) = L_+ \qquad a(e) = o.$$

As it is inefficient in practice to continually search the entire graph for incomplete vertices we instead maintain a list of references to these vertices. We terminate when there are no longer any incomplete vertices.

6.1.2.3 Generating Runs from Traversal Graphs

Traversal graphs can be constructed during traversal and subsequently utilised during reachability analysis. In order to check the reachability of states in runs represented by a traversal graph we require a means of enumerating these runs. Similar to incomplete vertices, we present the set of complete vertices of a graph as:

$$\mathcal{V}_C = \{v \in \mathcal{V} | \nexists v_2.(v, v_2) \in \mathcal{E}\}.$$

Complete vertices are simply leaf nodes of the graph. Given a traversal graph $\mathcal{TG}$ we generate a set of runs $\mathcal{R}$, where for each complete vertex we produce a single run by traversing from the leaf node to the root of the tree, adding specifications and operators as we progress. For each complete vertex $v \in \mathcal{V}_C$ we initialise the run $R = S(v)$ with a single specification and no operators. We keep account of the literals L that are added during each step of the process, where L is initially the empty set. Let the precursor vertex to v be v_P connected by the edge:

$$e = (v_P, v).$$

We have identified the transition between the relevant specifications and now update the set of literals by appending the refinement literals of the edge e to L:

$$L = L_+(e) \cup L.$$

We can now create the precursor specification S_S by taking the specification represented by v_P and adding the literals L to form the new specification:

$$S_S = S(v_P) \cup L.$$

Here, S_S is more specific than $S(v_P)$, and represents the subset of states represented by $S(v_P)$ where the operator for edge e is fully applicable. We complete this step of the iteration by appending the new specification and operator to R as follows:

$$R = S_S \xrightarrow{\mathsf{a(e)}} R.$$

If v_P is a root node with no incoming edges then we have completed iterating up the tree and can add the produced run R to the set of runs $\mathcal{R}$.

We now discuss the implications and benefits of utilising traversal graphs:

- Traversal graphs are a more *concise* representation. Runs in the set represented by a graph are inherently similar, in that they share state specifications and operators. One need only think of two runs that deviate in their final actions to realise that storing an independent representation for each of these runs is inefficient. Traversal graphs allow us to reduce this redundancy.
- Creating traversal graphs is more computationally *efficient*. New operators and specifications can be added to the graph without having to clone any of the data, reproduce state specifications or duplicate entire runs.

We conclude our discussion of traversal graphs by emphasising that both a run-based and traversal graph-based approach to conflict-rooted synthesis are identical. Traversal graphs represent no runs that would not originally be found during a run-based traversal.

6.1.3 Batch FF

Conflict-rooted synthesis utilises the FF planner to perform reachability analysis of synthesised runs, allowing us to harness automated planning techniques directly in our approach. Since the input to the planner is standard PDDL it is possible to replace the planner with any other planner that utilises the same input specification language.

The primary concern when selecting a planner is *efficiency*, not only in planning time but also in planner initialisation time. Since it is common for reachability analysis to solve many planning problems, there is merit in having a planner capable of batch processing planning problems as continually re-instantiating the planner for each new planning problem requires considerable resources. On our test machine a native process takes 30ms to initialise: to solve 10000 planning problems requires 5 minutes of processor time simply to initialise the planner process (without even performing any planning).

We chose the Fast Forward (FF) Planner originally proposed by Hoffmann and Nebel (2001) since it is a well understood planner with an accompanying stable and efficient implementation. Furthermore, it is well regarded in the literature and a good performer on the classical domains used in this thesis. Our adaptation of FF, called BatchFF, adds the ability to batch process planning problems without repeated process instantiation. To support this the planner was modified to perform complete memory management of internal data structures, since the assumption that the planner terminates once a problem is solved no longer holds. BatchFF incorporates memory management so as to support batch problem processing.

6.2 Model Checker Integration

A key requirement of our evaluation is for both the conflict-rooted synthesis and model checking approaches to execute identical test cases. To this end NuSMV is fully integrated into the test configuration, allowing us to easily switch between approaches using the same input domain specifications. We standardised on PDDL input and focused on translating our PDDL domain specifications into an SMV model, and post processed the resulting NuSMV computations to extract the synthesised norms. We provide an overview of this integration.

6.2.1 Encoding of Focal State Reachability

In Section 2.5.2.2, expression (2.1) detailed how van der Hoek et al. (2007) ensure goal reachability by encoding focal state reachability into the social objective. The resulting reachability expression is repeated below:

$$S, s_0 \models \langle\!\langle Ag \rangle\!\rangle \mathsf{G}\Big(\varphi \wedge \bigwedge_{s_i \in \Sigma_F} \Big[s_i \rightarrow \bigwedge_{s' \in \Sigma_F} \langle\!\langle Ag \rangle\!\rangle \mathsf{F} s' \Big]\Big). \tag{6.1}$$

We argue that model checking this expression will not result in a computation where all focal states will be reached, and more importantly, is not sufficient to ensure that any subsequent paths from reached focal states will themselves comply with the social objective. We prove each of these in turn.

Lemma 6.2.1. *The computation derived from the positive witness does not always ensure that all focal states are reachable.*

Proof. Consider the example system detailed in Figure 2.1 where we write a, b, c to represent propositions indicating that we in state A, B or C respectively. Let the computation λ^* satisfy the expression (2.1). Let B be focal such that $\Sigma_F = \{b\}$ and let $\varphi = \neg c$: we wish to avoid C.

Let $\lambda^* = \{a\} \xrightarrow{\texttt{idle}} \{a\} \xrightarrow{\texttt{idle}} \{a\} \ldots$ be the infinite computation where the agent never leaves state A. λ^* satisfies (2.1) since φ holds in every state of λ^*. Furthermore, the reachability subexpression $b \rightarrow \langle\langle Ag\rangle\rangle \mathsf{F} b$ also holds, since we never enter state B, so the implication is always true. The focal state is never reached *directly* by adhering to the positive witness, and our lemma is proved. ■

Lemma 6.2.2. *Computations that satisfy the ATL expression (2.1) and show reachability between focal states can violate the social objective.*

Proof. Again we adopt a proof by contradiction. For this we use a simplified version of our previous example, again with $\varphi = \neg c$ but with $\Sigma_F = \{a, b\}$ and with no path existing from A to B, as depicted below:

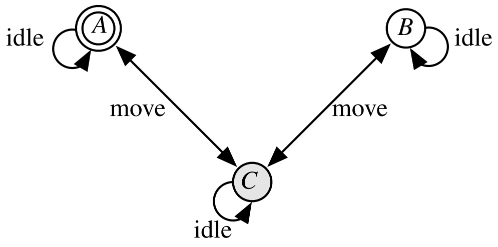
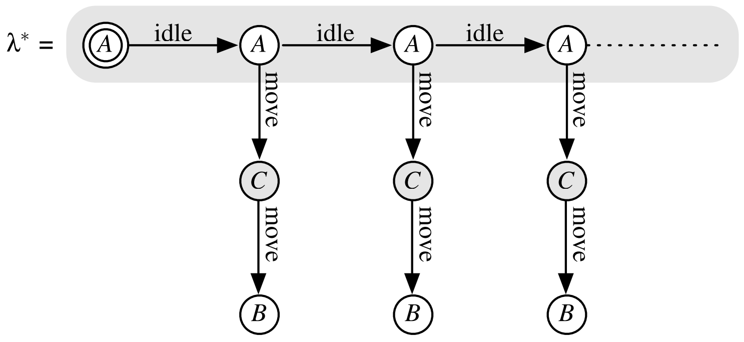


Figure 6.5: A refined three-state gridworld topology, with no direct path from A to B

Let $\lambda^* = \{a\} \xrightarrow{\texttt{idle}} \{a\} \xrightarrow{\texttt{idle}} \{a\} \ldots$ be the infinite computation where our agent never leaves state A. Such computation satisfies the expression detailed in (2.1), for in every state of the computation φ holds. However, the reachability subexpression $a \rightarrow \langle\langle Ag\rangle\rangle \mathsf{F} b$ *also* holds, since from every state in λ^* it is possible to achieve b:

$$\forall i \geq 0. \lambda^*[i] \models a \rightarrow \langle\langle Ag\rangle\rangle \mathsf{F} b.$$

Notice that this reachability requirement makes no mention of φ. Indeed, the alternative computations originating from state A must traverse through C to reach B, thereby conflicting with the social objective as depicted below:



Here λ^* ensures access to B, however only by traversing C. Therefore, the resulting computation from model checking (2.1) may result in norms that deny access to focal states. ■

From the above two lemmas we conclude that an additional mechanism, or more expressive representation, is required to ensure reachability of focal states when model checking is utilised to synthesise norms. We propose an alternative means of ensuring continued access to focal states through the specification of *fairness constraints*. Fairness constraints specify a set of states in the model that must be visited infinitely often by any resulting computation. Incorporating fairness constraints into CTL leads to a strictly more expressive logic often referred to as Fair CTL (Büchi, 1966). Fairness constraints reduce the set of computations investigated to only those that are guaranteed to traverse through the focal states specified in the constraints. The existing path operators apply to this reduced set only, and the remainder of the paths are discarded. Any resulting computation ensures focal state reachability since from any focal state one can follow the computation to reach any other focal state. We are now in a position where we have all the theoretical machinery required to synthesise norms using a model checker.

6.2.2 Choosing a Model Checker

We begin by justifying our choice of NuSMV as the model checker used in this evaluation. A candidate model checker must provide the following three capabilities in order to implement the norm synthesis approach:

1. **ATL or CTL**: model check either ATL or CTL expressions, or any more expressive temporal logic.[1]
2. **Witnesses**: the candidate model checker must return a counterexample when an expression does not hold in the model.
3. **Fairness Constraints**: support for the filtering of paths using fairness constraints to allow for the encoding of focal states.

van der Hoek et al. (2007) reference the MOCHA ATL model checker as a viable tool for implementing their synthesis approach. The MOCHA project provides two model checkers: `cMocha` (version 1.0.1) and `jMocha` (version 2.0). `cMocha` was proposed

[1]The more expressive CTL* is suitable yet other temporal logics, such as LTL, are not.

by Alur et al. (1998) as a tool with which to check *reactive models* (Alur and Henzinger, 1999). `cMocha` has one crucial flaw: it does not provide witness support so no counterexamples can be generated[2]. Furthermore, `cMocha` is no longer updated, with development of the MOCHA project now focusing on `jMocha` which in turn only supports invariant and refinement checking with no support for checking ATL expressions.

For our implementation we adopted the NuSMV symbolic model checker initially proposed by Cimatti et al. (1999). NuSMV is a redesign and reimplementation of the original CMU SMV model checker aiming to provide robust, state of the art model checker functionality that reaches industrial systems standards. For our purposes NuSMV offers a number of benefits:

- It supports CTL checking, fairness constraints and produces counterexamples.
- It is highly optimised with a strong focus on performance.
- It is current and frequently updated.
- It supports a range of models from asynchronous to synchronous.
- It uses optimised Binary Decision Diagram (BDD) and SAT-based model checking techniques.

What is particularly interesting about NuSMV is its integration with external libraries, utilising the CUDD BDD (Somenzi, 2005) and SIM SAT (Giunchiglia et al., 2001) libraries to offer improved checking performance. The only notable downside of using NuSMV is the lack of support for ATL, however no other applicable ATL model checker was found at the time of writing. Furthermore, since we are primarily interested in synthesising norms for all agents in the systems the grand coalition suffices, we substitute CTL for ATL without losing quality in our results.

NuSMV takes a SMV model as input that describes a set of synchronous or asynchronous finite state machines as reusable modular components that operate over a set of finite data types. For our purposes this is not a limitation since we assume our domains to be finite. Each machine includes a transition relation composed of propositional expressions allowing for a more succinct and compact representation. We will not present a complete analysis of the SMV language, but provide a sample model in Appendix B and detailed our integration in Section 6.2. Additional details can be found in the literature (Cimatti et al., 1999).

[2]The inability to produce a counter-example is documented in the model checker source code(Mocha, 2011). While initially planned, it was not implemented before work began on `jMocha`.

Our implementation of the conflict-rooted synthesis algorithm is called CRS and we refer to the model checking approach as NuSMV in order to differentiate between the theory and the resulting implementations. To invoke each approach on identical input domains we map PDDL input domains into SMV models, as detailed in Section 6.2. Our NuSMV integration operates as follows: Firstly, we automatically create SMV input from a set of PDDL files and a given conflict specification, enumerating variable parameters in the conflict specification. Next, the resulting set of ground expressions are checked against the generated model. Finally, the resulting computations found are automatically translated into a set of prohibitionary norms.

We do not automatically create fairness constraints to encode goal reachability. We justify this in Section 7.5.1 when describing how reachability conditions are encoded, but for now emphasise that this preprocessing is not included in our analysis of the model checking approach. We evaluate the best-case time to construct the model without performing any checking of the built model.

6.2.3 Incorporating NuSMV Domain Models

In order to utilise NuSMV to synthesise norms, a set of translation and extraction procedures are required to compose appropriate domains models to model check, and subsequently to produce prohibitionary norms from the model checker's output.

6.2.3.1 PDDL to SMV Translation

PDDL is translated into a finite state machine model that NuSMV model checks, and is composed of four key components:

1. **`VAR`**: The list of *state variables* used in the state model of the finite state machine. For example, in the Parcel Delivery domain we wish to model the location of agents and parcels:

```
VAR
   agentat_a1_n1 :boolean;
   parcelat_p1_n1 :boolean;
```

2. **`DEFINE`**: A list of *macro definitions* used to reduce the representation size of the finite state machine. Each macro symbol is associated with a Boolean expression over state variables. We use macro definitions for specifying non-fluents, as well as for encoding the preconditions of actions:

```
DEFINE
```

```
conn_n1_n2 := TRUE;
pickup_a1_p1_n3 := agentat_a1_n3 & parcelat_p1_n3;
```

Here we write & to be the logical conjunction of the left and right expressions.

3. **INIT**: The *initialisation constraints* specify the initial assignments of state variables, identifying the state that the checker will search from. In our example we specify the initial location of a_1 as follows, where ! is a logical negation:

```
INIT
   agentat_a1_n1 & !agentat_a1_n4 & !agentat_a1_n2 & !agentat_a1_n3;
```

4. **TRANS**: The *transition relation* is a set of current state / next state pairs specified as a Boolean expression. A given state pair are connected if the state variables satisfy the Boolean expression. We represent a transition invoked by Agent a_1 moving from $node_1$ to $node_2$ as:

```
TRANS
   move_a1_n1_n2 &                              -- Preconditions
   next(agentat_a1_n2) & !next(agentat_a1_n1) & -- Effects
   (agentat_a1_n4 <-> next(agentat_a1_n4)) &    -- Frame Conditions
   (agentat_a1_n3 <-> next(agentat_a1_n3))      --
```

Here <-> represents logical equivalence. For a transition to exist the action preconditions must hold, the next state must contain the effects of the action, and all unrelated state variables must remain constant.

Each of the four declarations above are encapsulated into a MODULE declaration called normative_system. In SMV modules can be instantiated into unique instances, with each module designed to represent an independent process. Here we are modelling asynchronous action so we utilise just a single module for the finite state machine. It should be clear how an arbitrary PDDL specification can be expanded from initial state specification to model the entire system. The set of state variables correspond to a conjunction of the predicates and planning domain objects specified in PDDL. We construct macro definitions for each of the non-fluents in the initial state specification, and we compose shorthand variables for the preconditions of each action. Finally, transitions follow directly through the application of each grounded action. We obtain the grounded actions by invoking the FF planner with a customised switch to print out all grounded actions for a given domain and subsequently parse this output to construct the SMV finite state machine. The result is that, given input PDDL we produce an SMV module that fully captures the domain.

6.2.3.2 Model Checking the Finite State Machine

In order to model check the `normative_system` module we require a second wrapper module, with the following components:

1. **`VAR`**: Here we define a single variable, a reference to the normative system module using the syntax:

   ```
   normative_system: process normative_system();
   ```

2. **`CTLSPEC`**: The CTL social object is specified next. Since SMV is purely propositional we must ground any variables in our conflict specifications accordingly. For the conflict specification $S_C = \{\mathtt{at(a_1,X)}, \mathtt{at(a_2,X}\}$, we ground all bindings of X to produce a set of constraints:

   $$\{\ \mathtt{at(a_1,node_1)}, \mathtt{at(a_2,node_1)}\ \}$$
   $$\{\ \mathtt{at(a_1,node_2)}, \mathtt{at(a_2,node_2)}\ \}$$
   $$\ldots$$

   We are interested in the computation where none of the above hold. As a result we produce a CTL expression of the form:

   ```
   CTLSPEC ! EG (
      !(normative_system.agentat_a1_n4 & normative_system.agentat_a2_n4) &
      !(normative_system.agentat_a1_n3 & normative_system.agentat_a2_n3) &
      !(normative_system.agentat_a1_n1 & normative_system.agentat_a2_n1) &
      !(normative_system.agentat_a1_n2 & normative_system.agentat_a2_n2))
   ```

   Recall that, since we are interested in the grand coalition we rewrite the original ATL expression $S, s_0 \models \langle\langle Ag \rangle\rangle \mathsf{G}\varphi$ to the equivalent CTL expression $S, s_0 \models \mathsf{EG}\varphi$. Additionally, since NuSMV produces counterexamples when a CTL expression does not hold we negate the original social objective.

3. **`FAIRNESS`**: We encode the reachability fairness constraints as a set of Boolean expressions over the state variables. For example, to ensure access to a state where Agent $\mathtt{a_1}$ is at location $\mathtt{node_1}$, and Agent $\mathtt{a_2}$ is at $\mathtt{node_2}$ we write:

   ```
   FAIRNESS
      normative_system.agentat_a1_n1 & normative_system.agentat_a2_n2
   ```

   We produce a single fairness constraint per focal state. Since we wish to guarantee access to all focal states we list each in its complete form.

While the translation from focal states to fairness constraints is straightforward, the question of how to produce the list of focal states is more complex. There are two issues associated with creating the list of focal states:

- Enumerating conflict-free states is not a viable solution since there is no guarantee that a synthesised conflict-free state is reachable in the original system. Since we wish to preserve reachability, we must require states to be reachable in the normative system only if they are reachable in the original.
- This approach does not support terminal focal states that are reachable but from which it is not possible to return to all other focal states.

In order to avoid the complete state enumeration and reachability check required to find all focal states we assume this knowledge to be supplied. While it is possible for us to automate the creation of these constraints in the Parcel Delivery domain the same does not hold for the IPC domains. For tests requiring timing of the model checker we record the time required to construct the model, prior to checking. As such we compare against the best case performance of the model checker in situations where enumerating the focal states is not feasible.

6.2.3.3 Prohibitionary Norm Extraction

NuSMV produces a computation where transitions between states are described by the change in the state variables. For example, where Agent $\mathtt{a}_2$ moves from $\mathtt{node}_2$ to $\mathtt{node}_4$ the output is of the form:

```
-> State: 1.2 <-
  normsystem.agentat_agent2_node4 = 1
  normsystem.agentat_agent2_node2 = 0
  ...
```

This output states that the proposition indicating $\mathtt{a}_2$ is in $\mathtt{node}_2$ is set to false (0), and the new location of $\mathtt{node}_4$ set to true (1). From this we construct a computation originating from the initial state specification that traverses through all focal states. The resulting prohibitionary norms are extracted from the computation, where for each state a prohibitionary norm is created prohibiting the agents from performing any action other than that specified in the run. The set of resulting prohibitionary norms is:

$$\left\{ \ \left\langle R[i], \neg R[\xrightarrow{\mathtt{i}}] \right\rangle \ \mid \ \forall i < |R| \right\}$$

where $R[i]$ indicates the i'th state specification in the run R, and $R[\xrightarrow{i}]$ the i'th operator. We negate the operator in the norms so as to prohibit the application of any other operator. Equivalently, we could synthesise *obligatory* norms to perform the action in each of of the states, rather than prohibit all other actions.

6.3 Empirical Evaluation Design

We now consider both NuSMV and CRS as black box implementations and describe the surrounding evaluation configuration used during our evaluation. In Figure 6.6 we detail the key components we utilised to perform a balance comparison between the two approaches.

To ensure accurate comparisons both approaches operate off identical domain representations. From the set of five domains the domain selector is configured to select the desired domain. The selected domain is the utilised as input into three subsequent components. First, the problem selector takes the domain as input and selects a specification from the set of problems. Next, both the problem and domain specifications are combined to form the PDDL input specification. Finally, the conflict generator randomly generates a conflict specification that will be used for the evaluation run.

The PDDL input and conflict specifications are passed to the synthesis approaches. CRS synthesises norms directly from these specifications, but a translation is required in order to produce a model that NuSMV is capable of checking. The PDDL to SMV translator (outlined in Section 6.2) produces a SMV model that fully describes the PDDL domains and conflict specification, and this model is subsequently passed to NuSMV for the checking to commence.

6.3.1 Conflict State Generation

In order to effectively evaluate our approach we require a means of generating conflict state specifications. Our approach randomly assigns literals to each specification, choosing with probability 0.5 whether to bind with existing literals in the specification or to introduce new variable symbols, allowing us to create specifications with dependencies between literals. We present our procedure for random conflict state specification in Algorithm 11.

RandomPredicate(P) randomly selects a predicate definition from the set P after which each of the parameters for the predicate are set. Parameters are selected randomly with

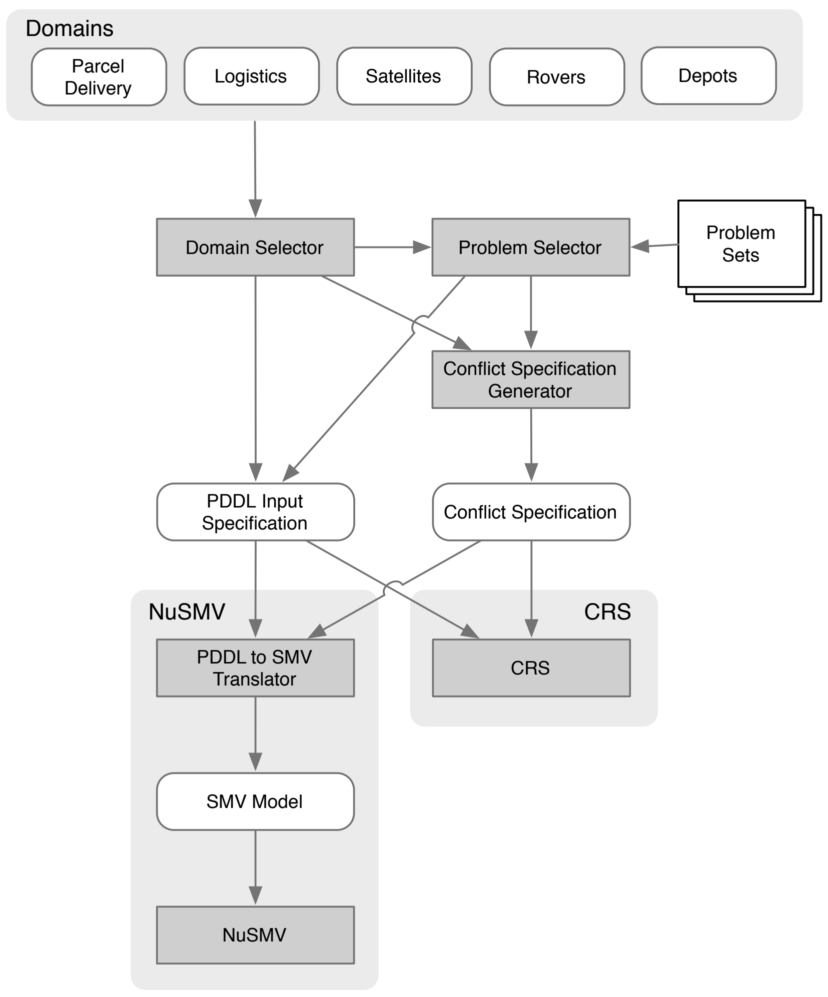


Figure 6.6: The architecture and key components of the CRS and NuSMV evaluation configuration.

probabilities sampled from a uniform distribution. Options are weighted equally in order to generate sufficiently varied conflict specifications that represent the possible specification space. *IntroduceNewParameter* returns true with probability 0.5, and is used to decide whether a new parameter should be introduced, or this predicate should share a parameter with an existing predicate in the specification. *NewParameter* will either create a new variable parameter with probability 0.5 or will randomly select a constant from the domain. In the event that an existing parameter is selected the method $SelectExistingParameter(S)$ randomly selects a parameter from the set of existing parameters of predicates in S.

Example As an example, consider the steps followed to generate a specification of the following form in the Parcel Delivery domain:

$$\texttt{agentAt}(\texttt{Agent},\texttt{node}_1),\texttt{hold}(\texttt{Agent},\texttt{Parcel})$$

Algorithm 11: Synthesising Prohibitionary Norms

```
Input: The size of the desired specification $n$, the set of predicates $P$
Result: A randomly generated specification $S$
1  begin
2      $S \leftarrow \{\}$
3      for $1 \ldots n$ do
4          $p = RandomPredicate(P)$
5          for each parameter $p_i$ of $p$ do
6              if IntroduceNewParameter(0.5) then
7                  $p_i \leftarrow NewParameter(0.5)$
8              else
9                  $p_i \leftarrow SelectExistingParameter(S)$
10         $S \leftarrow S \cup \{p\}$
11     return $S$
12 end
```

The construction of this specification would occur as follow:

- Randomly select a specification length of 2.
- The literals `agentAt` and `hold` are assigned randomly from the set of possible predicates in the domain.
- The first parameter is chosen to be a variable and `Agent` is introduced. The second parameter is chosen to be unique and ground, and $\texttt{node}_1$ is randomly selected from the set of available objects.
- Next we choose to let the first parameter of `hold` be dependent on an existing symbol, and so `Agent` is selected.
- Finally, the variable `Parcel` is introduced for the final parameter. □

This process generates conflict specifications that may or may not ensure goal reachability in the normative system and since we wish to test the empirical behaviour of both the positive and negative results of our algorithm we do not filter out any cases, but instead ensure that half of the randomly chosen specifications are shown to be reachable.

We utilise CRS to determine whether runs in the sample set are reachable, allowing us to construct an evaluation set that balances problems in which CRS succeeds, with those that CRS fails. Importantly, there is no danger of CRS gaining unfair advantage through this process. On the contrary, since CRS terminates once reachability analysis fails an increase in the number of problems in which CRS succeeds ensures that these more expensive executions are equally represented. This process allows us to investigate CRS performance across a range of input in a balanced manner.

6.4 Summary

CRS is a Java-based implementation of conflict-rooted synthesis which invokes FF to solve reachability planning problems. The key implementation details are:

- CRS is composed of a PDDL parser and associated model, traversal and reachability analysis procedures as well as implementations for each of the optimisations detailed in this work.
- To improve the performance of planning during reachability analysis CRS utilises a modified version of the FF planner called BatchFF, allowing for the batch solving of planning problems.
- It utilises a succinct data structure, called a *traversal graph* to represent sets of similar runs, saving both on computational and space requirements.

In order to ensure an accurate comparison NuSMV is integrated directly into our test mechanism. PDDL input is translated into a SMV finite state machine, and the conflict state specification is mapped to a CTL expression to be checked. We illustrate the syntax of the generated SMV as well as the form of the output produced by NuSMV, and briefly detail how prohibitionary norms can be created from the resulting computation.

Both the NuSMV integration and the CRS implementation were thoroughly tested. By constructing a normative system using the synthesised norms we repeatedly invoke FF to find a plan from the initial state to a conflict state in the normative system. By ensuring that no plans were found we could be certain that it was not possible for the normative system to transition into a conflict state, thereby indicating that both synthesis approaches were functioning correctly.

Chapter 7

Evaluation

We hypothesised that a synthesis procedure based on localised search around conflict states is more *efficient* at synthesising norms, that the resulting process is *anytime* and that the produced norms are of a higher *quality*. Our evaluation validates these claims through analytic analysis and empirical tests to achieve three objectives:

1. **Optimisation Effectiveness**: We illustrate that the core, unoptimised, synthesis procedure is not applicable in practical domains due to the computational complexity involved in performing combinatorial search. With this motivation we investigate how effective each optimisation is at reducing these computational requirements in a variety of benchmark domains.
2. **Theoretical Comparison**: We theoretically compare the conflict-rooted synthesis and model checking approaches comparing the methods and results produced, while analysing the impact these differences have on the resulting synthesised norms.
3. **Real-World Applicability**: We provide a structured, thorough empirical comparison between the conflict-rooted synthesis and model checking approaches in order to assess the benefits conflict-rooted synthesis provides for system designers and norm autonomous agents in practice.

We first present the evaluation design, detailing the choice of domains, experimental parameters and metrics captured. We then propose an empirical plan for each of the three objectives, and follow each plan with results and discussion. The combination of plan, results and discussion act as the basis from which we gauge whether or not the hypothesis in this work holds.

7.1 Empirical Evaluation Parameters and Metrics

We begin by detailing the components of our evaluation plan, including the evaluation domains used for empirical comparison, the control parameters and the resulting metrics that are collected and analysed. We do not present a plan for our evaluation here but simply detail the tools required.

7.1.1 Evaluation Domains

The automated planning community have a well defined set of evaluation domains that provide a basis for the empirical comparison of planning systems. The multi-agent systems community does not boast a single, generally accepted set of evaluation domains mainly due the broad nature of research in the field, and due to the lack of standardisation on agent system specifications. Typically, a customised evaluation domain is developed alongside each new body of work, making the direct comparison of agent technologies difficult.

Our alignment with planning-based theory provides a well structured and generally accepted domain representation language allowing us to easily harness the standardised planning domains present in the planning literature. There are three benefits to adopting planning domains:

- They are *generally accepted* testbeds for empirical evaluation and the community is familiar with the characteristics and nuances of these domains.
- A large set of tailored *predefined problem instances* exist, reducing the need for hand crafting or randomly generating problem instances.
- The benchmarks are designed to be *challenging*, realistically modelling characteristics of real-world domains.

Our domain set includes the Parcel Delivery domain and a subset of domains featured in the International Planning Competition.

7.1.1.1 Parcel Delivery Domain

We have previously introduced the Parcel Delivery domain in Section 1.4. The operator schemata used for our evaluation have been included in Appendix A, and model agents that utilise the `move`, `drop` and `pickup` operators detailed previously. This simple world allows us to easily generate problem instances that challenge modern classical planners simply by increasing the size of the grid, and the number of parcels in it. For the purposes of this evaluation we limit ourselves to grid-like configurations of the world, and randomly distribute parcels and agents within it.

7.1.1.2 International Planning Competition Domains

The International Planning Competition (IPC, 2011) is a key event in the field of automated planning, and is run during the International Conference on Automated Planning and Scheduling, with the goal to provide a standardised testbed of challenging problem domains that can be used to analyse the state of the art in automated planning systems. The competition provides a set of domains classified into different tracks that are used to evaluate different approaches to automated planning. The deterministic, observable track includes classical planning problems, including those adopted in this work. Other tracks include planning with uncertainty (for non-deterministic and probabilistic actions in fully or partially observable domains) and planning with learning, but we do not incorporate these into our domain set.

We adopt four sample domains from the competition's deterministic track based on how easily they can be represented in a multi-agent setting. When given the option we chose typed variants of each domain and all domains adopt a classical representation with parameterised operators. We summarise the essential details of each domain next, and present the operator schemata in Appendix A.

7.1.1.2.1 Logistics The Logistics domain is a common benchmark domain often used in AI planning. Each problem is composed of a set of *cities* and within each city are a set of *locations*. Each agent has a goal to move packages from a source location in some city to a target location (possibly in a different city). Agents can utilise different vehicles to traverse the world. *Trucks* are vehicles that are able to move between locations in the same city, whereas *aeroplanes* are able to traverse between cities but are limited to locations of type *airport*.

There are six operators defined in the domain. Packages can be *loaded* into trucks if the package and truck are in the same location, or *unloaded* from trucks. Similarly, packages can be loaded into aeroplanes. Aeroplanes can *fly* between airports in different cities, and trucks can *drive* between locations in the same city.

7.1.1.2.2 Depots Depots is similar to the Logistics domain presented above. Here we are concerned only with *trucks* driving between *depots* and *distributors* delivering packages. The additional complexity is introduced when trucks can no longer simply load or unload packages as before, but must now ensure that packages are stacked onto pallets at their destinations. The stacking is achieved using hoists: the resulting stacking problem is therefore very similar to the classic block stacking problem.

There are five operations defined in the Depots domain. Trucks can *drive* between locations. At each location a hoist is able to *lift* crates up off of a pallet and *drop* crates down onto a pallet. Similarly, hoists are able to *load* and *unload* crates from trucks.

7.1.1.2.3 Rovers The Rovers domain is inspired by planetary rover problems. This domain requires that a collection of rovers navigate a planet surface looking for rocks and soil to sample. Once sampled the results are sent back to the centralised lander. Different rovers have different capabilities: some are only able to analyse soil and others can analyse rocks. Some are equipped with a number of cameras, each of which can be used to photograph a particular object.

The domain has 9 operators. Agents can *navigate* between waypoints if it is possible to traverse between them. They can *sample soil* and *sample rocks* at a particular waypoint, which involves the rovers filling their internal stores with the sample. Once processed they are able to *drop* the contents of their store in order to empty it. If a rover is equipped with a camera they can *calibrate* the camera for a particular object, and can subsequently *take an image* of the object. Finally, they can communicate the *soil*, *rock* and *image* data back to the mothership.

7.1.1.2.4 Satellites The Satellites domain is our second domain inspired by space applications. It involves planning and scheduling a collection of observation tasks between multiple satellites, each of which is equipped in slightly different ways. The goal of this domain is to collect image data. Each satellite is able to orientate itself in a particular direction in order to take an image.

This domain has five operators. Satellites can *turn* so that they point in a new direction. In order to take a photo the instrument must be *calibrated* for each new direction it faces. A calibrated satellite can toggle an instrument on the satellite *on* or *off*. In this version of the domain the satellite can only have one instrument on at a time. Finally, the satellite can take an image of the direction it is facing, with its calibrated instrument. The goal of the domain is to collect a required set of such images.

7.1.1.3 Testing and Evaluation Domain Sets

For testing purposes, a set comprised of the Parcel Delivery domain and the Logistics domains were used. The final evaluation was performed using the set of all domains. We chose to utilise both the Parcel Delivery and Logistics domains in the evaluation set since the variability introduced through the generated conflict specification and problem instances is sufficient to greatly distinguish the input specifications in the evaluation set from those in the test set. While the sets of operators are identical, all other input parameters are variable.

7.1.2 Evaluation Parameters

Evaluation parameters are configurable attributes of the conflict-rooted synthesis process that are adjusted for each evaluation iteration. We detail each of the evaluation parameters in turn.

7.1.2.1 Conflict Specifications

The performance of our algorithm and the effectiveness of each of the optimisations is dependent on the particular conflict specification used. Conflict specifications dictate which successor operators will be considered during traversal, and in turn affect the size of the resulting traversal search space. In order to accurately estimate the behaviour and scalability of our approach we use a set of randomly generated conflict specifications, aggregating collected metrics over these varying specifications. The process of randomly generating conflict state specifications is presented in Section 6.3.1. We investigate specifications containing a minimum of 2 literals and a maximum of 5 providing a sufficient range from simple to complex conflict specifications.

7.1.2.2 Optimisation Activation

Any combination of conflict-rooted synthesis optimisations can be enabled. In Section 5.1.8 we argue for an ordering of optimisations based on the complexity of performing the optimisations, and we adopt this sequence permanently.

7.1.2.3 Problem Instance Specifications

We select 20 problem instances for each IPC domain, a figure that is dependent on the number of problems available in the competition, as follows:

- **Logistics**: 20 problem instances from the 2000 competition.
- **Rovers**: 20 problem instances from the 2002 competition.
- **Satellites**: 20 problem instances from the STRIPS track of the 2004 competition.
- **Depots**: 20 problem instances from the STRIPS track of the 2002 competition.

One benefit of the Parcel Delivery domain over the IPC domains is that it affords us a great degree of control over the problem instances generated. This makes it the ideal domain with which to begin our investigations. CRS is a domain-independent algorithm so it is possible that many other competition domains could be selected. The above domains are suitable for our purposes as they are the set that can most easily be interpreted as multiagent systems, and while later competitions have proposed other domains they have largely been extended to include advanced planning concepts that our formalism does not support.

7.1.3 Measured Metrics

We now discuss the set of metrics measured during the execution of our algorithm.

7.1.3.1 Computational Time

Measuring computational time as a basis for comparison between competing methods typically requires commonality in implementations, yet here CRS is Java-based and NuSMV is natively implemented in C. We are selective about what conclusions we draw from a head-to-head analysis since the codebases are significantly different. We still adopt the measurement of computational time as a fundamental comparison metric for a number of reasons:

- the relative change in computational time with respect to the varying of parameters and problem instances allows us to compare the rate of change without comparing the times themselves,
- computational time gives us an idea of real-world execution times, and
- we empirically quantify the relative effect of the optimisations on the execution time of the conflict-rooted synthesis process.

For CRS we measure the computational time required to perform traversal and reachability separately. For NuSMV we measure the time required to load (and in the case of the Parcel Delivery domain, search) the model, but do not include the time required to translate the input representations. Computational time is measured in milliseconds unless otherwise stated. Importantly, we choose to measure total computational time as opposed to user time since CRS performs significant input and output to disk and invokes a separate planner process. Overall computational time is a more effective measure of real-world performance in this case.

7.1.3.2 Conflict Runs and Reachability Checks

We quantify how effective an optimisation is by measuring the relative decrease in conflict runs or reachability checks. We measure both the number of complete and incomplete runs at the end of each iteration of the traversal process, as well as the total number of reachability checks performed.

7.1.3.3 Optimisation Success Rate

We use two measures of how effective an optimisation is. In the case of traversal optimisations we study the change in the number of runs produced with and without each of the optimisation combinations. One downside of this measure is that it is not independent of the conflict specification or problem domain since some inputs simply result in fewer conflict runs than others. A second metric improves this by analysing the relative reduction in number of runs, allowing us to normalise the result thus making it more suitable for comparison using different domains and conflict specifications. This metric has an additional downside: results are skewed and unreliable when bounds to the traversal process are applied. A less efficient approach may generate *fewer* resulting runs for a particular run length since the search is truncated, while a more efficient search may search beyond this limit without violating the bound.

Our chosen metric to effectively gauge the performance of an optimisation is its *success rate*. The success rate of an optimisation *Op* is written as:

$$Success\ Rate(Op) = \frac{Number\ of\ runs\ discarded\ by\ Op}{Total\ runs\ analysed\ by\ Op} \times 100.$$

When a run is processed by an optimisation we monitor whether or not the run is discarded. The success rate identifies the percentage of runs removed by an optimisation and is independent of the number of complete runs generated allowing us to compare results more effectively.

7.1.3.4 Quantity of Norms Generated

Our approach preserves generality during traversal to produce norms that apply to larger sets of states, thereby producing fewer norms for a given social objective. In order to quantify the generality of produced norms we measure the number of norms required to implement the social objective. The fewer norms required, the more generally applicable the synthesised norms are. It should be noted that additional runtime processing of unground norms is required. We discuss the benefits and limitations of norm expressiveness in Section 8.1.4.1.

7.1.4 Execution Environment

All empirical tests were run on identical hardware, under consistent, reproducible conditions. The required number of iterations for each test are minimised due to two reasons:

1. Conflict-rooted synthesis is an *exhaustive search*, so the order taken during search does not affect the norms synthesised.
2. The entire algorithm is *deterministic*. The outcomes of performing actions are certain and the traversal process produces identical conflict runs.

The exhaustive, deterministic nature of our algorithm implies that fewer repeated evaluation runs need to be executed for identical input. We account for the slight variance in certain metrics (most notably computational time) by executing multiple runs and aggregating the results accordingly. We compute both the mean and standard deviation for each metric. Furthermore, we track both the minimum and maximum values for all measured metrics. Finally, even though computation of the conflict-rooted synthesis process can be distributed we restrict the implementation to a single execution thread in order to better compare results with the model checker. All tests were run on a machine composed of a Intel Core 2 Duo 2.66 GHz processor and 4 GB of RAM running Linux Kernel 2.6.31-20.

7.2 Empirical Evaluation

The evaluation is split into three parts. First, we analyse the performance of the core conflict-rooted synthesis algorithm with none of the optimisations enabled. Next, we investigate what improvements the optimisations bring, and thirdly we conduct a comparison between CRS and NuSMV. We present the plan for each part separately. We introduced three mechanisms to bound the traversal process in Section 4.3.7.1: limiting the number of runs, limiting the maximum length of runs and restricting the number of unique predicates in specifications according to a given problem instance. We adopt all three of these approaches in our evaluation.

7.3 Part A - Core Synthesis Performance

Much of our work has focused on optimisations to improve the synthesis process. In this first part of our evaluation we justify this by investigating how efficient the synthesis process is without any optimisations enabled, and analyse how the algorithm scales when a full, unoptimised search is performed. While it is possible to theoretically deduce that the algorithm will conduct a full search of the space in the worst case, it is difficult to quantify how poor this search may be in practice. We show how the algorithm scales computationally as the domain size is increased and highlight how, even for very small domains, an excessive amount of computation is required.

7.3.1 Part A - Evaluation Setup

We execute our tests using instances of the Parcel Delivery domain, with increasing grid sizes. We iterate the process 10 times, collecting the computational time and run count metrics. Since we adopt no optimisations here we limit the length of the runs to 6 as longer runs were not manageable on our test machine. We summarise these settings in Table 7.1 below. We generate square grids of varying dimensions in the Parcel Delivery domain, with each grid node connected to its four neighbours. Note that we constrain the underlying graph representation to a grid, but emphasise that the domain operators allow agents to traverse a more general graph topology. For the purposes of this evaluation we use the following conflict specification:

$$S_C = \{\ \texttt{agentAt}(\texttt{a}_1,\texttt{X}),\ \texttt{agentAt}(\texttt{a}_2,\texttt{X})\ \}$$

where we randomly position two agents at arbitrary, unique locations on the grid. The conflict specification identifies states where each of the two agents ($\texttt{a}_1$ and $\texttt{a}_2$) occupy the same location, specified by variable $\texttt{X}$.

Domain	Parcel Delivery World
Conflict State Specification	100 Randomly Generated
Problem Instance Set	Grid sizes from 2x2 . . . 5x5
Optimisations	None
Measured Metrics	Computational Time (ms) Number of Complete Runs
Bound	Run Length ≤ 6

Table 7.1: Part A - Empirical Evaluation Setup for Core Synthesis Performance

7.3.2 Part A - Results

Table 7.2 presents the change in metrics as the grid size is increased. Note that only the traversal and reachability components are timed, since the remaining computation is negligible.

Grid Size	Computation Time (ms)	Traversal Time (ms)	Std. Dev.	Reachability Time (ms)	Std. Dev.
2 x 2	7164.43	147.46	22.16	6984 .36	243.73
3 x 3	54172.10	142.86	24.09	53994.73	513.12
4 x 4	284751.55	154.41	21.93	284564.22	651.75
5 x 5	1062935.13	165.93	20.34	1062672.91	1594.66

Table 7.2: Conflict-rooted synthesis computational time with no optimisations in the Parcel Delivery domain.

Since conflict traversal is independent of the initial state of the system it is not dependent on the size of the grid. Adjusting the grid size has no significant impact on the time taken to construct the traversal runs. The growth in time is entirely due to the grounding of the complete runs and subsequent reachability checks performed. This increase is also expected: the larger the grid, the more possible groundings exist for every unground run. Table 7.3 illustrates this growth by presenting the number of complete ungrounded runs generated during traversal, as well as the total number of grounded runs created prior to reachability analysis.

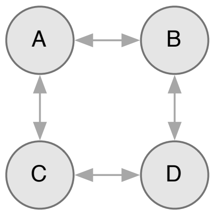
It is interesting to consider the rate of increase of the number of grounded runs. The increase factor is the number of grounded runs at a particular grid size, divided by the number of runs at the previous size, and gives an indication of how the search is scaling. Note that the increase factor decreases as the grid size gets larger. This is a domain specific feature: as the grid sizes increase there is a reduced increase in the

Grid Size	Ungrounded Run Count	Grounded Run Count	Increase Factor
2 x 2	180	3017	-
3 x 3	180	18688	6.19
4 x 4	180	57024	3.05
5 x 5	180	134384	2.35

Table 7.3: The number of ungrounded runs created during traversal and the number of grounded runs generated during reachability analysis in the Parcel Delivery domain.

number of grid nodes, resulting in fewer groundings being found.

Example Consider the Parcel Delivery domain where agents are only permitted to `move`. A conflict specification devised to avoid collisions results in runs of length 3, where agents `move` into conflict, and subsequently `move` out. In a 2x2 grid there are 16 unique conflict runs of length 2, as illustrated below.



1. A → B → A	9. C → A → C
2. A → B → D	10. C → A → B
3. A → C → A	11. C → D → C
4. A → C → D	12. C → D → B
5. B → A → B	13. D → C → D
6. B → A → C	14. D → C → A
7. B → D → B	15. D → B → D
8. B → D → C	16. D → B → A

In a 3x3 grid this increases significantly to 68. The factor of increase then decreases in a 4x4 grid, with 152 runs found. □

The decrease in the increase factor of grounded runs results is reflected in the computational time of the algorithm. We illustrate this decrease in computation time as the grid sizes increase, in Figure 7.1.

The core conflict-rooted synthesis method is a brute force state space search, akin to a complete breadth first search of a graph. Searching the exponentially growing space of possible runs is very complex: computation takes over 17 minutes to complete in a 5x5 Parcel Delivery world.

Let us now consider the expected variance of the results collected. Given a domain, the norm synthesis process is entirely deterministic, and efforts have been made to ensure that the results generated are as reproducible as possible. Repeated iterations of

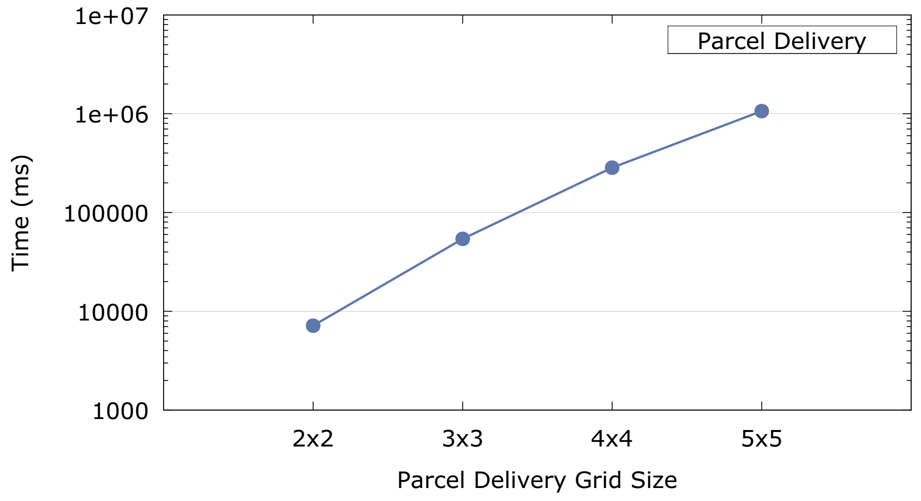


Figure 7.1: Logarithmic computation time plotted for increasing grid sizes in the Parcel Delivery domain.

the algorithm create the same number of runs, perform the same reachability checks, and produce the same set of synthesised norms. However, the algorithmic computation is directly linked to the input domains considered, and in this way is similar to classical planning: factors in the input domain dictate the size of the search space and the resulting performance of the algorithm. Therefore aggregating performance metrics over varying domains is expected to produce varied results and a corresponding high variance in the performance of the algorithm. The same holds for varying instances of domains (for example, adjusting the topology of the Parcel Delivery domain grid) and even for the conflict state specifications provided as input. In summary, when the input to the conflict synthesis algorithm is fixed, the process is deterministic and computational metrics are expected to converge. When the input varies we expect a significant variance in the measured metrics.

7.4 Part B - Optimisations

We now highlight the computational benefits of the optimisations proposed in Chapter 5 which take advantage of implicit dependencies between operators to reduce computation performed during synthesis. The dependences are specific to the operator schema in the problem domain and it is therefore unrealistic to analytically quantify the advantages provided by the optimisations without considering practical domains. The aims of this evaluation part are:

1. Investigate what effects each traversal optimisation has on the number of conflict runs generated during traversal.

2. Investigate what effects the optimisations have on the computational time and resources required for synthesis.
3. Investigate the effects the reachability optimisations have on the resulting groundings and reachability checks.

7.4.1 Part B - Evaluation Setup

For this analysis we use both the IPC and Parcel Delivery domains. We monitor five metrics for comparison: synthesis computational time, optimisation computational time, number of complete runs generated, number of reachability checks performed and the success rate of optimisations. For each domain we construct an evaluation set for each unique tuple of input with varying conflict specifications and problem instances, and vary which combinations of optimisations are enabled. Since our approach is deterministic we limit each test to 10 iterations. We set the active optimisations according to two policies:

1. **In Turn**: Perform a linear pass through all optimisations enabling each in turn and disabling all others to analyse the effects of each optimisation independently.
2. **Sequential**: Begin with all optimisations disabled and perform a sequential pass enabling each optimisation in the sequence, with ordering based on the computational complexity of each optimisation presented previously.

Table 7.4 presents a summary of the evaluation configuration for this part.

Domains	Parcel Delivery World	IPC Domains
Conflict State Specification	100 Randomly Generated	100 Randomly Generated
Problem Instance Set	Grid sizes from 2x2 . . . 5x5	IPC Domain Sets
Optimisations	In Turn, Sequential	
Measured Metrics	Synthesis Computational time Optimisation computational time Number of conflict runs Optimisation success rate Number of reachability checks	
Bound	Run Count $\leq$ 20000 In Turn - Run Length $\leq$ 6 Sequential - Run Length $\leq$ 15	

Table 7.4: Part B - Empirical Evaluation Setup for Optimisation Performance

7.4.2 In Turn Traversal Optimisation Results

We begin with the results for the Parcel Delivery domain, and follow this with the results for all the remaining domains. The In Turn empirical results in the Parcel Delivery domain are presented in Figure 7.2, and are plotted on a logarithmic scale. The results presented are averaged over 100 randomly generated conflict state specifications. Run lengths are bound to 6.

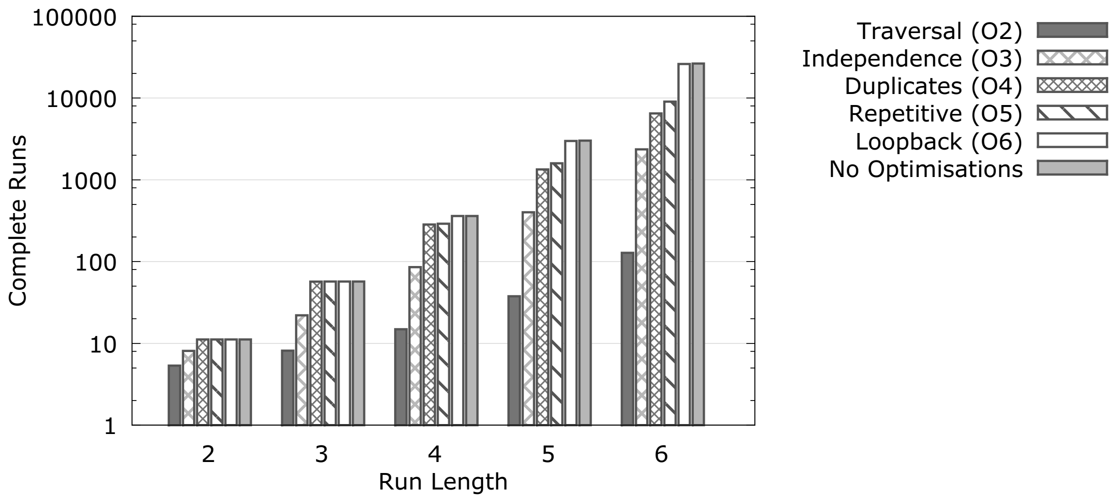


Figure 7.2: The number of complete runs produced during traversal in the Parcel Delivery domain with each optimisation enabled in turn.

The core, unoptimised performance of our approach is represented by the No Optimisations bar. Each preceding bar highlights the number of complete runs generated when each listed optimisation is enabled where smaller bars represent better results. The Loopback optimisation is the least effective optimisation as the number of runs is virtually identical to unoptimised performance. This may seem counter intuitive since the set of synthesised runs in the Parcel Delivery domain is sure to contain instances where agents undo a previous action by moving out of their initial node, and then subsequently moving back. For example consider the run where Agent $\mathtt{a_1}$ is initially at $\mathtt{node_1}$ and subsequently performs the following:

$$\bigcirc \xrightarrow{\mathtt{move(a_1,node_1,node_2)}} \bigcirc \xrightarrow{\mathtt{move(a_1,node_2,node_1)}} \bigcirc$$

The result of performing this run is that Agent $\mathtt{a_1}$ returns to their start location ($\mathtt{node_1}$), yet it does not register as a loop. The resulting initial and final state specifications differ: the final state specification contains the literal $\neg\mathtt{agentAt(a_1,node_2)}$ which is not present in the initial specification. As such, we expect the Loopback optimisation to improve as the run length increases. Similarly, the longer the runs the higher the likelihood that the Repetitive operator optimisation will be invoked.

The Traversal and Independence optimisations produce the best results: the effectiveness of these optimisations is attributed to the naive way in which successor actions are selected during traversal. We consider all possible groundings of each action, many of which are irrelevant. These optimisations remove these irrelevant actions from consideration, resulting in far fewer successor operators. In particular, both discard the shortest runs of length 2, thereby pruning the resulting search space significantly. The performance of the Duplicates optimisation lies between Loopback and the Traversal and Independence optimisations, with its effectiveness relative to the number of runs that have been considered. As the number of runs increases the performance of the duplicate run optimisation increases similarly.

Optimisation	Average Complete Runs	Minimum	Maximum	Standard Deviation
Traversal (O2)	854.31	318	2829	565.40
Independence (O3)	1475.16	19	5689	1430.41
Duplicates (O4)	4268.25	602	13424	3029.20
Loopback (O6)	5683.00	2566	13611	2039.55

Table 7.5: Metrics highlighting the dependence of the number of complete runs generated during conflict traversal on the input conflict state specification in the Parcel Delivery domain. Results are for runs of length 6 and are averaged over 100 random conflict specifications.

Since these results are averaged over randomly generated conflict state specifications we expect high deviations from the means. We highlight this in Table 7.5. The large variance does not allow us to be as objective as possible regarding the performance of each of the optimisations, since the number of complete runs generated is dependent on the input domain and conflict specifications. There is no benefit to constraining the forms of our conflict specifications either as single literal conflict specifications can result in large variance in performance.

One means of reducing the dependence on input parameters is to adopt the success rate of each optimisation as a metric that can be aggregated over different runs. In Figure 7.3 we illustrate the success rate of each optimisation, again averaged over 100 randomly generated conflict specifications, split into two graphs to avoid the overlap of error bars.

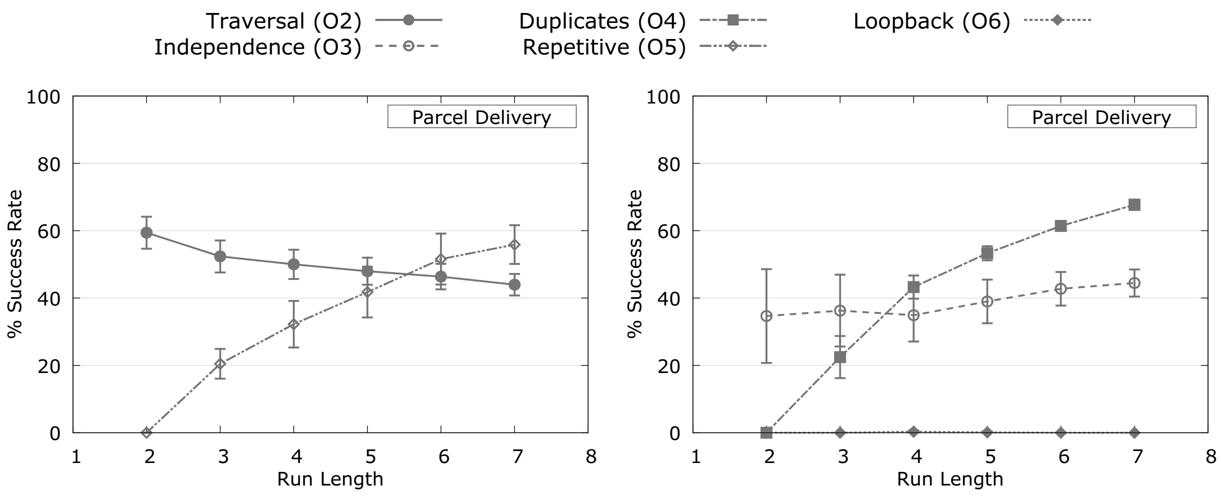


Figure 7.3: Optimisation success rate with error bars indicating the standard deviation from the mean, aggregated over 100 randomly generated conflict specifications.

The success rate results confirm our initial impressions while providing a more accurate deviation bound. Loopback has a very poor success rate that is not distinguishable from the baseline of 0. Traversal and Independence are the only optimisations able to remove runs of length 2, where all removals by the Traversal optimisation occur due to its reverse application as there are no intermediate operators to reorder: the first of the two operators is reordered after the second. As is expected the performance of the Duplicate and Repetitive optimisations increase as the length of the runs increases.

Perhaps the most interesting result from Figure 7.3 is the *reduction* in the success rate of the Traversal optimisation. We don't expect the optimisation to improve with run length since it only considers the final operator for reordering, with all previous operators already considered. Furthermore, as the run length increases the likelihood of the reordered operator conflicting with intermediate operators increases. The Traversal optimisation is therefore more likely to succeed with shorter runs.

Optimisation	Success Rate %	Minimum %	Maximum %	Standard Deviation
Traversal (O2)	46.36	37.00	53.80	3.78
Independence (O3)	42.74	28.80	52.50	4.99
Duplicates (O4)	61.40	59.74	63.81	0.89
Repetitive (O5)	51.57	39.41	70.50	7.56
Loopback (O6)	0.12	0	1.01	0.06

Table 7.6: The success rate of each traversal optimisation for runs of length 6 in the Parcel Delivery domain.

The success rate metric limits the variance significantly, allowing for a more objective statement regarding the performance of each optimisation. Table 7.6 highlights the aggregations for runs of length 6 averaged over 100 randomly generated conflict state specifications. We present the results of similar tests for the four IPC evaluation domains. For brevity we group our results together in Figure 7.4 and follow this with a unified discussion of the results.

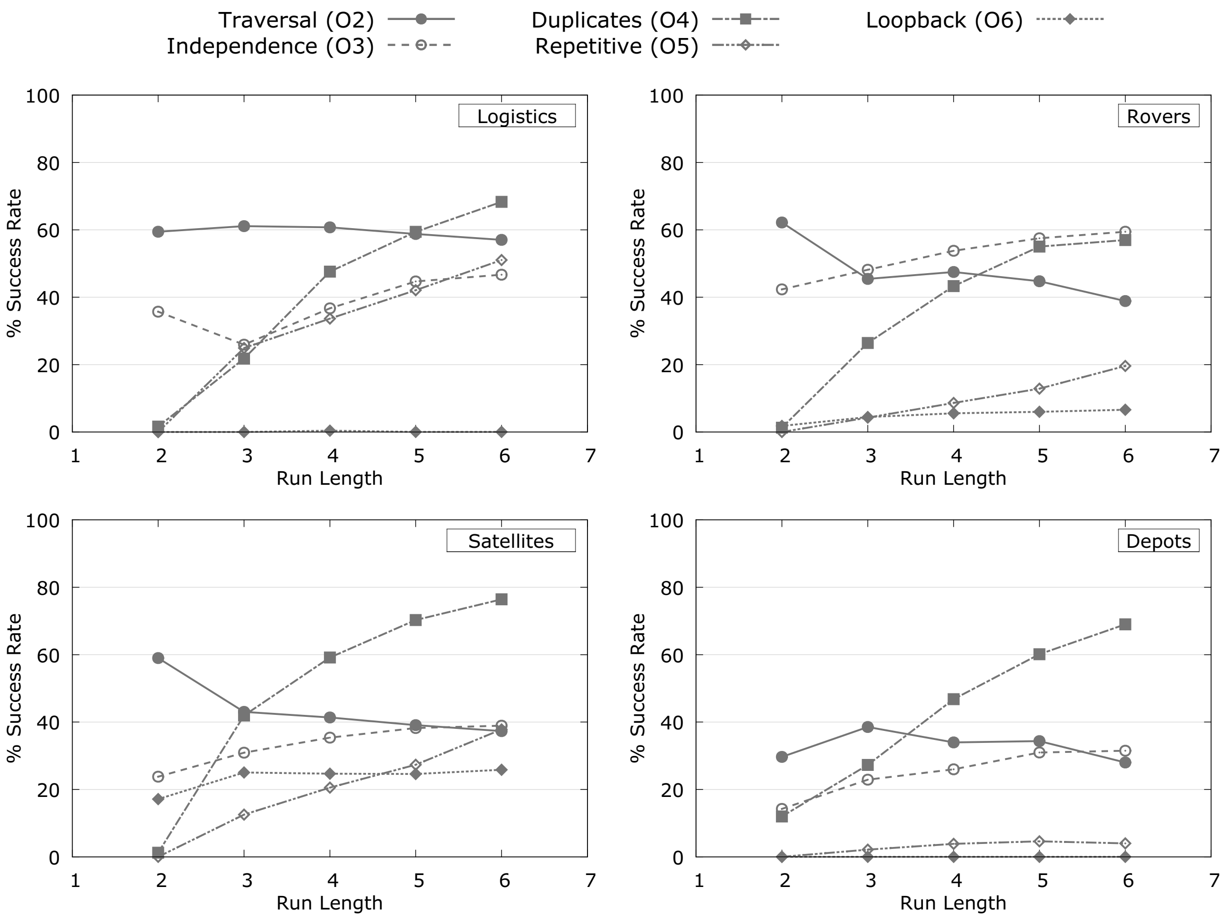


Figure 7.4: The percentage of runs removed by traversal optimisations in the IPC domains.

We analyse each optimisation in turn. As in the Parcel Delivery domain, the Traversal optimisation is very effective removing 35% of initial runs in Depots and 60% in the remaining domains. It is the most effective optimisations for short runs, which is particularly appealing, since the effects of pruning at this stage are amplified. The effectiveness of the optimisation gradually reduces in all domains as runs become longer, for reasons presented above.

The Independence optimisation is the only other optimisation that removes runs of length 2 in all domains. While always being less effective than Traversal initially, its success rate increases with the run length. As runs become longer it is possible to find larger sets of independent operators, as opposed to sets of only 2 operators initially. As the runs increase, so does the potential size of these sets, yet interestingly, the increase slows as run length increases. We attribute this to the fact that it is less likely that larger sets of independent operators will be found in longer runs.

The Loopback optimisation is poor in all but the Satellites domain where it has a success rate of approximately 20%. In order to understand this behaviour we detail the `calibrate` operator from the domain description:

```
OPERATOR: calibrate( ?s – satellite  ?i – instrument  ?d – direction )
     PRE: { on_board(?i,?s), target(?i,?d), pointing(?s,?d), power_on(?i) }
    POST: { calibrated(?i) }
```

The calibration status of a satellite is represented by the `calibrated` literal. Notice that none of the preconditions are consumed, meaning that the `calibrate` operator can be applied in a repeated fashion with *identical* parameters, resulting in the creation of runs of length 2 with identical `calibrate` operators that contain loops. This feature is also present in the Rovers domain, but in no other.

A similar initial deviation in success rate is present for the Duplicates optimisation in the Depots domain. In all other domains the initial success rate is 0 yet in Depots it is approximately 13% implying that duplicate runs of length 2 are found. For this to be the case we must be considering duplicate candidate operators. When identifying candidate operators we identify each operator in the domain and enumerate all possible bindings with literals in the final specification of the run. It is possible in the depots domain to bind separately to two different literals and obtain identical bindings. Hence, even though we bind to unique literals we identify the same binding set, and therefore a duplicate operator.

Finally we analyse the effectiveness of the Repetitive operator optimisation. In all domains the optimisation begins with a success rate of 0 and increases in a linear fashion. However there is significant difference in the rate of increase. In the Logistics and Satellites domain the optimisation ends with success rate of approximately 40%, yet in the Depots domain the success rate is below 5%. To better understand why the Repetitive optimisation in the Depots domain is less effective at these shorter run lengths we compare the domain operators to the Parcel Delivery operators. Recall that in the Parcel Delivery domain it is possible for an agent to repetitively apply a `pickup` or `drop` operator (if the appropriate conditions hold). Each of these operators does not consume anything it is dependent on. If each agent were only able to hold a single parcel at any time then the repetitive application of these operators no longer holds. The effect is that repetitive operators are identified for shorter runs. In the Depots domain a `hoist` is not `available` once it has lifted a `crate` from a surface or truck with the implication that a hoist may only lift a single crate at any time. This restriction results in far fewer repetitive actions found, particularly for shorter runs. For longer runs, the repetitive combination of actions (`Load`,`Unload` or `Lift`,`Drop`) are identified.

We conclude our In Turn empirical results by emphasising the importance of removing shorter runs effectively. Figure 7.5 below presents the % of complete runs removed by the Traversal and Duplicates optimisations in the Satellites domain over the unoptimised core synthesis results.

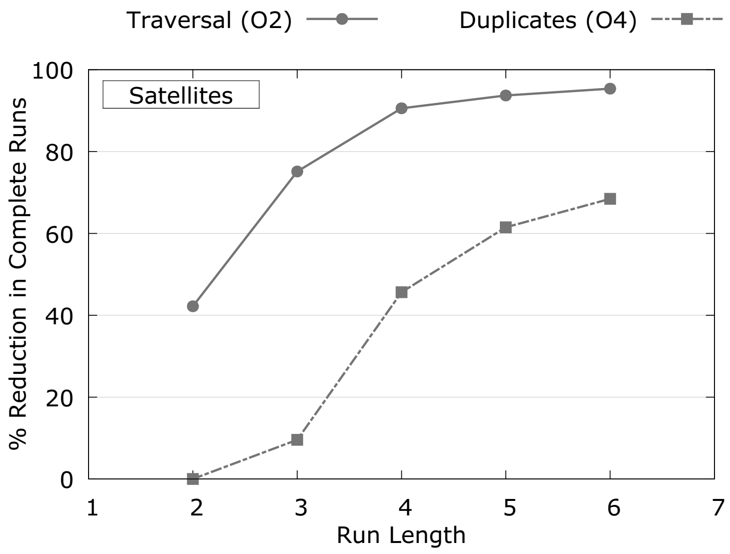


Figure 7.5: The percentage reduction in number of complete runs when using the Traversal and Duplicates optimisations in the Satellites domain.

Initially removing runs is advantageous. The success rate of the traversal optimisation is above the Duplicates optimisation for runs of length 2 and 3 only. Eventually the Duplicates optimisation will remove a higher percentage of complete runs than the Traversal optimisation, yet there are benefits to a higher initial success rate.

To conclude, of all the optimisations both Traversal and Independence were successful in all domains. As runs become longer the Duplicates optimisation becomes more successful, until for runs of length 6 or more it is the most successful optimisation in all domains. This comes at a cost as the more runs considered, the larger the space required to store these runs, and subsequently the more complex the process of checking for Duplicates. The Repetitive optimisation varied: very strong in the Logistics and Satellites domains, and less so in the Rovers and Depots domains. While the Loopback optimisation is not very effective in comparison to the other optimisations it is still able to eliminate runs, particularly as the length of the runs increase.

7.4.3 Sequential Traversal Optimisation Results

The second part of our traversal optimisation analysis investigates how effective the optimisations are in unison. Our presentation follows as before: we begin by presenting results and discussion for the Parcel Delivery domain, and then follow this by the results of the IPC domains. We abbreviate each of the optimisations as Tr (Traversal [O2]), I (Independence [O3]), D (Duplicates [O4]), R (Repetitive [O5]) and L (Loopback [O6]), and specify combinations of these optimisations as L+I+Tr. We begin with an analysis of the number of complete runs produced during traversal. The results, averaged over 100 randomly generated conflict specifications in the Parcel Delivery domain, are presented in Figure 7.6. We bound the length of runs to under 15, and terminated traversal if the number of runs exceeded 20000.

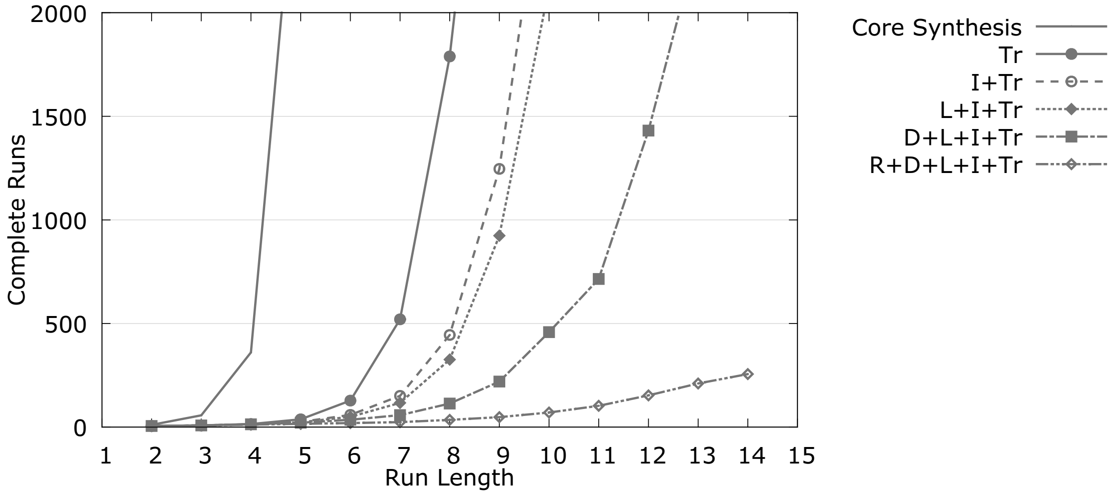


Figure 7.6: Cumulative reduction in the number of complete runs as optimisations are sequentially enabled during traversal.

It is clear that the optimisations provide a significant cumulative benefit over core synthesis. The core synthesis process with no optimisations exceeds 20000 complete conflict runs of length ≤ 6. We have graphically represented up to 2000 runs for clarity. The unoptimised process violates this bound at run length 4, the Traversal optimisation doubles this limit to runs of length 8, while Independence and Loopback allow runs of length 9 and discarding duplicates increases the maximum run length to 12. Finally, by including repetitive operators we significantly reduce the number of runs. The result is that, with all optimisations enabled, we find 256 complete runs of length up to 15.

We briefly discuss the variance of these results. Figure 7.7 individually presents the Traversal and cumulative Traversal and Independence results. The shaded area represents the range between the maximum and minimum values recorded, and the error bars present the standard deviation. Notice that the minimum number of runs is 0 since for some conflict specifications the optimisations remove all runs from consideration.

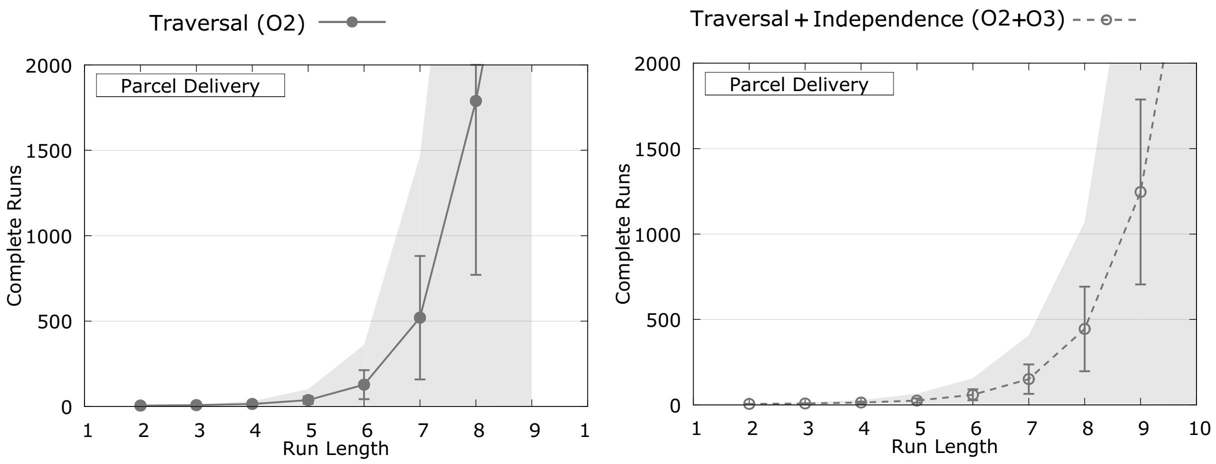


Figure 7.7: Variance in the run count with the Traversal and Independence optimisations. The shaded area represents the range between maximum and minimum values.

As with the In Turn optimisation analysis it is difficult to predict what effect an optimisation has on the traversal process. An analysis of the success rate of each optimisation allows us to better gauge performance. Additionally, since the optimisations are sequentially applied we are able to identify what impacts preceding optimisations have on a subsequent's success rate. Figure 7.8 presents the results split into two charts.

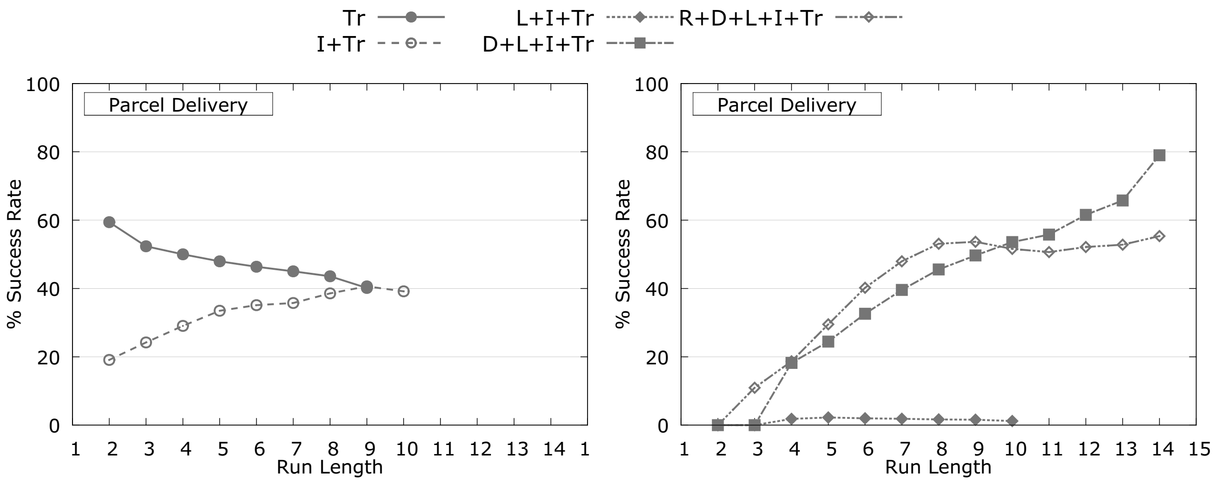


Figure 7.8: The success rate of sequentially applied traversal optimisations in the Parcel Delivery domain.

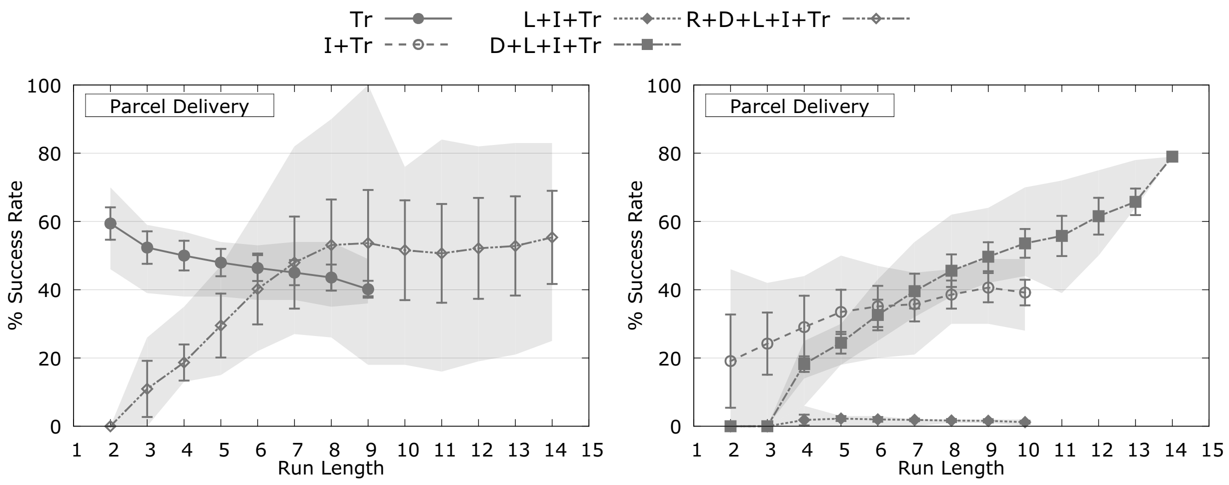


Figure 7.9: The average success rate of sequential optimisations with the area between the maximum and minimum shaded and standard deviation error bars depicted.

We plot the *relative* benefit of introducing each new optimisation: it is not the case that the combination of Loopback, Traversal and Independence (L+Tr+I) performs worse than just Traversal, but rather that the improvement by introducing Loopback is minor. This confirms our initial results: in the Parcel Domain there is little to be gained through the Loopback optimisation. Sequences on the chart terminate at different run lengths, on account of the maximum run limit of 20000. Only on the introduction of the Duplicates optimisation is the limit not reached. The increase in run lengths indicates that fewer runs are considered as each optimisation is introduced.

By comparing the Sequential success rate results with the In Turn results in Figure 7.3 it is clear that the optimisations are not independent. Only the Traversal optimisation maintains its success rate as it is the first optimisation applied. The sequential introduction of the Independence optimisation shows a reduced success rate since the reverse application of the Traversal optimisation removes runs of length 2 with independent operators. As run lengths increase the difference between the success rates of the Independence optimisation for the In Turn and Sequential results decreases. The Duplicates optimisation is also affected by preceding optimisations. When executed sequentially the Independence and Traversal optimisations reduce the success rate of the Duplicates optimisation, emphasised by the drop in success rate of the Duplicates optimisation from approximately 20% when applied alone to 0% when applied sequentially for runs of length 3, although the success rate does increase as run length increases. Similarly, the Repetitive optimisation increases for runs up to length 8 but then levels between 50% and 60%.

To summarise the performance of sequentially applied optimisations in the Parcel

Delivery domain we illustrate the deviations from the mean in Figure 7.9. The shaded area is the bound between maximum and minimum values and the error bars represent the standard deviations. There are two conclusions to be drawn from these charts. Firstly, the success rate metric reduces the deviation due to its independence from the conflict state specifications and number of complete runs generated. Secondly, from the samples taken over 100 conflict state specifications we are afforded tight bounds on the success rate of all optimisations except for the Repetitive optimisation. Where the other optimisations end with deviations of under 5% from the mean the Repetitive optimisation had a deviation of 13%. From this we conclude that the Repetitive optimisation has more of a dependency on the input conflict state specifications in the Parcel Delivery domain than the other optimisations.

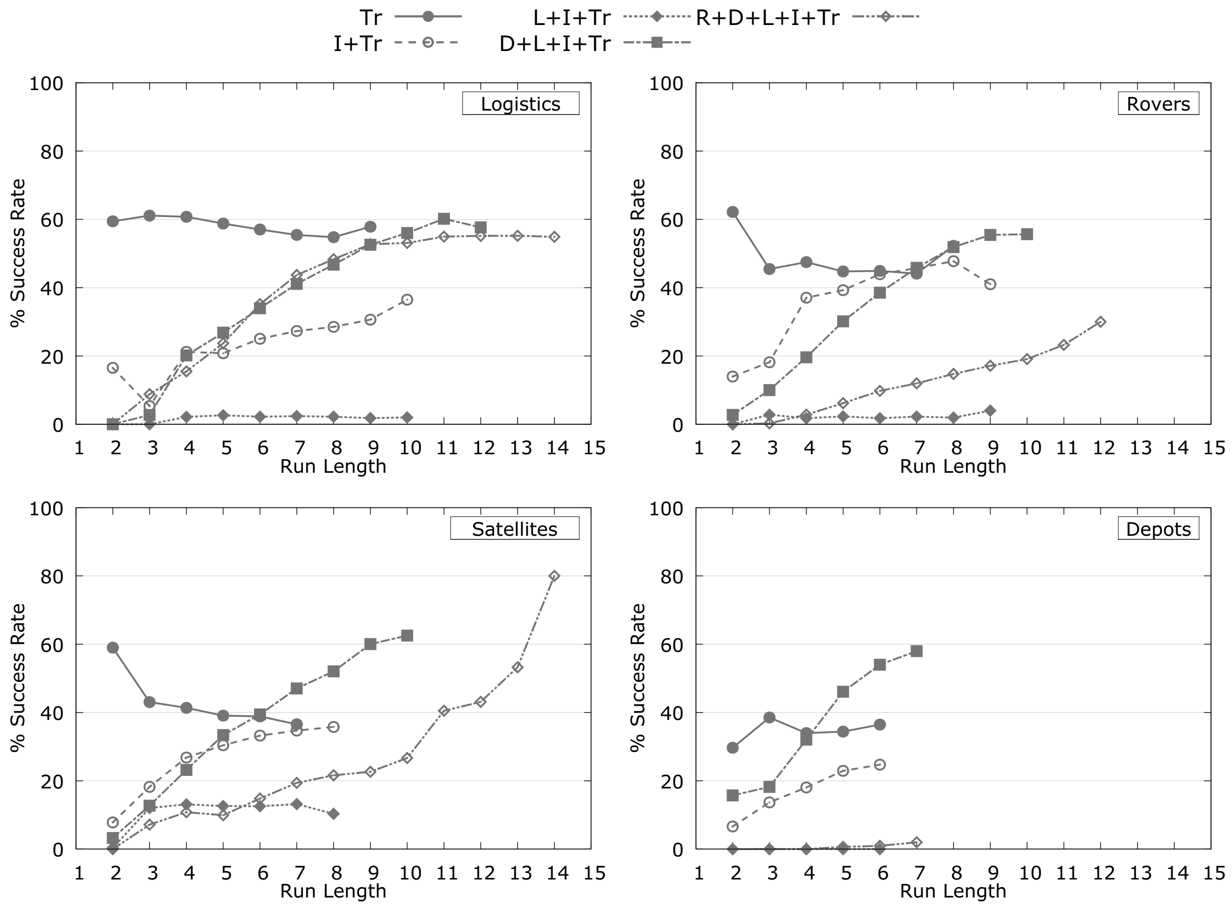


Figure 7.10: The average percentage success rate achieved through the sequential introduction of optimisations in the IPC domains.

In Figure 7.10 we present the success rate results for sequential optimisations in the IPC domains with traversals limited to 20000 complete runs. The Logistics domain is the simplest for traversal purposes followed closely by the Satellites domain. With all optimisations enabled, traversal in both domains is not inhibited by the run limit. For runs of length up to 10 the success rates of optimisations in the Rovers domain is very similar to those in the Satellites domain, yet for longer runs the Rovers domain is bound for two reasons: Firstly, the repetitive optimisation is not as effective in the Rovers domain leading to fewer runs being discarded. Secondly, and more importantly, the branching factor of the search traversal in the Rovers domain is larger, due to the 9 domain operators as opposed to Satellite's 5. Traversal in the Depots domain is poor in comparison. Here no runs exceed length 7 without violating the imposed run bound.

There is significant room for improvement in the way optimisations remove runs from consideration. For example an entire third class of optimisation could deal with intelligently constructing the set of candidate operators instead of simply enumerating all operators and subsequently attempting to reduce the run set, thereby reducing the need to construct runs that are subsequently removed. While we present further discussion of future work in Section 9.3 we briefly illustrate the benefit of devising optimisations that perform well initially. Figure 7.11 is a cumulative plot highlighting the percentage of runs removed by the sequential addition of each optimisation, applied in the Logistics domain. The Processed Runs segment refers to the percentage of runs that were not removed by a traversal optimisation. The percentage of runs re-

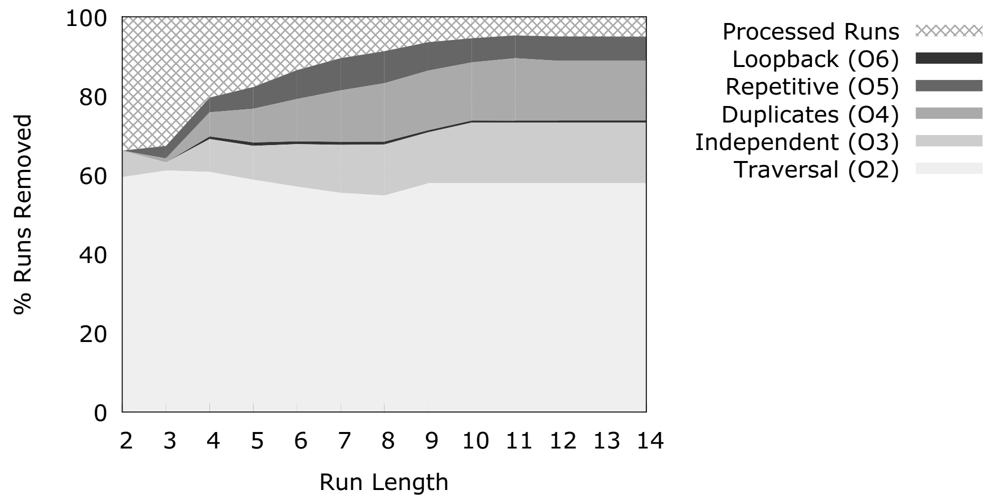


Figure 7.11: The cumulative percentage of runs removed by sequentially applied traversal optimisations in the Logistics domain.

moved by the Traversal optimisation itself is impressive, not because it had the highest success rate overall, but because it had the highest success rate initially. If the goal of any future optimisations is to reduce the number of complete runs then it pays to be effective initially for short runs.

7.4.4 Planning with Limited Knowledge

We now present an analysis of the Planning with Limited Knowledge optimisation performance. This optimisation is performed on each run prior to grounding, and runs are assumed reachable if the optimisation is successful.

We present the success rate of the optimisation not as a single metric but as the number of *attempts* made by the optimisation and the number of these that are *successful*. We ran the Parcel Delivery domain on grids of size 2x2, 3x3 ...21x21, and ran each of the IPC domains on the same set of 20 problem instances. We kept running totals of attempts and successes across 100 randomly generated conflict specifications for each, and present the results in Figure 7.12. For the purposes of this test we limited the maximum number of traversal runs to 2000.

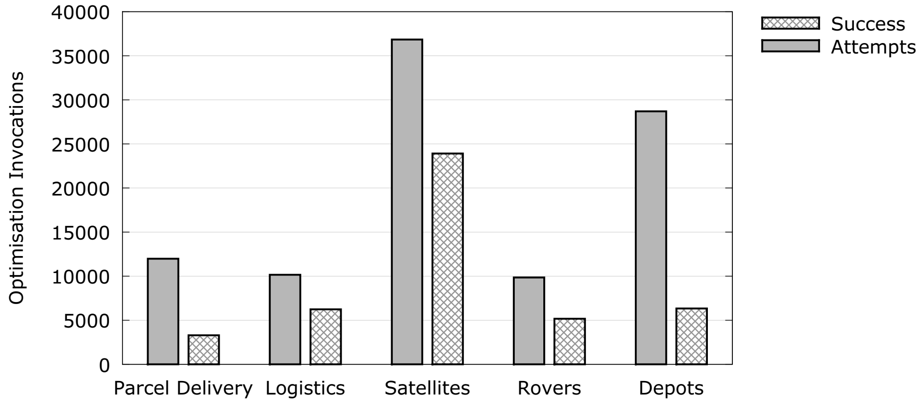


Figure 7.12: Planning with Limited Knowledge attempts and successes.

The success rate of the optimisation is good, ranging from 22% in the Depots domain to 64% in the Satellites domain. Both the Depots domain and Parcel Delivery domains have low success rates for this optimisation, highlighting the fact that in these domains additional problem instance knowledge is required to find alternative plans during reachability analysis. In the Depots domain this might correspond to what additional `hoists` are at a particular `distributer`, or the number of `palettes` available to be loaded. In the Parcel Delivery domain this knowledge is related to the topology of the grid world. In these situations the reachability optimisation is less effective.

Compare this with the Satellites or Logistics domain where there are fewer constraints imposed by the problem instance. A satellite can change direction at any point, irrespective of where it is facing. Similarly, vehicles in the Logistics domain can move to any location arbitrarily. The lack of reliance on additional domain knowledge results in the optimisation being more effective in these domains.

7.5 Part C - A Model Checking Comparison

This final evaluation part identifies whether our conflict-rooted synthesis implementation (CRS) is superior to a model checking approach that utilises NuSMV. We are interested in the performance of conflict-rooted synthesis as a whole and hence treat the approach as a black box component without configuring or adjusting internal parameters. Our analysis is a two step process: we present a theoretical comparison and follow this with empirical results to reaffirm our theoretical conclusions. The aims and objectives of this empirical evaluation component are:

1. Compare the *computational requirements* of CRS with NuSMV to determine the benefits of adopting the proposed approach.
2. Compare the *quality* of the synthesised norms.

We begin with the theoretical comparison of the approaches.

7.5.1 Theoretical Comparison

We perform a complete theoretical comparison from the input to the approach to the output produced. We begin by analysing the domain and conflict state specifications provided to each algorithm. We compare the expressiveness of each representation and comment on how properties of the input affect the norms produced. Finally, we illustrate that the dependence on initial system states is a further key difference between the two approaches.

We begin by revising the model checking approach. van der Hoek et al. (2007) showed how a model checker can be used to solve the problem of prohibitionary norm synthesis. The basis of this approach is that by encoding conflict state avoidance as a CTL expression a model checker can find a computation which is conflict-free. Given this computation a set of norms are synthesised that regiment the behaviour of all agents. We showed that if we wish the computation to satisfy conditions regarding conflict-free state reachability, these can be encoded into the checker using fairness constraints. The resulting computation avoids conflict states but ensure that all specified conflict-free states are reachable.

7.5.1.1 Domain Representation

Let us begin by analysing the domain representation used by each approach. Conflict-rooted synthesis takes, as input, a set of PDDL files. We utilise only the domain operators in order to synthesise generally applicable norms and then incorporate the problem instance when checking goal reachability. NuSMV requires the specification of a finite state automaton describing the global behaviour of the system, including the behaviour of agents within it. There are no abstract operator schemata: actions are encoded as transitions between states in the model where each edge identifies a unique action. The automaton is constructed from an initial state and fully describes the system from that point. There are two significant differences between the domain representations adopted by each approach:

1. CRS uses parameterised operator schemata providing a more concise means of specifying agent actions. NuSMV models are more specific and verbose relying on a propositional representation and explicit transition relations to define the finite state automata.
2. CRS takes advantage of the separation between the operator set and initial state information to synthesise norms that are independent of the initial state of the system, while NuSMV is implicitly bound to a single initial system state from which the model was generated. The result is that norms synthesised are domain-specific in CRS, and problem-specific in NuSMV.

7.5.1.2 Conflict Specification

Conflict-rooted synthesis restricts the expressiveness of its conflict specification, allowing only ungrounded state specifications without any temporal or branching relations. The resulting conflict specifications are state-centric, and cannot be used to model conflict based on action. For example, we cannot define a conflict state to be the third sequential state where an agent in the Parcel Delivery domain is holding the same parcel.

Model checking provides a significant increase in expressiveness as CTL includes state information and temporal modalities to reason about future states of the system. ATL is even more expressive as it can reason about specifications brought about through compliance by a subset of the system agents. These specifications are strictly more expressive than our approach, yet are limited in not supporting any quantification over variable symbols. A concise conflict specification such as:

$$\{\mathtt{agentAt}(\mathtt{a}_1,\mathtt{X}),\mathtt{agentAt}(\mathtt{a}_2,\mathtt{X})\}$$

must be enumerated beforehand to remove the variables, as in:

$$\{\texttt{agentAt}(\texttt{a}_1,\texttt{node}_1),\texttt{agentAt}(\texttt{a}_2,\texttt{node}_1)\}$$
$$\{\texttt{agentAt}(\texttt{a}_1,\texttt{node}_2),\texttt{agentAt}(\texttt{a}_2,\texttt{node}_2)\}$$
$$\{\texttt{agentAt}(\texttt{a}_1,\texttt{node}_3),\texttt{agentAt}(\texttt{a}_2,\texttt{node}_3)\}$$
$$\ldots$$

Additionally a CTL model checker is required even if the conflict specification does not require temporal or path modalities, since the expression over the social objective is

$$S, s_0 \models \mathsf{EG}\varphi$$

where φ represents the social objective. Even if φ were a simple Boolean logical expression, the resulting expression is still a CTL expression due to the preceding path and temporal quantifications, implying that CTL model checking is always required and thereby reducing the set of appropriate model checkers.

7.5.1.3 Encoding Reachability

A comparison is not complete without mentioning the mechanisms each approach adopts to encode reachability. This is in some ways an unfair comparison. Conflict-rooted synthesis is an approach specifically engineered with reachability analysis as a primary consideration, while a model checker is a general purpose tool in which reachability analysis is encoded into the problem. Regardless of these differences, it is an important consideration since reachability analysis is essential when synthesising useful norms.

CRS ensures goal reachability by identifying conflict-free alternative plans for each run generated. It is only interested in the conflict-free states that are direct precursors or successors of conflict states and not interested in showing reachability between all conflict-free states. In order to ensure similar reachability requirements using a model checker an enumeration of the set of conflict-free states is required. van der Hoek et al. (2007) encode focal states into the social objective, yet since we assume that all conflict-free states are focal it follows that every *joint* conflict-free state is focal and must be encoded as input into the model checker. There are two issues with this approach:

1. The resulting expression is potentially very large as it grows with the number of joint conflict-free system states.

2. An automated mechanism is required to identify these conflict-free states so that they can be fed as input into the model checker.

The second point above is important. Goal reachability must ensure that goals that were reachable in the original system are still reachable in the normative. Put differently, it is not the case that all conflict-free states must be reachable since the resulting norms might be overly restrictive. Requiring that a conflict-free state that is not reachable in the original system be reachable in the normative system is clearly incorrect. It is not sufficient to list all conflict-free states, but rather the set of *reachable* conflict-free states. The task is therefore not simply an enumeration of the state space but rather a *plan* enumeration in order to identify all of the conflict-free states that are reachable. Conflict-rooted synthesis has three advantages over this approach:

1. The search for conflict-free states is an intrinsic component of the algorithm so no additional search is required.
2. It deals with unground incomplete state specifications, avoiding a compulsory complete state enumeration.
3. It maintains the exact reachability properties of the original system, ensuring that candidate norms are not too restrictive.

7.5.1.4 Social Norms Produced

Let us compare the norms produced by each method. Recall that the output of the model checking approach is a single computation dictating a master plan: a sequence of actions, for all agents in the system, where no deviation is allowed. We argue that there are significant issues with this approach:

1. Agent autonomy is greatly reduced since agents are forced to follow a single course of action, regardless of whether this course becomes impossible to achieve.
2. The norms are prohibitions conditional on complete states and are not abstracted away from the state representation. The resulting norms are in no way abstracted away from the propositional domain representation, often resulting in more norms than system states.
3. The norms prohibit all actions not prescribed by the master plan, which either requires a norm representation language able to deal with negations of actions,

or alternatively requires an enumeration of all prohibited actions possible in each system state.

4. The master plan revisits system states. In order to decide what action to perform, agents must keep a history of what actions have already occurred.

5. If an agent deviates from the master plan then the sequence is broken and it is unclear what behaviour the other agents should perform.

Conflict-rooted synthesis does not suffer from these drawbacks. We preserve agent autonomy by prohibiting only the actions leading to conflict and do not pre-specify behaviour in conflict-free states. Agents are able to achieve any goals, or to adapt to new domain knowledge without recalculating the set of norms. Norms are conditional on sets of system states, resulting in far fewer, more concise norms.

Furthermore, since the computation produced by the model checker contains loops, every conflict-free state is visited infinitely often. The property that every conflict-free state is reachable from every other is unrealistic in practice.

Proposition 7.5.1. *Reachability between focal states is unrealistic in systems where goal states are terminal.*

Proof. Consider a simple version of the Parcel Delivery domain where a single parcel is placed, and the agent that delivers this Parcel receives maximum utility. This configuration is similar to achieving check mate in a game of chess. Once this state is achieved the system no longer continues since the core objective has been satisfied. If we consider a single computation that satisfies this objective then it is clear that once the system is in this terminal state it is no longer possible to adapt it to return to the initial state. Since these terminal states were reachable in the original system they must still be made reachable in the normative system. Enforcing an infinite computation that dictates a master plan cannot satisfy this requirement. ■

7.5.1.5 Space and Time Complexity

In this work we make the assumption that the conflict state specification identifies a set of states that is smaller than the conflict-free set. We traverse this conflict space, avoiding grounding where possible, and incorporate state abstractions to reduce the search space further. All these factors ensure that we perform an efficient search avoiding a compulsory enumeration of the state space.

NuSMV is unable to take advantage of these factors since the domain model is fully enumerated. Furthermore, it searches the entire conflict-free state space rather than an abstract representation of the conflict state space. The model checker scales poorly with respect to the size of the domain since the search for a single computation through the entire conflict-free state space is akin to finding a master plan for all agents. On small domains we expect the model checker to be very efficient due to two factors:

1. NuSMV is a mature model checker adopting techniques that have been thoroughly researched. The implementation is native and highly optimised.
2. The enumeration of the conflict-free states is performed prior to model checking and the propositional finite state automaton reduces the need for grounding.

It is not possible to fully analyse how these two approaches compare on practical domains without performing an empirical evaluation. Next we present the empirical results of our tests to attempt to identify whether conflict-rooted synthesis provides a tangible benefit over a model checking approach to norm synthesis.

7.5.2 Part C - Evaluation Plan

We begin by investigating how CRS and NuSMV compare first for small instances of the Parcel Delivery domain, subsequently on larger instances, and finally on the IPC domains. In this entire part we evaluate conflict-rooted synthesis with all optimisations enabled and monitor the computational time and number of norms generated.

For each domain, we construct an evaluation set for each unique tuple of input. As neither of the approaches are subject to non-determinism, and since the problem instances are provided, we limit each set to 10 iterations, where each iteration represents a single run of each synthesis approach with identical input domains, problem instances and conflict specifications. The comparison metrics for each test are then aggregated over all instances.

The input for each approach is a two dimensional space, where the conflict state specification parameter is varied along with the problem instances of a particular domain. Average behaviour over all domains can then be computed by aggregating the results obtained from each domain. Conflict specifications are randomly generated as before. Table 7.7 presents a summary of the configuration for this component.

Domains	Parcel Delivery World	IPC Domains
Problem Instance Set	Grid sizes from 2x2 ... 10x10	IPC Domain Sets
Conflict Specification	100 Random Per Problem	100 Random Per Problem
Optimisations	All	
Measured Metrics	Computational Time Number of Synthesised Norms	

Table 7.7: Part C - Empirical Evaluation Setup for Model Checking Comparison

7.5.3 Empirical Results

We take this opportunity to emphasise that the analysis of these two approaches is not a direct head-to-head comparison. Section 7.5.1 shows that, even though both approaches produce norms, they do so in very different ways. Neither the input to the algorithms, the methods adopted nor the resulting norms are identical in representation or expressivity. In order to standardise the input to each process we provide CRS with additional domain knowledge regarding the number of objects in each problem instance. This allows CRS to bound the traversal process, producing norms that are guaranteed to hold in all instances of the domain with objects less than or equal to the limit provided, and to provide a better comparison of the output of each of the algorithms. We present the results next.

7.5.3.1 Computational Time

We first illustrate the benefits in computational time afforded by CRS in the Parcel Delivery domain and subsequently in each of the IPC domains. We begin by monitoring the change in computational time when the size of the grid world is increased in a single dimension, with the conflict specification:

$$S_C = \{\texttt{agentAt}(\texttt{a}_1,\texttt{node}_2), \texttt{agentAt}(\texttt{a}_2,\texttt{node}_2)\}$$

where nodes are numbered linearly according to the grid topology. We do not consider parcels in the domain and restrict agents to the `move` operator. The results are presented in Figure 7.13.

The slight increase in CRS computation times with increasing grid size is due to increased processing times required to parse the larger problem instances. The same number of runs are generated for each of the problem instances, and an identical number of reachability checks are performed in all but the 1x2 grid. NuSMV scales poorly, taking over 7 seconds to synthesise norms in the largest grid size, as opposed to the

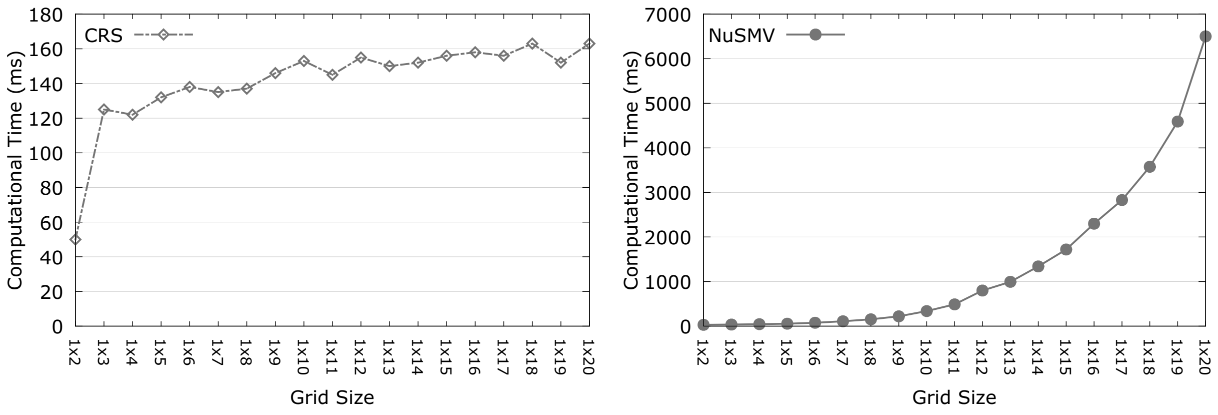


Figure 7.13: A comparison of the increase in computation times produced by increasing the size of the grid in the Parcel Delivery domain.

160ms that CRS took. As we have shown theoretically this is expected since NuSMV's search is dependent on the conflict-free state space. As the graph size increases more complex and longer master plans are investigated in the hope of finding a single computation that traverses all conflict-free states. We verify this analysis by observing the number of norms generated by each approach. The conflict-rooted synthesis approach generates exactly 2 norms in each problem instance, independent of the underlying topology. The number of norms produced by NuSMV is presented in Figure 7.14.

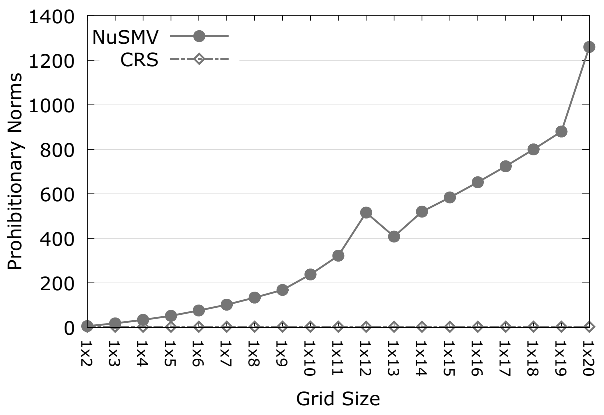


Figure 7.14: The number of norms produced by NuSMV for increasing grid sizes.

The model checking results are deterministic in the sense that the same number of norms are generated for repeated runs of the model checker, however they are not minimal as can be seen by the sharp increase in norms for the 1x12 size grid. The sheer number of norms produced illustrates why the model checking approach scales so poorly. By producing over 1200 norms in a 1x20 grid the model checker has searched and found a computation composed of 1200 actions, which is far more than the number of states in the system. This search is dependent on the size of the enumerated joint conflict-free state space. By increasing the grid width 10 fold from 2 to 20, we increased the size of the set of joint states 100 fold from 4 to 400 and the set of synthesised norms produced by the model checker from 6 to 1260.

Let us now analyse the change in computational time produced by increasing the grid size in two dimensions. We begin with a 2x2 grid size and increase to a 10x10 grid size. Furthermore, instead of the *static* conflict specification used above, we introduce a *dynamic* alternative:

$$S_C = \{\texttt{agentAt}(\texttt{a}_1,\texttt{X}), \texttt{agentAt}(\texttt{a}_2,\texttt{X})\}.$$

As the grid size increases so too does the conflict state space, since we are prohibiting concurrent access in *all* nodes of the graph. We present the computational time results in Figure 7.15. Notice we have presented a logarithmic plot of the results, and we omit the dynamic NuSMV results since they are indistinguishable from the static.

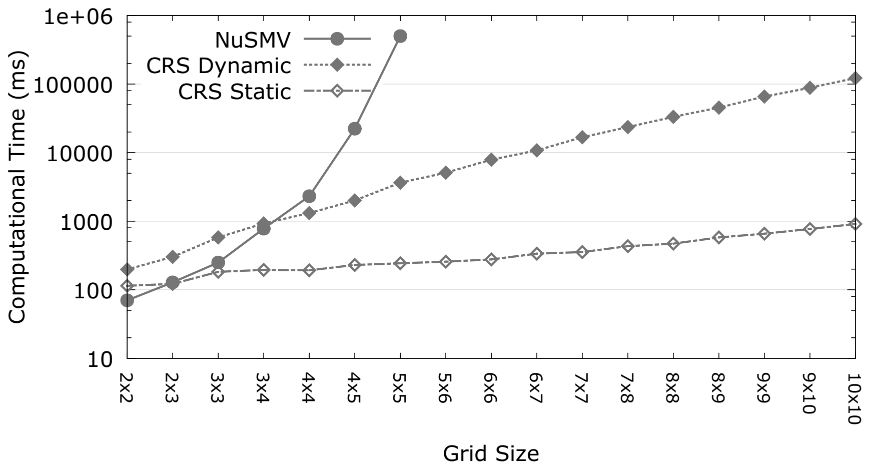


Figure 7.15: A comparison of the increase in computation times produced by increasing the size of the grid polynomially in the Parcel Delivery domain.

The largest grid size with which we are able to synthesise norms using NuSMV in the Parcel Delivery domain with agents only permitted to move, is 5x5. This computation takes 501 seconds to complete, with the model checker consuming over 6 gigabytes of memory to construct the model[1]. The implementation is not able to deal with the increase in the enumerated joint state space with the complexity unable to be bound by an exponential function of the form $O(c^n)$ where c is a constant and n increases with the size of the system. CRS is a less efficient implementation, but in practice the

[1]In order to determine this figure additional memory was installed in our test machine.

computation time required for larger grid sizes grows more favourably. For systems with a small number of states the model checker's efficiency of implementation is able to produce output in reasonable time, but even the slightest increase in system size sees NuSMV's time increase exponentially. The approaches are in different complexity classes. NuSMV cannot be bound above by any exponential function of the form c^n, while CRS is bound above by the same function and is $O(c^n)$.

Finally, we comment on the difference between static and dynamic conflict state specifications in CRS. The size of the grid worlds grow polynomially as we increase the grid size. For the dynamic conflict state specification this implies at least a corresponding polynomial increase in the number of reachability checks performed (the traversal process is identical for both). However for the static conflict specification we perform the identical number of reachability checks for all grid sizes, implying an increase in the time taken to perform each reachability check. This is attributed to the increase in the size of the problem instances, resulting in more time required for FF to instantiate the operator schemata into ground, STRIPS-style actions prior to planning. Since the size of the domain is increasing polynomially, so too is the set of actions and the corresponding planning initialisation time.

We conclude computational time analysis by presenting the results for each of the IPC domains in Table 7.8. We present the mean time over 100 randomly generated conflict specifications, shading cells depending on whether all specifications were solved, a portion were solved, or none were solved. In these tests we limited the plan time for each reachability analysis check of a grounded run to 5 seconds in order to bound the amount of time required to run the tests, and to simulate realistic computation time requirements[2]. Recall however that the anytime nature of our algorithm allows us to be certain of reachability up to the point where the planner exceeded the time limit, ensuring that plans of length less that or up to the current point are guaranteed to be reachable. NuSMV was not bound in any way except by the resources of the test machine. Additionally, NuSMV performed no reachability checks on the constructed model, and the times reported do not include the PDDL to SMV domain translation. As such, the NuSMV results are best case times.

The Logistics domain is the simplest test domain, and NuSMV is able to solve the first 9 problem instances while CRS solves all problem instances with no bounds exceeded. The Satellites domain is particularly challenging for both approaches, with

[2]The figure of 5 seconds appropriately limits the execution time of the planner. The heuristic search employed by FF typically requires far less time to complete, while the fallback breadth first search requires far more. This time limit is sufficient to allow the heuristic search to complete.

– No Results ░ Partial Results [Planner Timeout] ▓ Partial Results [Run Limit]

	Times (ms)							
	Logistics		Satellites		Rovers		Depots	
Problem	NuSMV	CRS	NuSMV	CRS	NuSMV	CRS	NuSMV	CRS
1	54814	206	92	184	169	652	679	2596
2	56169	193	9874	2306	86	1326	96794	58759
3	53182	158	–	18428	2030	3281	–	97152
4	58982	180	–	5666	2345	6144	–	96546
5	54287	165	–	85903	79594	11362	–	63683
6	53207	189	–	33953	–	15171	–	63145
7	52723	197	–	21196	–	34959	–	57165
8	53604	161	–	32960	–	316983	–	62510
9	53993	170	–	41980	–	62229	–	56535
10	–	315	–	123427	–	27715	–	56246
11	–	276	–	53836	–	81258	–	60681
12	–	319	–	176536	–	23520	–	60352
13	–	219	–	150137	–	41787	–	57086
14	–	232	–	239478	–	94441	–	60342
15	–	452	–	718131	–	30599	–	50723
16	–	1438	–	565845	–	16710	–	44755
17	–	608	–	343207	–	25140	–	61090
18	–	1471	–	189672	–	26405	–	55948
19	–	830	–	238083	–	86920	–	51243
20	–	502	–	378319	–	91576	–	52530

Table 7.8: IPC computation times. Empty cells represent runs where NuSMV produced no output. Lighter shaded cells represent runs where CRS solutions were approximate.

NuSMV solving the first two domains, and CRS the first 4. Both approaches fare better in the Rovers domain, with NuSMV solving 5 and CRS half of the problems. The results for the Depots domain confirm what we deduced from the traversal optimisation analysis: Depots is challenging to any search based approach. Both NuSMV and CRS solve the two simplest Depots problems.

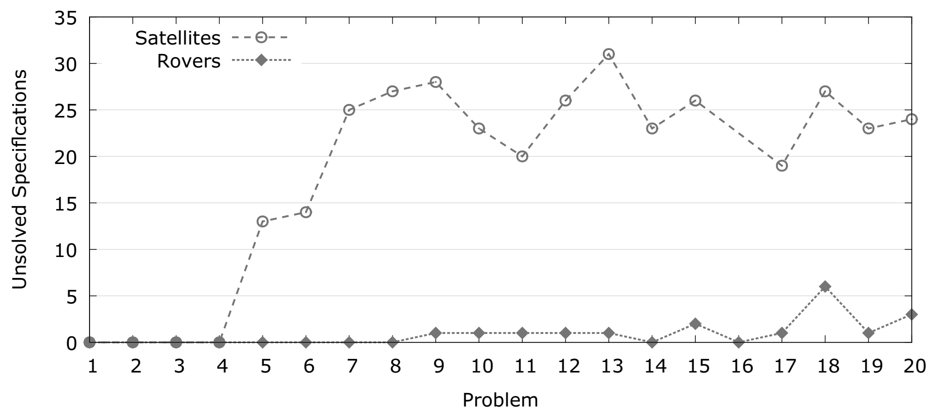


Figure 7.16: The number of specifications where reachability analysis is not completed due to planner timeout.

It is not clear from Table 7.8 how many of the conflict specifications could not be solved by CRS. The shaded cells simply indicate that at least one of the conflict specifications resulted in the planner exceeding the time limit. Figure 7.16 details the number of conflict specifications (out of the total 100 randomly generated) that are unsolved for each problem in the Satellites and Rovers domains. We have omitted the Depots and Logistics domains as the planner did not time out in any of these.

Conducting reachability planning in the Satellites domain is a complex task, with CRS only able to solve 77.5% of the conflict specifications encountered. Planning in the Rovers domain is considerably simpler, with the number of unsolved specifications topping 5 only for problem 18. In total CRS solved 99.1% of the specifications across the Rovers domains. Given more time to plan, reachability analysis would have completed the remainder.

Finally, we discuss the results obtained for the Depots domain. None of the reachability analysis planning steps exceeded the 5 second limit, yet only the first two problem are solved entirely by CRS. In all other problems CRS is limited not by planning time, but by run limit. The result is that, in at least one conflict specification in each subsequent problem instance, the traversal process exceeds the run limit. Reachability analysis continues on the set of produced runs, but there is no guarantee that the resulting norms ensure goal reachability in the normative system. One reason the planner takes more time on certain problems is that when the planning heuristics fail to identify a plan the planner falls back onto a far less efficient A^* search of the state space. The tendency for the Depots domain to exceed the practical run limit is also not surprising as the traversal optimisation analysis showed that the run count grows far more quickly in the Depots domain than any other. Even with the additional problem instance knowledge assumed in these tests CRS is not able to fully solve 18 of the 20 Depots domains.

To conclude, we have shown that NuSMV performs very well on small domain instances. However, as the domain size increases the state enumeration performed leads to NuSMV failing to construct the model and subsequently to synthesise any norms. While CRS is not as efficient an implementation the benefits of a localised search are clear. CRS is able to synthesise norms in all problem instance that NuSMV is, and in many problem instances where NuSMV fails. Additionally, the anytime nature of our algorithm allows for partial solutions to be drawn at any time during the process. Finally, should CRS be afforded more execution time then it would solve the more difficult problems too.

7.5.3.2 Norm Quality

In order to evaluate the quality of norms we compare the number of norms produced by each synthesis approach. Recall that we define higher quality norms to be abstract and generally applicable. Each of these more abstract norms govern interactions in numerous system states rather than each dictating behaviour specific to individual states. A direct result of more abstract norms is that fewer norms are required to bring about the same social objective.

We compare the number of norms produced by each approach, beginning with the average number of norms synthesised for the IPC domains in Figure 7.17. The number of norms is a function of the number of contributing actions, and hence the size of the conflict state specification. In all but the Depots domain the number of norms grows linearly with respect to the conflict specification size, since in all other domains an additional literal introduces only a single new contributing operator (and hence a single new norm), whereas in Depots more operators are often introduced. For example, a hoist is `available` if it `Loads` a crate onto a truck or `Drops` a crate onto a surface. The literal `available` is contributed to by both operators.

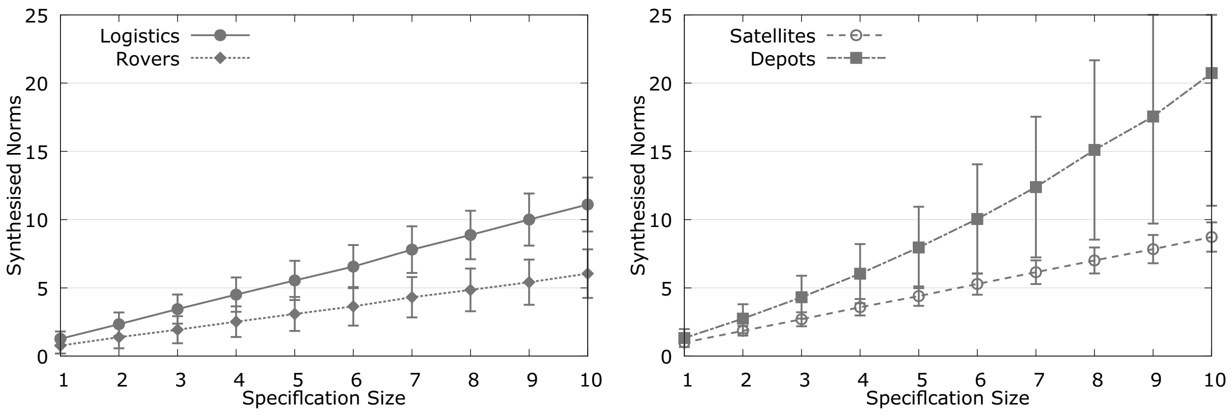


Figure 7.17: The increase in the number of synthesised norms for conflict specifications of increasing length, where the length is the number of literals in the specification.

Interestingly, even with this increase in the number of contributing operators the average norm count is slightly over 20 for conflict specifications of size 10 in the Depots domain, and even lower for the other IPC domains. We know from our traversal results that these norms correspond to thousands of complete conflict runs, but we also know that these small sets of norms prohibit every one of the complete runs.

Synthesising norms for arbitrary IPC domains is not possible without enumerating all reachable states. Previously, in Figure 7.14 we have already presented the norm

Grid Size	2x2	2x3	3x3	3x4	4x4	4x5	5x5
NuSMV	24	62	160	310	554	896	1937
CRS	2	2	2	2	2	2	2

Table 7.9: A comparison of the number of norms produced by the CRS and NuSMV approaches in the Parcel Delivery domain.

count for varying grids in the Parcel Delivery domain where agents are not permitted to collide. Here, NuSMV synthesises 6 norms in a 1x2 grid, but 1260 in a 1x20 grid. This growth is even further emphasised as we increase grid sizes in both dimensions. Table 7.9 presents these results.

The number of norms synthesised by NuSMV appears extreme. A closer inspection of the synthesised norms is required. Let the conflict specification be:

$$S_C = \{\texttt{agentAt}(\mathtt{a}_1,\mathtt{X}), \texttt{agentAt}(\mathtt{a}_2,\mathtt{X})\}.$$

In a 2x2 grid CRS synthesises the following generally applicable norms based on the two contributing actions, where $\mathtt{a}_1$ moves into conflict or where $\mathtt{a}_2$ moves in:

$$\big\langle\{\texttt{agentAt}(\mathtt{a}_1,\mathtt{X}'), \texttt{agentAt}(\mathtt{a}_2,\mathtt{X}), \texttt{conn}(\mathtt{X}',\mathtt{X})\},\ \ \texttt{move}(\mathtt{a}_1,\mathtt{X}',\mathtt{X})\big\rangle$$
$$\big\langle\{\texttt{agentAt}(\mathtt{a}_2,\mathtt{X}'), \texttt{agentAt}(\mathtt{a}_1,\mathtt{X}), \texttt{conn}(\mathtt{X}',\mathtt{X})\},\ \ \texttt{move}(\mathtt{a}_2,\mathtt{X}',\mathtt{X})\big\rangle.$$

NuSMV produces a master plan dictating exactly what each agent should do in each state of this system. For the 2x2 grid the sequence of actions is:

1.	move(agent2,node2,node4)	13.	move(agent2,node1,node3)
2.	move(agent2,node4,node3)	14.	move(agent2,node3,node4)
3.	move(agent2,node3,node4)	15.	move(agent2,node4,node3)
4.	move(agent1,node1,node3)	16.	move(agent1,node2,node4)
5.	move(agent2,node4,node2)	17.	move(agent2,node3,node1)
6.	move(agent2,node2,node1)	18.	move(agent2,node1,node2)
7.	move(agent2,node1,node2)	19.	move(agent2,node2,node1)
8.	move(agent2,node2,node4)	20.	move(agent2,node1,node3)
9.	move(agent2,node4,node2)	21.	move(agent1,node4,node2)
10.	move(agent1,node3,node4)	22.	move(agent2,node3,node4)
11.	move(agent2,node2,node1)	23.	move(agent1,node2,node1)
12.	move(agent1,node4,node2)	24.	move(agent2,node4,node2)

For each of these actions a norm is synthesised prohibiting all other action. The set of synthesised norms is very specific to the particular domain and problem instance, and the autonomy of agents is removed entirely. However there is an additional property with norms generated from this master plan: in order to select which operator

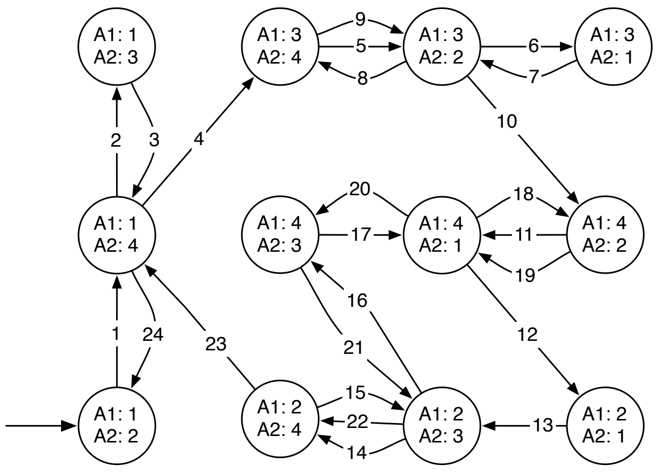


Figure 7.18: An automaton describing the normative behaviour of two agents (A1, A2) in a 2x2 grid.

to perform in a particular state the history of the system must be kept. Consider the automaton generated from the above master plan, illustrated in Figure 7.18.

The start state has Agent $\mathtt{a}_1$ at $\mathtt{node}_1$ and $\mathtt{a}_2$ at $\mathtt{node}_2$. Once the first action in the plan is performed the system transitions to the state where $\mathtt{a}_2$ is now at $\mathtt{node}_4$. We know the next action to be 2 since we have the knowledge that we have just performed action 1. However, without this knowledge it is unclear whether to perform action 2, 4, or 24. Not only are the norms generated by the model checking approach overly restrictive, but they require knowledge of the history of the system.

7.5.4 NuSMV Memory Analysis

So far we have only reported that NuSMV was not able to synthesise norms, but we have not mentioned the reason. In our tests when NuSMV failed to synthesise norms this was due to the test machine not having sufficient memory for the model checker to build the model.

Consider the square grid tests conducted using the Parcel Delivery domain, where agents were restricted to only moving. Here we are unable to retrieve results for the 5x5 Parcel Delivery Domain on our test machine due to the process exceeding the machine's available memory. Executing the model checker on a machine with additional resources did allow us to synthesise norms. In the 5x5 Parcel Delivery domain the model checker synthesised 1937 norms. Once the model was loaded the process occupied 6.291GB of memory. Yet for a 2x2 grid the model checker consumed a negligible amount of memory. The super-exponential time performance of the model checker can be directly correlated to the amount of memory it is consuming. Figure 7.19 illustrates how memory consumption grows as grid sizes are increased from 2x2 to 5x5, not in terms of process memory but rather in terms of the size of the model itself.

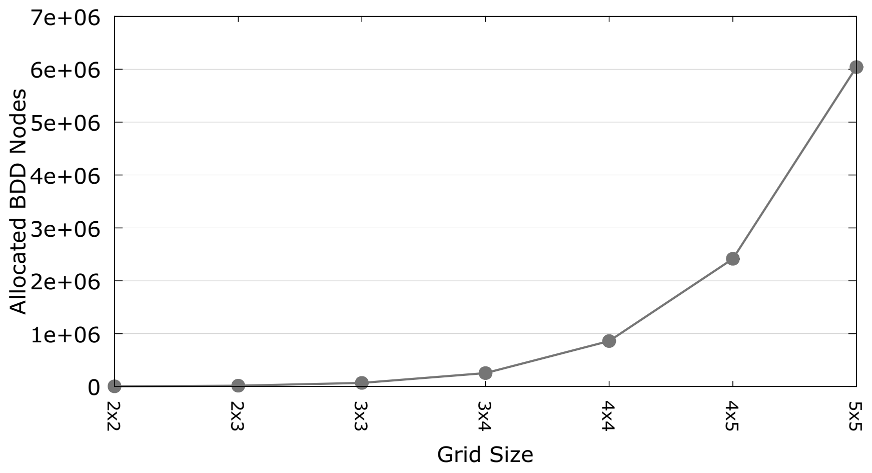


Figure 7.19: The increase in allocated nodes used to construct the NuSMV model with increasing Parcel Delivery domain sizes.

This behaviour is common in approaches that enumerate the state space. The trade off between scalability and efficiency is common too in automated planning. Rather than plan at the more complex abstract operator level many planners (including FF) enumerate the state space representation because planning with ground terms and operators is more efficient. This approach has its limitations as the size of the domain grows. The same limitations are present when using NuSMV to synthesise norms. Even though the checker itself is highly optimised and efficient, once the size of the domain scales beyond a manageable point model checking is no longer possible.

7.6 Conclusion

We claim that conflict-rooted synthesis is a more efficient approach to synthesising social norms that produces higher quality results. To validate these claims we compared CRS to NuSMV, finding that while NuSMV is superior on small problems, CRS is more efficient when problems increase in size. The anytime nature of our algorithm ensures that in situations where CRS exceeds resource limitations we are still able to place some guarantees for the norms synthesised. Furthermore, the norms synthesised are conditional on sets of system states, resulting in fewer norms being required to enforce the social objective. CRS is susceptible to scalability problems in systems with many operators, or with operators that contain many parameters, as illustrated by the poor performance in the Depots domain.

We presented a comprehensive analysis of the proposed optimisations, investigating the standalone and cumulative effects of each one. The benefits of the optimisations are clear, as they reduce the resources consumed during traversal, and subsequently reduce the number of reachability checks performed. The breadth-first nature of the traversal search rewards optimisations that reduce the number of runs early in the traversal: we showed that the Traversal and Independence optimisations were consistently more effective than the others, across all benchmark domains.

Chapter 8

Discussion

The discussion of conflict-rooted synthesis is separated into three sections. The first gathers the evidence presented, both theoretical and empirical, and discusses the findings of this work in the context of the thesis research statement to judge whether or not the thesis hypothesis holds. The second section details a set of extensions to this work that adapts conflict-rooted synthesis in order to apply it to new classes of problem domains. Finally, the third section discusses the significance of this work in the fields of Multiagent Systems, Automated Planning and Artificial Intelligence in general. Throughout this discussion chapter we address common criticisms explicitly, through individual analysis of each point in turn.

It is important to clarify the claims of this work. Firstly, a scalable approach to synthesising social norms is compulsory both for designers and for norm autonomous agents. We argue goal reachability must be incorporated into norm synthesis, but highlight situations where utilising focal state knowledge results in theoretical and practical issues. We introduce a new description of the synthesis problem that ensures goal reachability without any explicit goal knowledge, and we develop conflict-rooted synthesis as a solution to this problem which is superior to the model checking approach analysed in the evaluation.

8.1 Conflict-Rooted Synthesis

Synthesising norms in the absence of goal knowledge is a novel problem that differs from related problems in the literature. We show that our approach to solving this problem has a number of advantages over adopting a model checking approach. For the purposes of discussion we reiterate our hypothesis:

We can devise an algorithmic process that automates the synthesis of social norms given a declarative description of a planning domain and a specification of undesirable conflict states so that:

1. *the process is more efficient than state enumeration approaches, since a complete joint state enumeration is not always necessary,*
2. *the norms produced are of a higher quality as they are fewer, more abstract and generally applicable, and*
3. *the norms produced do not prevent agents from achieving their goals.*

We argue that conflict-rooted synthesis is a superior technique for synthesising social norms when compared to a model checking approach, as it is more computationally *efficient*, produces higher *quality* norms and can provides better *coverage* as it synthesises correct norms where a model checking approach fails. We reiterate our arguments for each of these points next, and provide possible counter arguments for discussion.

8.1.1 Efficiency of Abstract Localised Search

Conflict-rooted synthesis is more efficient than competing approaches based on propositional state groundings and enumerations for two reasons:

1. A *localised search* in the conflict state space means a conflict-free state space search is avoided.
2. Abstract operator schemata facilitate a more *abstract search*, avoiding a compulsory enumeration of individual states.

Chapter 7 showed that CRS scales more favourably than NuSMV, when evaluated against a set of challenging benchmark domains from the planning literature. In small domains NuSMV produces results more quickly than CRS, and consumes fewer resources.

▷ *Would enumerating the system states produce a simpler representation with which to synthesise norms more efficiently?*

Planning research has shown that grounding actions into propositional representations is an effective way of simplifying plan synthesis. This technique, commonly used by leading planners such as FF, simplifies the representation of the domain at the cost of an increase in representation size. There are three reasons why such an approach is not suitable for norm synthesis:

- NuSMV showed that the cost of enumerating the domain prior to synthesis is an inhibiting factor, even for small domains, resulting in the entire synthesis process failing during the grounding process.
- CRS performance is dependent on the number of runs generated during traversal. An abstract specification search produces fewer runs than a state search, enabling CRS to process larger domains than if grounded sequences were considered.
- Abstract search is not only beneficial from a computational perspective, but is key in generating abstract norms. An approach that grounds the domain prior to synthesis is likely to discard the knowledge needed to synthesise abstract norms.

Importantly, this discussion highlights that approaches used in plan synthesis may not be beneficial to conflict-rooted synthesis. Optimisations in planning commonly relax the problem domain in the hope of efficiently identifying a single plan, yet the same set of optimisations are not effective for conflict-rooted synthesis since traversal searches not for a single plan, but for every valid plan, equating norm synthesis to the worst case behaviour of plan synthesis: a plan enumeration.

▷ *Traversal is independent of the conflict-free state space, but reachability analysis is not?*

Each planning invocation during reachability analysis searches for an alternative in the conflict-free state space requiring resources proportional in size. We present two points for discussion:

- While reachability is checked through conflict-free state search, no plans are required to search the entire state space. NuSMV illustrates that searching for a master plan through every conflict-free state is infeasible, while CRS shows that searching for many, simpler plans between conflict-free states is more efficient.
- This search is required when problem-specific knowledge is required to check reachability. When the Planning with Variables optimisation succeeds this search is avoided, since sufficient knowledge is contained in the traversal run to synthesise an alternative plan.

8.1.2 Generally Applicable Social Norms

Conflict-rooted synthesis produces norms of higher quality, based on three key properties:

1. Norms are *generally applicable* since they are conditional not on individual system states, but rather on sets of states, thereby producing fewer norms.
2. Norms are more expressive, using *variables* to produce fewer succinct norms.
3. *Knowledge* of the history of the system, or the ability to perceive other agent action, is not required.

Additionally, unlike norms produced by NuSMV, CRS norms govern only the space from which conflict may arise, avoiding the need for a master plan that governs behaviour in conflict-free states. Finally, a severe limitation of a model checking approach to norms is the sheer quantity of norms: in many examples, NuSMV synthesises more norms than system states.

When drawing comparisons to more general state enumeration approaches many of the same arguments hold. Consider a naive approach that searches all system states in order to identify and prohibit transitions that lead to conflict states. Here, no master plan is produced, but the number of norms is still conditional on the number of states in the enumerated system. We again analyse some points for discussion.

▷ *How do agents incorporate abstract norms into their means-end reasoning? Are abstract norms more complex?*

CRS produced norms containing variables that are bound to the agent's current state at runtime in order to identify whether the norm is applicable. While binding at runtime is more complex, there are additional factors to consider:

- There are fewer abstract norms, thereby saving computation since fewer norms are checked.
- Agents can incorporate abstract norms into their practical reasoners through the use of control knowledge, allowing for efficient norm-compliant plan synthesis.
- Abstract norms can be grounded if required to produce a variable-free representation. Importantly, the resulting norms are ground, but still conditional on sets of states rather than individual system states.

In short, conflict-rooted synthesis produces succinct norms that can be enumerated into state-specific norms: a simpler process than inferring a more general, abstract representation from a more specific one.

▷ *If CRS does not complete, what partial guarantees does its anytime nature provide?*

Let the resources available to CRS be bound so that it is not able to complete synthesising norms. The anytime nature of the algorithm provides guarantees for sequences of actions up to the length of the runs investigated during traversal. Should agents in the normative system create plans longer than this limit, then there is no guarantee that goal reachability holds. A system designer can enforce goal reachable behaviour by limiting the plan lengths, or by forcing agents to re-plan once the limit is exceeded.

▷ *Would generating a master plan and accordingly regimenting agent behaviour produce more efficient systems?*

Domains can be found where regimenting agent behaviour is beneficial, both for norm-compliance but also for system efficiency since agents are required to reason less about their actions. For these domains a model checker is the ideal tool for synthesising such plans, yet in practice it is unlikely that a designer would wish to reduce the autonomy of agents in a system. Regimented systems are not necessarily more efficient since there is no guarantee that the resulting master plan would be efficient, from an agent's perspective to achieve its goals, or from a systems perspective. In Section 7.5.3.2 we highlighted this empirically in a 2x2 Parcel Delivery world where the master plan composed of 24 actions was inefficient for agents, primarily since it ensured reachability of *joint* states, but also since the resulting plan visited these states multiple times. Once Agent $\texttt{a}_1$ left $\texttt{node}_1$ after action 4 of the sequence, it only returned 13 actions later. This is less efficient than allowing autonomous behaviour so that $\texttt{a}_1$ can move back to $\texttt{node}_1$ immediately, or even by forcing agents to continually move in a single direction.

▷ *Can the synthesised norms be used in enforcement-based systems?*

Conflict-rooted synthesis allows a system designer to synthesise norms for goal autonomous agents as the produced norms still allow agents to achieve their goals, no matter what these goals are. The designer can regiment the system with these new rules, and be guaranteed that goal achievement is preserved. However, these rules provide no guarantees over the efficiency of the system as it is possible that goals are achieved at a higher cost in the normative system. In systems where there is no cost associated with a particular course of action, no additional incentives are required for agents to adopt and comply with the norms, as agents still achieve their goals through norm compliance. However, in systems where agents choose to deviate from the norms then additional norm enforcement mechanisms are required to ensure that agents are incentivised to comply with the norms.

Norm synthesis is a core function of norm autonomous agents, where agents are required to synthesise and evaluate norms at runtime. Our approach is a viable tool that allows norm autonomous agents to synthesise these norms.

8.1.3 Improved Norm Coverage

Conflict-rooted synthesis offers improved norm coverage, since it synthesises norms in domains where a model checking approach fails independent of any computational limitations. In Section 7.5.1 we detail a theoretical comparison of the two approaches, and show them to differ in how they ensure goal reachability. Conflict-rooted synthesis ensures that every goal that was previously achievable in the system is still achievable in the normative system while the model checker ensures that every focal state is reachable an infinite number of times. Terminal focal states cannot be visited infinitely often, and it is therefore not possible to encode goal reachability of these states into the checker's model. We discuss some of the implications of this next.

This is best illustrated in the Parcel Delivery domain. Consider that when an agent drops a parcel the system consumes the parcel, thereby altering the number of parcels in the domain. The goal state to have a parcel delivered is key to allowing agents to achieve their goals. When considering norms that prohibit agents from colliding we must consider whether or not these norms allow agents to deliver parcels. The model checking approach does not facilitate this: once a particular parcel is dropped the states where the parcel existed are no longer achievable. Any master plan that attempted to bring them about would fail.

▷ *Should social norms be problem-specific?*

Consider the case where the operator specifications remain constant, but the problem specification of the domain changes at runtime. A master plan is unable to cater for these dynamic systems since the plan itself requires updating to ensure reachability of any altered conflict-free states. For example, suppose that the topology of the Parcel Delivery world changes at runtime. Conflict-rooted synthesis norms ensure goal reachability and it ensures continued system function. Regimenting agent action is therefore detrimental to agent autonomy, and subsequently to the performance of the system itself, and is an inherent problem with a model checking approach to norm synthesis.

▷ *Could social norms be re-synthesised at runtime?*

It is possible to re-synthesise norms at runtime when the system changes but this may prove to be expensive and is contrary to the core objective of social norms: they are long term and persistent. An approach that does not require this re-synthesis is superior to one that does, since agents are able to complete their individual plans safe in the knowledge that the social objective is satisfied.

8.1.4 Limitations of Our Approach

We now detail the limitations of this work and the assumptions made in its development, and comment on the impact on the applicability of our approach in practice. We address some common concerns, such as the incorporation of conditional operator effects, through extensions in Section 8.2.

8.1.4.1 Limited Conflict Specification Expressivity

The language used to specify conflict state specifications is derived directly from the adopted planning formalism, with additional support for universally quantified variables. This conflict state representation was inspired by the work on Social Laws by Shoham and Tennenholtz (1995) and further motivated by the fact that, even for these simple specifications, no efficient process exists to synthesise norms. This representation is limited in two ways:

1. Conflict specifications are state-specific and do not include any action information, thereby excluding specifications of conflict based on sequences of agent action.

2. There are no temporal or path operators allowing for specifications of conflict based on future states of the system.

The fact that conflict-rooted synthesis does not allow for the specification of conflict using action information makes sense if there are no temporal relations since if a designer wishes to prohibit a particular action then they can do so simply by altering the operator schema, or by creating a trivial norm that unconditionally prohibits the action. Since actions are asynchronous in our model it is never the case that concurrent action can be specified as leading to conflict.

With temporal operators, action-based specifications of conflict are more viable, since it is now possible to specify a sequence of actions as undesirable. In the Parcel Delivery domain this may equate to prohibiting agents from picking parcels up and subsequently dropping them immediately. Currently we have no mechanism to represent this undesirable behaviour.

8.1.4.2 Dependence on Planning-Based Operators

Our approach requires planning-based operator schemata, with abstract search not possible in an enumerated system. We adopted planning-based formalisms since they are well known, flexible in expressivity, and general in that an entire class of domains can be specified. As a result, our approach is applicable in any domain that can be expressed through the adopted formalism, and the resulting norms are independent of any planning instance.

Abstract operator representations are central to our approach, yet planning formalisms are not the only source of languages. Plan libraries in AgentSpeak (Rao, 1996) contain similar action abstractions, allowing us to define acceptable agent behaviour succinctly. A mapping from any custom action specification to a planning-based one would allow our approach to be applied as is.

8.1.4.3 Expressiveness in the Planning Formalism

The planning formalisms presented in this work lack many of the expressivity features of current planning languages. For example, they do not include basic planning notions such as conditional effects, nor do they consider action costs, numeric fluents or non-deterministic action outcomes.

8.1.4.4 Support for Obligatory, Permissive Norms

This work has primarily focused on the synthesis of prohibitionary social norms, yet it is simple to extend this norm synthesis approach to produce obligatory norms through a rewrite procedure: an agent is obliged to perform a particular action if they are forbidden to perform any other action. We detail an approach to synthesising obligations in Section 8.2.5. There are limitations to the obligations produced by this method:

- Obligations are *single step* where agents in a precursor state are obliged to make a transition to an obligatory state. These obligatory norms cannot dictate agent behaviour over a sequence of actions, or plans. This is an effect of the limited expressivity of the conflict state specifications in this work: with temporal operators it would be possible to specify desirable sequences of states, and to produce multiple action obligations.

- Obligations are not motivations that are incorporated into the agent's deliberative process. Agents deliberate, select goals and perform means-end reasoning to achieve these goals. Typically, norm autonomous agents select these goals based on their current set of activated norms. The problem of normative deliberation is not one we approach here, but rather focus on the effects norms have on means-end reasoning. That is, given a goal, how do planning agents synthesise norm-compliant sequences of actions to achieve these goals.

There are many additional norm-related concepts that we have not discussed. Prohibitions and obligations are commonly grouped with *permissions*. Permissions act as exceptions to prohibited action with semantics typically defined by the institution or normative system. Kollingbaum and Norman (2003) define permissions as superior to prohibitions in the NoA architecture, allowing general prohibitionary norms to be overridden with more specific permissive ones. For our purposes permissions are not utilised since we are only interested in a static set of undesirable system states. This is particularly appealing, since it avoids assuming a set semantics for the interactions between obligatory, prohibitory and permissive norms.

Power in normative systems indicates how pivotal an agent is with respect to the ability of the agents to achieve the social objective. Agents critical to the objective are more powerful than those that are not. In our system all agents referenced, directly or through a variable binding, by the set of produced social norms are required to achieve the social objective. As such, the norms we generate distribute power equally among these agents. Furthermore, Ågotnes et al. (2009) defined a set of social laws as being *minimal* if no subset of the norms achieves the social objective. In this work, the set of social norms produced are always minimal in the sense that if any norm is removed the social objective is not guaranteed to hold.

8.1.4.5 Violation of Synthesised Norms

This thesis does not describe how system designers can implement their normative systems to ensure that agents abide by the synthesised norms. Reachability analysis assumes compliance, yet in practice agents may choose to deviate from the suggested behaviour. We intentionally make little distinction between regimented and enforced norm implementation mechanisms in our work as we believe our approach to social norm synthesis to be applicable in both paradigms, yet it is important to acknowledge that our evaluation of conflict-rooted synthesis assumes that agents comply with the norms. While we derive significant inspiration from work done in artificial social systems we also note that our approach caters more generally towards norms found in normative multiagent system research, but emphasise that for a complete normative systems implementation the synthesised norms must be coupled with an appropriate norm implementation strategy.

8.2 Conflict-Rooted Synthesis Extensions

We discuss a set of extensions to the conflict-rooted synthesis algorithm, which differ from optimisations as they aim to alter the class of problems that the synthesis approach can solve.

8.2.1 Extension 1: Incorporating Initial and Goal Knowledge

Synthesising norms using agent operator schemata produces prohibitions that are common for different problem instances of the domain, yet the guarantees produced during reachability analysis may not hold for all these problem instances as ensuring a conflict-free path often requires problem-specific knowledge. In fact, it is possible that a large subset of the runs produced during traversal are not reachable from a specific initial state in a given problem instance. Similarly, when guaranteeing reachability we may not wish to ensure global goal state reachability, but rather reachability to a defined set of goal states. We now present an extension of our work for discussion that incorporates initial and goal state knowledge into the traversal process so that each of the runs produced during traversal are reachable from the initial states, and subsequently that alternative plans exist for each of these runs to the goal states.

Example Consider the specification $\{\texttt{agentAt}(\texttt{a}_1,\texttt{node}_1),\texttt{agentAt}(\texttt{a}_1,\texttt{node}_2)\}$ in the Parcel Delivery domain where $\texttt{a}_1$ is in two different locations. It makes little sense for an agent to occupy two locations, yet there is nothing in the operator schemata alone that forbids this. Instead, the initial state dictates that agents begin in a single location, and since the occupation of a single location is preserved during operator application it follows that agents cannot occupy more than a single location at one time.

Without knowledge of the initial state, traversal must consider states where agents are able to occupy an arbitrary numbers of states at a time.. □

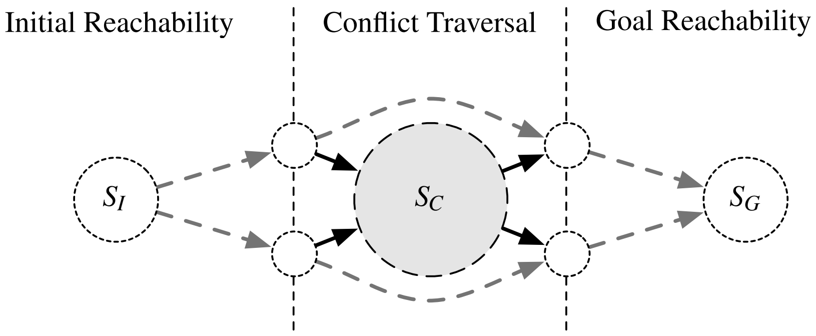


Figure 8.1: Incorporating Initial and Goal State Knowledge

Our approach to incorporating knowledge into conflict-rooted synthesis is illustrated in Figure 8.1, with an initial reachability step to the left, and goal reachability step to the right. The specification S_I represents the initial specification, while S_G the goal specification. For each run produced during the traversal process, reachability is checked from S_I to the first specification of the run, and from the last specification of the run to S_G. If no path exists to the goal state specification then the run is discarded since it is not reachable in the unrestricted system.

This extension is incorporated into reachability analysis since the goal and initial state reachability can only be checked once the runs have been grounded. For each grounded run, the planner is invoked not only to check conflict reachability, but initial and goal reachability as well.

8.2.2 Extension 2: Invariant Constraints

Initial and goal state knowledge is incorporated into the reachability analysis step of conflict-rooted synthesis where runs are removed from consideration once they have been identified during traversal. A second extension of our approach allows us to easily incorporate additional domain-specific knowledge into the traversal process.

A simple means of incorporating domain knowledge is the use of invariant constraints. For our purposes we define an invariant constraint as a tuple $\langle S, \kappa \rangle$ composed of a unground state specification S and set of constraints κ over the variables in S. The expressivity of S and κ follow from the planning formalisms as before.

Example To prohibit a single agent from occupying two locations at the same time we can introduce the following invariant constraints into the Parcel Delivery world:

$$\Big\langle \{\texttt{agentAt(A,X)}, \texttt{agentAt(A,Y)}\}, \quad (\texttt{X} = \texttt{Y}) \Big\rangle \qquad \square$$

Invariant constraints are applied each time a new specification is created during traversal. Let S_n be a specification created during inference or refinement in traversal and $I = \langle S, \kappa \rangle$ be an invariant constraint. We can verify the invariant constraint with S_n as follows:

1. Identify every substitution binding ς where $S_n \models \varsigma[S]$.
2. For each ς, incorporate the constraints $\varsigma[\kappa]$ into the run containing S_n.
3. If the modified constraints are no longer consistent, then the run can be discarded.

Alternatively, the incorporation of invariant constraints can be introduced after runs are grounded during reachability analysis. Here a ground run is discarded under S_I if any of its specifications can be bound to S, and if the binding results in κ becoming inconsistent.

8.2.3 Extension 3: Conditional Operator Effects

Our final extension addresses the lack of support for conditional operator effects. We detail this extension using the classical planning formalism. Conditional effects are conditional operator effects of the form $\Gamma \Rightarrow E$, where Γ are effect conditions and E the effects, where the effects E are only applied if the conditions Γ hold in the current specification. There are two approaches to incorporating conditional effects:

1. **Algorithm Adaptation**: Alter the conflict traversal algorithm to handle the conditional effects.
2. **Operator Compilation**: Compile conditional effects away using an appropriate technique (such as that presented by Gazen and Knoblock (1997)) to produce a larger set of operators but with no conditional effects. This new set of operators can be used to perform the traversal and subsequent reachability analysis.

We begin by briefly detailing the changes required to the conflict-traversal process to incorporate conditional effects, before discussing which is favourable in practice:

- **Contributing Operators**: Conditional effects bring a change to how contributing operators are identified during traversal. Here, an operator o contributes to

S_C if any of the conditional effects can bring about a literal in S_C. Furthermore, inferring the precursor specification now involves not only the preconditions of the operator ($pre(o)$) but also the preconditions of the conditional effects that bring about S.

- **Successor Specifications**: Assume a specification S and an operator o that is partially applicable in S. Traversal typically refines S to ensure that o is fully applicable. With conditional effects this refinement process does not result in a single refined specification, but rather a set of possible specifications, where each specification satisfies all the preconditions of o as well as some subset of the conditional effects. To ensure that all possible specifications are produced every combination of conditional effects must be modelled.

Even though it is possible to incorporate conditional effects directly into the conflict traversal process, it remains uncertain whether it is advantageous to do so. Many of the steps required to incorporate conditional effects into conflict traversal are identical to those performed during operator compilation, except that this added complexity is introduced at every step of the traversal process. Furthermore, while operator compilation results in an increase in the number of operators, it does not affect the number of resulting runs produced during conflict traversal, and there is therefore no representational benefit in preserving conditional effects during traversal.

8.2.4 Extension 4: Asynchronous Action

A key distinction between our approach and competing approaches is our assumption that agent actions are asynchronous. We consider social norms to be prohibitions on local, agent-specific behaviour, yet prohibiting single agent action in a system with asynchronous action results in overly restrictive norms. For example, let $S_C = \{\texttt{at}(\texttt{a}_1,\texttt{node}_1),\texttt{at}(\texttt{a}_2,\texttt{node}_{1)}\}$ in the Parcel Delivery domain. We depict the set of three contributing *joint* actions in Figure 8.2. Each contributing action's edge is labelled with the relevant norm synthesised to prohibit access to the conflict state, where we write "..." to signify that an agent can perform any action except one that alters its current location. We prohibit $\texttt{a}_1$ and $\texttt{a}_2$ from moving into $\texttt{node}_1$ should another agent occupy $\texttt{node}_1$. Now consider the joint actions that are prohibited according to these norms, as depicted in Figure 8.2. By restricting joint action based on prohibited individual action, our norms prohibit joint actions that do not lead to conflict. For example, consider the joint action where $\texttt{a}_1$ moves into conflict, but $\texttt{a}_2$ moves away.

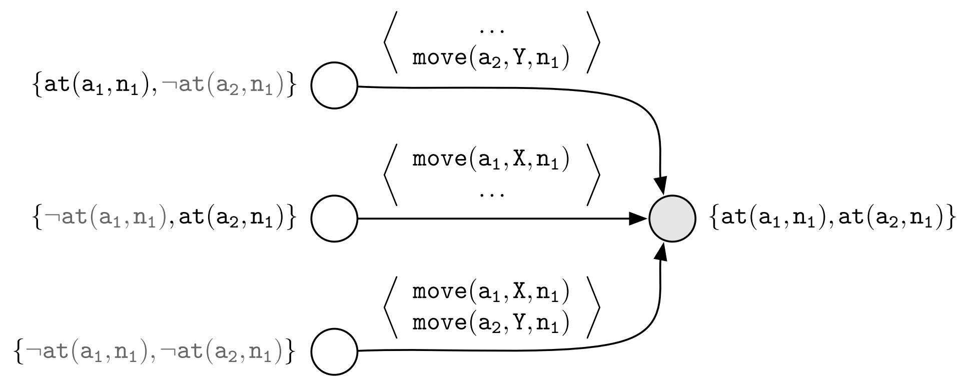
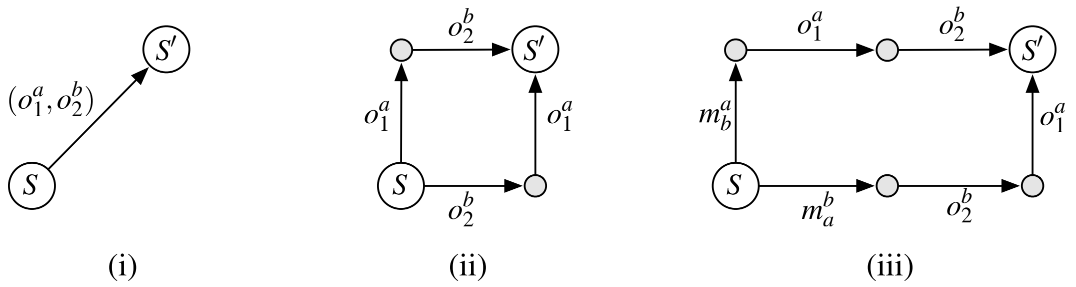


Figure 8.2: The three joint actions contributing to conflict in the Parcel Delivery domain.

The resulting state after this action is conflict-free, yet the joint action is prohibited. As a result, single agent prohibitions are overly restrictive when synchronous actions are considered. Assuming asynchronous action allows us to synthesise more accurate action prohibitions that can be applied by agents based solely on the current state of the system, independent of the actions of other agents.

To avoid these shortcomings we serialise synchronous agent systems so that our approach can be applied directly. Figure 8.3 illustrates one way of transforming a synchronous state transition to an asynchronous transition. The transition from $(i) \rightarrow (ii)$ applies in systems where the actual action execution is not synchronous while $(i) \rightarrow (iii)$ is applied when actions are guaranteed to execute synchronously. In the Figure, o_i^j represents agent j performing operator i and m_i^j represents a message operation.



$$\begin{tikzcd} & S' \\ S \arrow[ur, "{(o_1^a, o_2^b)}"] & \end{tikzcd} \qquad \begin{tikzcd} \bullet \arrow[r, "o_2^b"] & S' \\ S \arrow[u, "o_1^a"] \arrow[r, "o_2^b"'] & \bullet \arrow[u, "o_1^a"'] \end{tikzcd} \qquad \begin{tikzcd} \bullet \arrow[r, "o_1^a"] & \bullet \arrow[r, "o_2^b"] & S' \\ S \arrow[u, "m_b^a"] \arrow[r, "m_a^b"'] & \bullet \arrow[r, "o_2^b"'] & \bullet \arrow[u, "o_1^a"'] \end{tikzcd}$$

Figure 8.3: Modelling joint action in an asynchronous action system.

In systems with global state transitions over joint actions, where the underlying operation execution is serialised, the norm synthesis procedure can be applied on a transformed system where joint operators are replaced by serialised actions with the introduction of intermediate states. In Figure 8.3 (ii), the joint operator (o_1^a, o_2^b) transitioning from S to S' is replaced by two paths, the first where agent a performed operator o_1 before agent b performs o_2, and vice versa. The intermediate states are introduced to represent that the first operator has been performed after state S. In Figure 8.3 (iii), where actions are guaranteed to be performed synchronously, the introduction of intermediate states does not suffice. In this situation, some communication is required for agents to ensure that the combination of their actions will not violate the introduced norms. The prescribed normative behaviour is for an agent to communicate their operator preference prior to execution. We assume some mechanism, such as an ordering of agent IDs, for the resolution of communication ties. The norm is to notify other agents in states that might lead to conflict, to ensure that the joint operator executed subsequently by all agents ensures the conflict state is avoided.

8.2.5 Extension 5: Synthesising Single Step Obligations

An agent is obliged to perform a particular action if they are forbidden to perform any other action. While both prohibitions and obligations restrict the system it is obligations that are more restrictive since they prohibit all behaviour that does not lead to desirable system states. We define an obligation as an identical tuple to prohibitions:

$$\langle \varphi, o \rangle$$

where φ is a specification modelling the states in which this norm applies, and o is the obligatory operator to be performed. Obligatory norms can be synthesised by our approach using the following steps:

1. Let S_O be the specification of obligatory states.
2. Traversal identifies the contributing operators that lead to S_O, and infers the set of possible successor states from S_O, creating a set of runs of length 3.
3. For each run, and obligation is synthesised to perform the contributing operator conditional on the first specification in the run.
4. To check reachability each obligation is rewritten as a prohibition, with identical conditions but where all actions other than the obligatory one are prohibited. From here, reachability analysis continues as before.

Importantly, the obligations synthesised are single step obligations, in that they only apply in the precursor specification. There is no mechanism to ensure that obligations are satisfied at a future point in the system.

8.3 Significance and Impact

We now take a more high-level perspective in analysing the possible contributions of this work in the fields of multiagent systems and automated planning. We separate our discussions on the significance and impact of our work along these lines and additionally present comments on more general contributions to the field of Artificial Intelligence.

8.3.1 Contributions to Multiagent Systems

The use of institutional models to structure and regulate agent-based systems has introduced a number of key concepts into multiagent systems research. By using social norms as the mechanism to define roles within an institution designers are able to express an expectation of agent behaviour that can bring about predictability in the system. Systems comprised of norm autonomous agents have been proposed that are self-regulated where agents reason about proposed norms, choose whether to adopt norms, and decide whether to violate adopted norms. Most of this work is focused at the system design level (norm negotiation, norm specification languages, sanction mechanisms . . .), or at the agent level on norm governed practical reasoning, yet very little research has been conducted into how these social norms come about even though the ability to synthesise norms is a fundamental capability of a norm autonomous agent. If agents are not able to synthesise norms, then these self-regulated multiagent systems are not possible, regardless of the level of expressivity of the norms in the system or the completeness of negotiation dialogues.

Synthesis is important for another reason. The problem of how agents choose whether to adopt proposed norms, and how agents choose to create new norms are inherently related. The key question in both of these is how an agent quantifies the effects that a norm has on its ability to achieve its goals. If an agent is able to list the goals it can no longer achieve, it can weigh this loss in utility against any penalties for non-collaboration. Our approach to norm synthesis takes this quantification into account when synthesising norms by performing an analysis of the effects of the synthesised norms. Our contributions are therefore not only related to the synthesis of social norms but also to the adoption of social norms.

We need not limit the contributions of this work to online norm proposal and adoption, for even as a system designer's tool there is utility in being able to guarantee the effects of system modifications. Just as designers utilise planners to synthesise plans contained in a plan library, so they require a tool to synthesise norms in an offline manner. There is no requirement for the designer to manually create these norms even if the agents are not able to synthesise norms themselves, yet without an automated approach to norm synthesis even the offline design of norms becomes a complicated task for system designers.

We split our work into three contributions to the multiagent systems community:

1. **Norm Synthesis**: Conflict-rooted synthesis is a scalable, practical approach to online and offline automated synthesis and reachability analysis in planning environments.
2. **Norm Adoption**: An anytime approach to reasoning about proposed norms based on the same reachability analysis procedure.
3. **CRS**: An implementation of the conflict-rooted synthesis algorithm and associated optimisations.

Additionally, one might consider the formalisation of the problem of norm synthesis with no explicit goal information as an additional contribution. We have shown that practical synthesis using focal states is not possible without explicit knowledge of what states should be reachable in the normative system. Our approach at attempting to ensure all states are reachable is novel, and potentially an avenue to be further investigated. By using standard benchmark planning problems we also ensure that any future work can empirically be compared to our approach, allowing for more precise comparisons to be drawn. To summarise, we believe our work to be the first viable norm synthesis approach able to synthesise concise, generally applicable norms. The ability to synthesise norms is a key capability of norm autonomous agents, and essential for agents in agent-mediated, or self-regulated, electronic institutions. Our main contribution is a vital component of autonomous agents.

8.3.2 Potential Contributions to Automated Planning

The contributions of this work to automated planning are more speculative. The motivating problem behind the development of our approach is norm synthesis, however there are certainly aspects of our approach that interest the automated planning community. We emphasise that these contributions are purely for the purposes of discussion, and have not been formally presented to the community.

8.3.2.1 Applicability to Plan Synthesis

As presented previously, the plan existence problem differs from our approach to reachability analysis: instead of searching for a single plan from initial to goal specifications, we search for any plans that traverse through some state specification. If we analyse the approaches more closely it seems reasonable to state that we are interested in enumerating plans rather than searching for a single plan. Even if we were to adapt our approach accordingly there is another key difference, our approach utilises refinement to identify plans through *any* state represented by the conflict specification, rather than identifying a single plan that can be applied to *all* states in the initial specification. That is, given an initial and goal specification, conflict-rooted synthesis could be adapted to search for a plan, yet would produce one that traverses from some subset of represented initial states to the goal states.

Conflict-rooted synthesis is different to sequential planning, yet there appears scope for using our approach in situations where a single plan does not exist and a sequential planner fails. Refinement allows us to search for plans originating from subsets of the initial state specification. A solution to this planning problem is a set of plans, each traversing from different (possibly overlapping) subsets of the initial state specification. For example, consider the classical planning problem where we wish to locate a plan that traverses from initial specification S_I to goal specification S_G. It could be that no single sequential plan exists, and sequential planning fails. By placing restrictions on S_I we could construct *different* plans that are independently able to achieve S_G, and through their combination are applicable in all states represented by S_I. Consider that a literal l might be added to S_I during refinement, resulting in a more specific state specification $S_I \cup \{l\}$. Now, we additionally require a plan from the subset of S_I that has been discarded, $S_I \cup \{\neg l\}$. One might consider this a form of conformant planning with deterministic action outcomes where uncertainty exists in the initial specification. The outcome of applying our approach would not be a single plan, but rather a set of plans, each traversing from some subset of S_I. This is potentially beneficial in situations where a single sequential plan does not exist.

8.3.2.2 Applicability to Generalised Planning

Our work is similar to approaches in the sub-field of generalised planning where, rather than synthesise a single sequential set of actions, we wish to create a general plan that, given a problem, can be instantiated into a sequence of problem-specific actions. One key subfield of generalised planning is the synthesis of plans with loops. Intuitively, if we create runs with the possibility of repetitive operators, then we are creating runs with loops. Consider a single instance of such a run containing repetitive operators. By unfolding the repetitive operators we produce an infinite number of sequential plans, each of which solves a different planning problem. We discuss this approach to generalised planning according to the following desirable criteria presented by Srivastava et al. (2009):

1. **Applicability Test**: It is desirable for a generalised plan to provide an efficient test of *applicability*: whether or not the generalised plan can be applied in a given problem instance. In the context of our approach this equates to identifying whether a given run solves the planning problem, which can easily be done by checking the initial and goal specifications. If the run's specifications model the planning problem's, then a solution exists where repetitive literals in the run can be mapped to any number of grounded instances.

2. **Quality of Instantiations**: Generalised plans should produce sound plan instantiations. Since traversal is based on a breadth first search of the search space we are guaranteed to not only find a plan, but also to find the shortest one.

In the context of generalised planning it appears that conflict-rooted synthesis has much to offer yet further work is required to quantify its significance.

By searching the state space in an ungrounded fashion our approach avoids a compulsory state enumeration, at the cost of added complexity involved in the continual binding of variable symbols. This tradeoff is a common theme in the planning community, with planners (such as FF) that ground operators prior to planning proving very effective in practice. This characteristic is also evident in our empirical evaluation, where on smaller domains NuSMV requires fewer resources than CRS. As such, we do not expect an adaptation of our work to automated planning to be superior to existing planning approaches, yet suggest that it may be applicable to problems where a state enumeration is not feasible, or in situations where the abstract nature of operator schemata should be preserved in the resulting plans.

8.3.3 Potential Contributions to Artificial Intelligence

Conflict-rooted synthesis, automated planning and model checking can all broadly be classed as approaches to searching a state space. It seems sensible to therefore discuss the contributions that conflict-rooted synthesis may have on the Artificial Intelligence community at large through the characterisation of the norms synthesis problem purely in terms of search. Search problems in which an approach similar to conflict-rooted synthesis may be suitable should contain the following core properties:

1. **Abstract Specifications**: Systems defined at a high level using relational representations that have clear operational semantics. Norm synthesis utilises this through the use of state and operator abstractions.
2. **General Results**: Problems where the result of the search process should preserve the abstract nature of the system, lowering the level of generality only when required. Our approach is applicable to problems where the abstract nature of the underlying system specification is key to the results produced. Discarding this generality through an initial grounding is not desired.
3. **Dynamic Systems**: Systems that are iteratively modified, where guarantees must be placed over potential modifications are well suited to our approach.
4. **Unground Search**: Systems that are more efficiently searched at an abstract level, where enumerating and building a grounded model is simply not feasible.

Broadly speaking, many approaches in the field of Artificial Intelligence are unable to search abstract state spaces, choosing rather to ground representations for efficiency purposes. Conflict-rooted synthesis shows that for a specific Artificial Intelligence problem there are scalability benefits to searching at this more abstract level, even though the abstract representation is more complex in principle.

8.4 Summary

Our discussion is separated into three parts. First, we discuss different criticisms of our approach, particularly concerned with the validation of our research statement. Conflict-rooted synthesis has limitations, particularly in complex domains and in domains that require higher levels of expressivity or conflict representations. In these more complex domains our approach may not suffice, yet in domains where these assumptions are acceptable our approach is a viable means of synthesising norms. By assuming simpler representations we are able to present an approach that encompasses all domains within these restrictions, and are able to theoretically show it to be sound. This initial contribution has significant scope for future extensions and refinement, both in the core algorithm, optimisation extensions, and resulting implementation.

This research presents an approach to solve the fundamental problem of norm synthesis. As multiagent systems models more complex institutions it is necessary that agents be able to reason about norms, and to propose new norms. Our solution is primarily aimed at multiagent systems, yet contributions exist more generally general in the field of automated planning, although future work is required to quantify any potential contribution.

Chapter 9

Conclusion

This research focused on the problem of norm synthesis in planning-based environments. We demonstrated that the problems of reachability analysis and norm synthesis are fundamentally related and that any approach to norm synthesis must provide some goal reachability guarantees. Existing approaches to norm synthesis are able to create norms that ensure reachability when additional goal knowledge is provided, yet in practice it is often infeasible, or even impossible to list focal states in this manner. Conflict-rooted synthesis avoids a focal state enumeration by assuming that all conflict-free states are focal, and synthesises norms that guarantee access to all previously reachable states.

Conflict-rooted synthesis is not only a more efficient means of synthesising norms, but also produces norms of higher quality by utilising high level, abstract agent action specifications to produce fewer abstract norms. These norms are independent on individual states and instead govern entire sets of states, yet have clear operational semantics and can easily be instantiated into state-specific representations.

Although norm synthesis is not a new problem, the lack of approaches that preserve operator and state abstractions was a primary motivation behind this work. Approaches based on compulsory state enumerations that require a priori grounding of agent actions are counterproductive when abstract norms are required. The number of norms synthesised by the model checking approach for even the simplest domains is evidence that approaches based on state grounding are less effective at solving this problem of norm synthesis. Furthermore, the numerous norms synthesised results in draconian restrictions on agent behaviour thereby severely limiting agent autonomy, and is not practical in systems where agents are independently implemented.

Perhaps the most significant motivation for conflict-rooted synthesis is provided by the work on norm autonomous agents. For agents to be autonomous of the norms they adhere to, they must have an explicit notion of these norms, must be able to propose new norms and must assess proposed norms for adoption. With conflict-rooted synthesis, designers now have the necessary tooling with which to equip their agents to reason about norms. The anytime nature of our approach allows agents to make preliminary decisions regarding norms even in situations with very limited resources. The ability for agents to reason about norms is a fundamental capability of norm autonomous agents, and is essential tooling for institution-based agents.

From a designer's view, our work can be incorporated into agent machinery to facilitate this norm autonomy, or designers can use it to regiment systems of goal autonomous agents. Our approach allows a system designer to synthesise norms that are entirely independent of the goals of the agents within the system, allowing for system-wide restrictions and guarantees no matter how agents deliberate.

9.1 Thesis Summary

An approach to norm synthesis that avoids a compulsory state enumeration must adopt abstractions over the underlying system. Automated planning provides us with these abstraction mechanisms, and it follows that the background and related work for the fields of multiagent systems and automated planning are provided. In Chapter 2 we detailed key background in multiagent systems, beginning with an overview of coordination techniques in order to motivate social norms as a form of pre-planning coordination. The key properties of social norms were outlined: explicitness, persistence, generality and compliance incentives. We followed this with relevant social norm representations from the literature, identifying a number of representations that are typically overly formal and lack operational semantics, or state-specific and lack generality. Broadly we classified approaches to synthesising norms as emergent, online, and offline approaches, and discussed each in turn, but chose to focus on the social law model which acts as inspiration for our subsequent research. We outlined the computational complexity of synthesising social norms in this framework but noted that no algorithmic means is provided to synthesise them. We concluded the chapter by presenting the details of the most credible approach to norm synthesis, based on ATL model checking.

Our approach relies heavily on automated planning representations. In Chapter 3 we detailed two such formalisms: a propositional set theoretic approach, and a more expressive classical approach. Since our approach is built on search in planning domains, we presented related work split into three core sections. First, we detailed approaches to generalised planning that have strong similarities with optimisations in our approach, but differentiate the two by arguing that our approach reasons about runs which are liable to change due to refinement. Similarly, we presented related work on incorporating control knowledge into planning and argued that while these approaches can satisfy a social objective, they cannot produce explicitly represented norms, nor can they reason about the negative effects of these norms. Finally, we detailed graph-based problem representations that was an inspiration for our traversal graph data structures.

Our presentation of the conflict-rooted synthesis algorithm is split into the core approach in Chapter 4 and a set of domain independent optimisations in Chapter 5. Conflict traversal searches the conflict state space to identify what agents might achieve if they are able to enter conflict states. We then synthesise norms and conduct a reachability analysis for each run, determining whether the same states are achievable in the normative system. While this approach to norm synthesis is sound, it is also complex since traversal is a complete search of the state space. In propositional domains this traversal terminates, but in classical domains the lack of initial state knowledge results in no termination guarantees. The optimisations introduced aim to improve the performance of the traversal and reachability analysis steps, while ensuring that the optimised algorithms is still sound.

In order to judge the performance of conflict-rooted synthesis we evaluated it on a set of benchmark domains common in planning literature. The specifics of our implementation, called CRS, were presented in Chapter 6. We refrained from presenting every detail of the implementation, instead focusing on the novel aspects of our implementation. CRS was compared empirically and theoretically to NuSMV in Chapter 7 and further discussion followed in Chapter 8. CRS was shown to scale more favourable than NuSMV as the domain size increases, particularly in domains where the conflict state space remains constant. Here, while NuSMV enumerates the entire system unnecessarily, CRS simply searches locally around conflict specifications. This results in improved performance and scalability. Furthermore, we detailed how the norms produced by CRS are minimal, in the sense that no smaller subset of norms exists that ensures the social objective is met, without violating the autonomy of agents in the system. Importantly, CRS provides greater domain coverage as NuSMV is not able to synthesise norms in domains where focal states are terminal. Finally, CRS is anytime, implying that some guarantees over the effects of synthesised norms are possible when the algorithm is terminated prematurely.

9.2 Main Contributions

We now summarise the main contributions of this research:

- **Norm Synthesis without Goal Knowledge**: Previous approaches have required explicit knowledge of focal states. The most fundamental contribution of this work is the definition of the problem of synthesising norms in domains where no explicit goal knowledge is provided, or where listing goal states is not feasible.
- **Conflict-Rooted Synthesis**: The next contribution is the conflict-rooted synthesis algorithm, showing that it is possible to synthesise norms in domains with no focal state knowledge, and to still provide guarantees over goal reachability. Our technique is theoretically sound, and designed to act as the formal underpinnings required to further investigate extensions in future work.
- **Optimisations**: The optimisations to the conflict-rooted synthesis approach are as important as the core algorithm, enabling this approach to be applied in more challenging benchmark domains.
- **CRS Implementation**: CRS is the main software deliverable of this work providing a stable implementation of conflict-rooted synthesis that can be adopted into future agent-based tooling. Furthermore, it provides a default, standard implementation that can be used by future approaches for comparison purposes.
- **Technique for Ungrounded Search**: More generally, this work contributes to Automated Planning and Artificial Intelligence, by illustrating a problem domain in which an ungrounded search-based approach is favourable to a ground one. We provide essential theory that can act as inspiration in applying ungrounded search-based techniques to new problems.

9.3 Future Work

The following possible directions represent avenues of future work to advance the conflict-rooted synthesis approach and to apply this work to new interesting problems and domains.

- **Improved Optimisations and Search Techniques**: There are strong similarities between our approach and other search-based solutions, and potentially existing techniques and optimisations can be incorporated into this work leading to performance gains that would increase the applicability of this work. Possible avenues for future work include:

  1. Investigate existing techniques adopted by search-based planners, and migrate the core concepts to this work, similar to how the work by Etzioni (1993) inspired our existing optimisations.
  2. Select a more recent planner with which to perform reachability analysis, providing improved performance and greater domain applicability. For example, SGPlan$_6$ (Hsu and Wah, 2008) was very successful in the deterministic track of IPC 2008, and full source code is available.
  3. Introduce a new class of optimisations and heuristics to simplify the problem of enumerating all possible specification bindings. Identifying bindings for sets of predicates is a common requirement for predicate-based programming languages and techniques, and this work would identify heuristics to be incorporated into conflict-rooted synthesis.

- **Increase Conflict Specification Expressivity**: Our conflict specification representation is simple, and potentially too limiting for many real world requirements. There are two feasible approaches to increasing this expressivity:

  1. Introduce a formal logic-based representation for conflict states. In their work on TLPLAN, Bacchus and Kabanza (2000) introduced progression as a technique to efficiently incorporate LTL control rules into planning. Future work could adapt these progression concepts to allow for LTL-based conflict state representations. If not, a more restricted form of LTL could be incorporated, still allowing for increased conflict-state expressivity.
  2. Conflict specifications are purely state-based in that they contain no action information. An avenue of research is to investigate how action-based specifications are incorporated into the conflict representation, perhaps taking cues from the action logics adopted by TALplanner (Kvarnström and Doherty, 2001). This allows for the specification of conflict using both action and state information.

- **Increase Formalism Expressivity**: A limitation of our approach is that it only operates on classical planning domains that allow incomplete state specifications and literal preconditions. Additional expressivity could be introduced by incorporating support for Boolean expressions in operator preconditions. Since preconditions are no longer sets of literals, our refinement and inference operators should be adjusted. Boolean formulae reduction techniques should be investigated to ensure that the resulting formulae generated after the reverse operator application will be minimal in representation (with respect to the ordering of literals within the formulae). Ordered Binary Decision Diagrams are a viable means of performing such a reduction (Bryant, 1992).

- **Identify Termination Properties**: Traversal in classical domains is not guaranteed to terminate if no bounds are applied. However, in certain domains such as Logistics and Parcel Delivery, the Repetitive Operators optimisation resulted in the traversal terminating. In all other domains bounds were required. A valuable avenue of future work might identify the properties of domains that terminate, so that these domains can be classified. Then, future optimisations can be gauged not only on performance, but also on the set of domains that are entirely solved by them.

- **Improve CRS Performance**: The performance of norm synthesis is important in online systems. A revision and reimplementation of CRS would lead to greatly improved performance, reducing the problems in which NuSMV is applicable and increasing the number of domains in which CRS completes. A native, C-based implementation would result in such improvements, while the process would also result in a cleaner codebase for future work.

Additionally, a large set of possible applications of our research exist outside of the problem of norm synthesis. For instance, it would be interesting to identify whether techniques developed in this work could be used for plan synthesis, and whether the optimisations offer are avenues to synthesising generalised plans. In this way, traversal graphs may represent an interesting representation for such generalised plans, incorporating uncertainty in state specifications and looping over operators with an efficient mechanism to check generalised plan applicability.

9.4 Final Remarks

At the most fundamental level, this work is concerned with the preservation of intuitive, abstract domain representations during search. The guiding principle behind our work is one of least commitment, where we only enumerate abstract representations if it is necessary, allowing us to produce results that are independent of individual states of the system. Contrary to most other Artificial Intelligence search problems, norm synthesis illustrates that searching a simpler, enumerated system is less advantageous to searching a more abstract representation. In problems such as these, traditional search techniques produce results of poorer quality since the intuitive, abstract representation of the system is discarded.

This thesis presents a contribution towards the development of multiagent systems and truly autonomous agents, describing the first viable approach to social norm synthesis that produces concise, generally applicable norms. Conflict-rooted synthesis is an algorithm that allows designers to shape the global computation of large-scale distributed systems, requiring participating agents to coordinate in order to achieve a social objective while ensuring that any system modifications maintain system flexibility. From an agent designers perspective, conflict-rooted synthesis is the first anytime approach to online norm design, enabling intelligent agents to reason with, and affect the norms that govern them, allowing for the design and implementation of truly norm autonomous agents.

Appendix A

PDDL Operator Schemata

For reference purposes we present the complete PDDL domain specifications for our evaluation domains.

Parcel Delivery Domain

```
(define (domain ParcelDomain)
    (:requirements :typing)
    (:types parcel location agent)


    (:predicates
        (parcelAt ?parcel - parcel ?x - location)
        (has ?agent - agent ?parcel - parcel)
        (agentAt ?agent - agent ?x - location)
        (conn ?l1 - location ?l2 - location))


    (:action MOVE
        :parameters (?agent - agent ?l1 - location ?l2 - location)
        :precondition (and
                        (conn ?l1 ?l2)
                        (agentAt ?agent ?l1))
        :effect (and
                        (not (agentAt ?agent ?l1))
                        (agentAt ?agent ?l2)))


    (:action DROP
        :parameters (?agent - agent ?x - location ?parcel - parcel)
        :precondition (and
                        (has ?agent ?parcel)
                        (agentAt ?agent ?x))
        :effect (and
                        (not (has ?agent ?parcel))
                        (parcelAt ?parcel ?x)))
    (:action PICKUP
        :parameters (?agent - agent ?x - location ?parcel - parcel)
        :precondition (and
                        (parcelAt ?parcel ?x)
                        (agentAt ?agent ?x))
        :effect (and
                        (not (parcelAt ?parcel ?x))
                        (has ?agent ?parcel)))


    (:action IDLE
        :parameters (?agent - agent)
        :precondition ()
        :effect ())
)
```

Logistics Domain

```
(define (domain logistics)
    (:requirements :strips :typing)
    (:types
        city - object
        place - object
        physobj - object
        package - physobj
        vehicle - physobj
        truck - vehicle
        airplane - vehicle
        airport - place
        location - place)


    (:predicates  (in-city ?loc - place ?city - city)
        (atLocation ?obj - physobj ?loc - place)
        (in ?pkg - package ?veh - vehicle))


    (:action LOAD-TRUCK
        :parameters    (?pkg - package ?truck - truck ?loc - place)
        :precondition  (and
                        (atLocation ?truck ?loc)
                        (atLocation ?pkg ?loc))
        :effect (and
                        (not (atLocation ?pkg ?loc))
                        (in ?pkg ?truck)))


    (:action LOAD-AIRPLANE
        :parameters   (?pkg - package ?airplane - airplane ?loc - place)
        :precondition (and
                        (atLocation ?pkg ?loc)
                        (atLocation ?airplane ?loc))
        :effect (and
                        (not (atLocation ?pkg ?loc))
                        (in ?pkg ?airplane)))


    (:action UNLOAD-TRUCK
        :parameters   (?pkg - package ?truck - truck ?loc - place)
        :precondition (and
                        (atLocation ?truck ?loc)
                        (in ?pkg ?truck))
        :effect (and
                        (not (in ?pkg ?truck))
                        (atLocation ?pkg ?loc)))


    (:action UNLOAD-AIRPLANE
        :parameters    (?pkg - package ?airplane - airplane ?loc - place)
        :precondition  (and
                        (in ?pkg ?airplane)
                        (atLocation ?airplane ?loc))
        :effect (and
                        (not (in ?pkg ?airplane))
                        (atLocation ?pkg ?loc)))


    (:action DRIVE-TRUCK
        :parameters (?truck - truck ?loc-from - place ?loc-to - place ?city - city)
        :precondition (and
                        (atLocation ?truck ?loc-from)
                        (in-city ?loc-from ?city)
                        (in-city ?loc-to ?city))
        :effect
                        (and (not (atLocation ?truck ?loc-from))
                        (atLocation ?truck ?loc-to)))


    (:action FLY-AIRPLANE
        :parameters (?airplane - airplane ?loc-from - airport ?loc-to - airport)
        :precondition (and
                        (atLocation ?airplane ?loc-from))
        :effect (and
                        (not (atLocation ?airplane ?loc-from))
                        (atLocation ?airplane ?loc-to)))
)
```

Satellites Domain

```
(define (domain satellite)
    (:requirements :strips :equality :typing)
    (:types satellite direction instrument mode)


    (:predicates
        (on_board ?i - instrument ?s - satellite)
        (supports ?i - instrument ?m - mode)
        (pointing ?s - satellite ?d - direction)
        (power_avail ?s - satellite)
        (power_on ?i - instrument)
        (calibrated ?i - instrument)
        (have_image ?d - direction ?m - mode)
        (calibration_target ?i - instrument ?d - direction))


    (:action TURN_TO
        :parameters (?s - satellite ?d_new - direction ?d_prev - direction)
        :precondition (and
                        (pointing ?s ?d_prev))
        :effect (and
                        (pointing ?s ?d_new)
                        (not (pointing ?s ?d_prev))))


    (:action SWITCH_ON
        :parameters (?i - instrument ?s - satellite)
        :precondition (and
                        (on_board ?i ?s)
                        (power_avail ?s))
        :effect (and (power_on ?i)
                        (not (calibrated ?i))
                        (not (power_avail ?s))))


     (:action SWITCH_OFF
        :parameters (?i - instrument ?s - satellite)
        :precondition (and
                        (on_board ?i ?s)
                        (power_on ?i))
        :effect (and
                        (not (power_on ?i))
                        (power_avail ?s)))


    (:action CALIBRATE
        :parameters (?s - satellite ?i - instrument ?d - direction)
        :precondition (and
                        (on_board ?i ?s)
                        (calibration_target ?i ?d)
                        (pointing ?s ?d)
                        (power_on ?i))
        :effect (and (calibrated ?i)))


    (:action TAKE_IMAGE
        :parameters (?s - satellite ?d - direction ?i - instrument ?m - mode)
        :precondition (and
                        (calibrated ?i) (on_board ?i ?s)
                        (supports ?i ?m) (power_on ?i)
                        (pointing ?s ?d) (power_on ?i))
        :effect (and (have_image ?d ?m))))
)
```

Rovers Domain

```
(define (domain Rover)
    (:requirements :typing)
    (:types rover waypoint store camera mode lander objective)


    (:predicates
        (at_rover ?x - rover ?y - waypoint)
        (at_lander ?x - lander ?y - waypoint)
        (can_traverse ?r - rover ?x - waypoint ?y - waypoint)
        (equipped_for_soil_analysis ?r - rover)
        (equipped_for_rock_analysis ?r - rover)
        (equipped_for_imaging ?r - rover)
        (empty ?s - store)
        (have_rock_analysis ?r - rover ?w - waypoint)
        (have_soil_analysis ?r - rover ?w - waypoint)
        (full ?s - store)
        (calibrated ?c - camera ?r - rover)
        (supports ?c - camera ?m - mode)
        (available ?r - rover)
        (visible ?w - waypoint ?p - waypoint)
        (have_image ?r - rover ?o - objective ?m - mode)
        (communicated_soil_data ?w - waypoint)
        (communicated_rock_data ?w - waypoint)
        (communicated_image_data ?o - objective ?m - mode)
        (at_soil_sample ?w - waypoint)
        (at_rock_sample ?w - waypoint)
        (visible_from ?o - objective ?w - waypoint)
        (store_of ?s - store ?r - rover)
        (calibration_target ?i - camera ?o - objective)
        (on_board ?i - camera ?r - rover)
        (channel_free ?l - lander))


    (:action NAVIGATE
        :parameters (?x - rover ?y - waypoint ?z - waypoint)
        :precondition (and
                        (can_traverse ?x ?y ?z) (available ?x)
                        (at_rover ?x ?y) (visible ?y ?z))
        :effect (and
                        (not (at_rover ?x ?y)) (at_rover ?x ?z)))


    (:action SAMPLE_SOIL
        :parameters (?x - rover ?s - store ?p - waypoint)
        :precondition (and
                        (at_rover ?x ?p) (at_soil_sample ?p)
                        (equipped_for_soil_analysis ?x) (store_of ?s ?x)
                        (empty ?s))
        :effect (and
                        (not (empty ?s)) (full ?s)
                        (have_soil_analysis ?x ?p) (not (at_soil_sample ?p))))


    (:action SAMPLE_ROCK
    :parameters (?x - rover ?s - store ?p - waypoint)
    :precondition (and
                   (at_rover ?x ?p) (at_rock_sample ?p)
                   (equipped_for_rock_analysis ?x) (store_of ?s ?x)
                   (empty ?s))
    :effect (and (not (empty ?s)) (full ?s)
                   (have_rock_analysis ?x ?p) (not (at_rock_sample ?p))))


(:action DROP
    :parameters (?x - rover ?y - store)
    :precondition (and
                   (store_of ?y ?x) (full ?y))
    :effect (and
                   (not (full ?y)) (empty ?y)))


(:action CALIBRATE
     :parameters (?r - rover ?i - camera ?t - objective ?w - waypoint)
     :precondition (and
                    (equipped_for_imaging ?r) (calibration_target ?i ?t)
                    (at_rover ?r ?w) (visible_from ?t ?w)(on_board ?i ?r))
     :effect (and
                    (calibrated ?i ?r)))


(:action TAKE_IMAGE
    :parameters (?r - rover ?p - waypoint ?o - objective ?i - camera ?m - mode)
    :precondition (and
                   (calibrated ?i ?r) (on_board ?i ?r)
                   (equipped_for_imaging ?r) (supports ?i ?m)
                   (visible_from ?o ?p) (at_rover ?r ?p))
    :effect (and
                   (have_image ?r ?o ?m) (not (calibrated ?i ?r))))


(:action COMMUNICATE_SOIL_DATA
    :parameters (?r - rover ?l - lander ?p - waypoint ?x - waypoint ?y - waypoint)
    :precondition (and
                   (at_rover ?r ?x) (at_lander ?l ?y)
                   (have_soil_analysis ?r ?p) (visible ?x ?y)
                   (available ?r) (channel_free ?l))
    :effect (and (not
                   (available ?r)) (not (channel_free ?l))
                   (channel_free ?l) (communicated_soil_data ?p)
                   (available ?r)))


(:action COMMUNICATE_ROCK_DATA
    :parameters (?r - rover ?l - lander ?p - waypoint ?x - waypoint ?y - waypoint)
    :precondition (and
                   (at_rover ?r ?x)(at_lander ?l ?y)
                   (have_rock_analysis ?r ?p)
                   (visible ?x ?y)(available ?r)(channel_free ?l))
    :effect (and
                   (not (available ?r))(not (channel_free ?l))
```

```
                    (channel_free ?l)(communicated_rock_data ?p)
                    (available ?r)))


    (:action COMMUNICATE_IMAGE_DATA
        :parameters (?r - rover ?l - lander ?o - objective ?m - mode
                                ?x - waypoint ?y - waypoint)
        :precondition (and
                    (at_rover ?r ?x)(at_lander ?l ?y)
                    (have_image ?r ?o ?m)(visible ?x ?y)
                    (available ?r)(channel_free ?l))
        :effect (and
                    (not (available ?r))(not (channel_free ?l))
                    (channel_free ?l)(communicated_image_data ?o ?m)
                    (available ?r)))
)
```

Depots Domain

```
(define (domain Depot)
(:requirements :typing)
    (:types place locatable - object
        depot distributor - place
        truck hoist surface - locatable
        pallet crate - surface)


    (:predicates
        (atLocation ?locatable - locatable ?place - place)
        (on ?crate - crate ?surface - surface)
        (in ?crate - crate ?truck - truck)
        (lifting ?hoist - hoist ?crate - crate)
        (available ?hoist - hoist)
        (clear ?surface - surface))


    (:action DRIVE
        :parameters (?truck - truck ?place - place ?place - place)
        :precondition (and
                    (atLocation ?truck ?place))
        :effect (and
                    (not (atLocation ?truck ?place)) (atLocation ?truck ?place)))


    (:action LIFT
        :parameters (?hoist - hoist ?crate - crate ?surface - surface ?place - place)
        :precondition (and
                    (atLocation ?hoist ?place) (available ?hoist)
                    (atLocation ?crate ?place) (on ?crate ?surface)
                    (clear ?crate))
        :effect (and
                    (not (atLocation ?crate ?place)) (lifting ?hoist ?crate)
                    (not (clear ?crate)) (not (available ?hoist))
                    (clear ?surface) (not (on ?crate ?surface))))
    (:action DROP
        :parameters (?hoist - hoist ?crate - crate ?surface - surface ?place - place)
        :precondition (and
                        (atLocation ?hoist ?place) (atLocation ?surface ?place)
                        (clear ?surface) (lifting ?hoist ?crate))
        :effect (and
                        (available ?hoist) (not (lifting ?hoist ?crate))
                        (atLocation ?crate ?place) (not (clear ?surface))
                        (clear ?crate) (on ?crate ?surface)))


    (:action LOAD
        :parameters (?hoist - hoist ?crate - crate ?truck - truck ?place - place)
        :precondition (and
                        (atLocation ?hoist ?place) (atLocation ?truck ?place)
                        (lifting ?hoist ?crate))
        :effect (and
                        (not (lifting ?hoist ?crate)) (in ?crate ?truck)
                        (available ?hoist)))


    (:action UNLOAD
        :parameters (?hoist - hoist ?crate - crate ?truck - truck ?place - place)
        :precondition (and
                        (atLocation ?hoist ?place) (atLocation ?truck ?place)
                        (available ?hoist) (in ?crate ?truck))
        :effect (and
                        (not (in ?crate ?truck)) (not (available ?hoist))
                        (lifting ?hoist ?crate)))
)
```

Appendix B

SMV Sample Input Model

Below we present an example NuSMV input model representing a 2x2 Parcel Delivery domain, composed of 2 agents that are restricted to just the `move` operator.

```
MODULE main
    VAR
        normsystem: process normsystem();
    ASSIGN
        SPEC ! EG (
            !(normsystem.agentat_agent1_node4 & normsystem.agentat_agent2_node4) &
            !(normsystem.agentat_agent1_node3 & normsystem.agentat_agent2_node3) &
            !(normsystem.agentat_agent1_node1 & normsystem.agentat_agent2_node1) &
            !(normsystem.agentat_agent1_node2 & normsystem.agentat_agent2_node2)
        )
        FAIRNESS
            normsystem.agentat_agent1_node4 & normsystem.agentat_agent2_node3
        FAIRNESS
            normsystem.agentat_agent1_node4 & normsystem.agentat_agent2_node2
        FAIRNESS
            normsystem.agentat_agent1_node4 & normsystem.agentat_agent2_node1
        FAIRNESS
            normsystem.agentat_agent1_node2 & normsystem.agentat_agent2_node4
        FAIRNESS
            normsystem.agentat_agent1_node2 & normsystem.agentat_agent2_node3
        FAIRNESS
            normsystem.agentat_agent1_node2 & normsystem.agentat_agent2_node1
        FAIRNESS
            normsystem.agentat_agent1_node3 & normsystem.agentat_agent2_node4
        FAIRNESS
            normsystem.agentat_agent1_node3 & normsystem.agentat_agent2_node2
        FAIRNESS
            normsystem.agentat_agent1_node3 & normsystem.agentat_agent2_node1
        FAIRNESS
            normsystem.agentat_agent1_node1 & normsystem.agentat_agent2_node4
        FAIRNESS
            normsystem.agentat_agent1_node1 & normsystem.agentat_agent2_node3
        FAIRNESS
            normsystem.agentat_agent1_node1 & normsystem.agentat_agent2_node2
```

```
MODULE normsystem
    VAR
        agentat_agent2_node4 :boolean;
        agentat_agent1_node4 :boolean;
        agentat_agent2_node3 :boolean;
        agentat_agent2_node2 :boolean;
        agentat_agent1_node2 :boolean;
        agentat_agent2_node1 :boolean;
        agentat_agent1_node3 :boolean;
        agentat_agent1_node1 :boolean;
    DEFINE
        conn_node2_node1 := TRUE;
        conn_node3_node4 := TRUE;
        conn_node4_node2 := TRUE;
        conn_node4_node3 := TRUE;
        conn_node2_node4 := TRUE;
        conn_node1_node2 := TRUE;
        conn_node3_node1 := TRUE;
        conn_node1_node3 := TRUE;
        move_agent1_node4_node3 := agentat_agent1_node4;
        move_agent1_node4_node2 := agentat_agent1_node4;
        move_agent1_node2_node1 := agentat_agent1_node2;
        move_agent1_node2_node4 := agentat_agent1_node2;
        move_agent2_node3_node4 := agentat_agent2_node3;
        move_agent1_node3_node4 := agentat_agent1_node3;
        move_agent2_node3_node1 := agentat_agent2_node3;
        move_agent1_node3_node1 := agentat_agent1_node3;
        move_agent2_node4_node3 := agentat_agent2_node4;
        move_agent2_node4_node2 := agentat_agent2_node4;
        move_agent2_node1_node2 := agentat_agent2_node1;
        move_agent1_node1_node2 := agentat_agent1_node1;
        move_agent2_node1_node3 := agentat_agent2_node1;
        move_agent1_node1_node3 := agentat_agent1_node1;
        move_agent2_node2_node1 := agentat_agent2_node2;
        move_agent2_node2_node4 := agentat_agent2_node2;
    INIT
        agentat_agent2_node2 & agentat_agent1_node1 &
        !agentat_agent1_node4 & !agentat_agent2_node4 &
        !agentat_agent2_node3 & !agentat_agent1_node2 &
        !agentat_agent1_node3 & !agentat_agent2_node1;
    TRANS
        (move_agent1_node4_node3 &
                next(agentat_agent1_node3) & !next(agentat_agent1_node4) &
                (agentat_agent2_node4 <-> next(agentat_agent2_node4)) &
                (agentat_agent2_node3 <-> next(agentat_agent2_node3)) &
                (agentat_agent1_node2 <-> next(agentat_agent1_node2)) &
                (agentat_agent2_node2 <-> next(agentat_agent2_node2)) &
                (agentat_agent2_node1 <-> next(agentat_agent2_node1)) &
                (agentat_agent1_node1 <-> next(agentat_agent1_node1)))
        xor
        (move_agent1_node4_node2 &
                 next(agentat_agent1_node2) & !next(agentat_agent1_node4) &
                 (agentat_agent2_node4 <-> next(agentat_agent2_node4)) &
                 (agentat_agent2_node3 <-> next(agentat_agent2_node3)) &
                 (agentat_agent2_node2 <-> next(agentat_agent2_node2)) &
```

```
          (agentat_agent1_node3 <-> next(agentat_agent1_node3)) &
          (agentat_agent2_node1 <-> next(agentat_agent2_node1)) &
          (agentat_agent1_node1 <-> next(agentat_agent1_node1)))
  xor

...
  xor
  (move_agent2_node2_node4 &
         next(agentat_agent2_node4) & !next(agentat_agent2_node2) &
          (agentat_agent1_node4 <-> next(agentat_agent1_node4)) &
          (agentat_agent2_node3 <-> next(agentat_agent2_node3)) &
          (agentat_agent1_node2 <-> next(agentat_agent1_node2)) &
          (agentat_agent1_node3 <-> next(agentat_agent1_node3)) &
          (agentat_agent2_node1 <-> next(agentat_agent2_node1)) &
          (agentat_agent1_node1 <-> next(agentat_agent1_node1)));
```

Bibliography

Agents Group (2011). University of edinburgh agents group. `http://www.cisa.inf.ed.ac.uk/agents/`.

Ågotnes, T., van der Hoek, W., Tennenholtz, M., and Wooldridge, M. (2009). Power in normative systems. In Decker, K. S., Sichman, J. S., Sierra, C., and Castelfranchi, C., editors, *Proceedings of the Eigth International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009)*, pages 145–152.

Ågotnes, T., Wooldridge, M., and van der Hoek, W. (2007). Normative system games. In Huhns, M. and Shehory, O., editors, *Proceedings of the Sixth International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2007)*, pages 876–883.

Aldewereld, H., García-Camino, A., Dignum, F., Noriega, P., Rodríguez-Aguilar, J. A., and Sierra, C. (2007). Operationalisation of norms for electronic institutions. In Noriega, P., Vázquez-Salceda, J., Boella, G., Boissier, O., Dignum, V., Fornara, N., and Matson, E., editors, *Coordination, Organizations, Institutions, and Norms in Agent Systems II*, volume 4386, pages 163–176.

Alur, R., Henzinger, T., and Kupferman, O. (2002). Alternating-time temporal logic. *Journal of the ACM*, 49(5):672–713.

Alur, R., Henzinger, T., Mang, F., Qadeer, S., Rajamani, S., and Tasiran, S. (1998). MOCHA: Modularity in model checking. In Hu, A. and Vardi, M., editors, *Proceedings of the Tenth International Conference on Computer-Aided Verification (CAV)*, volume 1427 of *Lecture Notes in Computer Science*, pages 521–525.

Alur, R. and Henzinger, T. A. (1999). Reactive modules. *Formal Methods in System Design*, 15:7–48.

Axelrod, R. (1986). An evolutionary approach to norms. *The American Political Science Review*, 80(4):1095–1111.

Bacchus, F. and Kabanza, F. (2000). Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 16(1-2):123–191.

Baier, J. A., Fritz, C., Bienvenu, M., and McIlraith, S. A. (2008). Beyond classical planning: Procedural control knowledge and preferences in state-of-the-art planners. In Fox, D. and Carla, P. G., editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008)*, pages 1509–1512.

Baier, J. A., Fritz, C., and McIlraith, S. A. (2007). Exploiting procedural domain control knowledge in state-of-the-art planners. In Boddy, M. S., Fox, M., and Sylvie, T., editors, *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS 2007)*, pages 26–33.

Barret, A., Christianson, D., Friedman, M., Kwok, C., Golden, K., Penberthy, S., Sun, Y., and Weld, D. (1995). UCPOP user's manual. Technical report, University of Washington, Seattle, WA 98105.

Blum, A. L. and Furst, M. L. (1997). Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1–2):281–300.

Boella, G. and van der Torre, L. (2005). Enforceable social laws. In Dignum, F., Dignum, V., Koenig, S., Kraus, S., Singh, M. P., and Wooldridge, M., editors, *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005)*, pages 682–689.

Boella, G. and van der Torre, L. (2007). Norm negotiation in multiagent systens. *International Journal of Cooperative Information Systems (IJCIS) Special Issue: Emergent Agent Societies*, 16(1):97–122.

Boella, G., var der Torre, L., and Verhagen, H. (2006). Introduction to normative multiagent systems. *Computational and Mathematical Organization Theory*, 12(2-3):71–79.

Bryant, R. E. (1992). Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Survey*, 24(3):293–318.

Büchi, J. R. (1966). On a decision method in restricted second order arithmetic. In Ernest Nagel, P. S. and Tarski, A., editors, *Logic, Methodology and Philosophy of Science, Proceeding of the 1960 International Congress*, volume 44, pages 1–11. Elsevier.

Bylander, T. (1994). The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1–2):165–204.

Castelfranchi, C., Dignum, F., Jonker, C., and Treur, J. (2000). Deliberative normative agents: Principles and architecture. In Jennings, N. R. and Lespérance, Y., editors, *Proceedings of the Sixth International Workshop on Intelligent Agents, Agent Theories, Architectures, and Languages (ATAL 1999)*, pages 364–378.

Christelis, G. and Rovatsos, M. (2009). Automated norm synthesis in agent-based planning environment. In Decker, K. S., Sichman, J. S., Sierra, C., and Castelfranchi, C., editors, *Proceedings of the Eigth International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009)*, pages 161–168.

Christelis, G., Rovatsos, M., and Petrick, R. P. (2010). Exploiting domain knowledge to improve norm synthesis. In van der Hoek, W., Kaminka, G. A., Luck, M., and Sen, S., editors, *Proceedings of the Ninth International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2010)*, pages 831–838.

Cimatti, A., Clarke, E. M., Giunchiglia, F., and Roveri, M. (1999). NUSMV: A new symbolic model verifier. In Peled, D. and Halbwachs, N., editors, *Proceedings of the Eleventh International Conference on Computer Aided Verification (CAV 1999)*, pages 495–499.

Clement, B. J. and Barrett, A. C. (2003). Continual coordination through shared activities. In Rosenschein, J. S., Sandholm, T., Wooldridge, M., and Yokoo, M., editors, *Proceedings of the Second International Conference on Autonomous agents and Multiagent Systems (AAMAS 2003)*, pages 57–64.

Cohen, P. and Levesque, H. (1991). Teamwork. *Noûs: Special Issue on Cognitive Science and Artificial Intelligence*, 25(4):487–512.

Conte, R., Castelfranchi, C., and Dignum, F. (1999a). Autonomous norm acceptance. In Müller, J., Singh, M. P., and Rao, A. S., editors, *Proceedings of the Fifth International Workshop on Intelligent Agents: Agent Theories, Architectures, and Languages (ATAL 1999)*, volume 1555, pages 99–112.

Conte, R., Falcone, R., and Sartor, G. (1999b). Introduction: Agents and norms: How to fill the gap? *Artificial Intelligence and Law*, 7(1):1–15.

Cook, S. (1971). The complexity of theorem proving procedures. In Harrison, M. A., Ranan, B. B., and Jeffrey, D. U., editors, *Proceedings of the Third Annual ACM Symposium on Theory of Computing (STOC 1971)*, pages 151–158.

Cortés, U. (2004). Electronic institutions and agents. In *AgentLink News 15*, pages 14–15.

Cresswell, S. and Coddington, A. M. (2004). Compilation of LTL goal formulas into PDDL. In Lopez de Mantaras, R. and Saitta, L., editors, *Proceedings of the Sixteenth European Conference on Artificial Intelligence (ECAI 2004)*, pages 985–986.

Decker, K. and Lesser, V. (1995). Designing a family of coordination algorithms. In Lesser, V. R. and Gasser, L., editors, *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS 1995)*, pages 73–80.

Decker, K. and Li, J. (2000). Coordinating mutually exclusive resources using GPGP. *Autonomous Agents and Multi-Agent Systems*, 3(2):133–157.

Delgado, J. (2002). Emergence of social conventions in complex networks. *Artificial Intelligence*, 141(1):171–185.

Dignum, F. (1999). Autonomous agents with norms. *Artificial Intelligence and Law*, 7(1):69–79.

Dignum, F., Kinny, D., and Sonenberg, L. (2002). From desires, obligations and norms to goals. *Cognitive Science Quarterly*, 2(3–4):407–430.

Durfee, E. and Lesser, V. (1991). Partial global planning: a coordination framework for distributed hypothesis formation. *IEEE Transactions on Systems, Man and Cybernetics*, 21(5):1167–1183.

Edelkamp, S. (2006). On the compilation of plan constraints and preferences. In Long, D. and Smith, S., editors, *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling (ICAPS 2006)*, pages 374–377.

Elster, J. (1989). Social norms and economic theory. *The Journal of Economic Perspectives*, 3(4):99–117.

Epstein, J. M. (2001). Learning to be thoughtless: Social norms and individual computation. *Computational Economics*, 18(1):9–24.

Esteva, M., Padget, J. A., and Sierra, C. (2002). Formalizing a language for institutions and norms. In Meyer, J.-J. C. and Tambe, M., editors, *Proceedings of the Eighth International Workshop on Intelligent Agents*, volume 2333 of *Lecture Notes in Computer Science*, pages 348–366.

Estlin, T. A. and Mooney, R. J. (1996). Hybrid learning of search control for partial-order planning. In Ghallab, M. and Milani, A., editors, *New Directions in AI Planning*, pages 129–140.

Etzioni, O. (1993). A structural theory of explanation-based learning. *Artificial Intelligence*, 60(1):93–139.

Excelente-Toledo, C. B., Bourne, R. A., and Jennings, N. R. (2001). Reasoning about commitments and penalties for coordination between autonomous agents. In Müller, J. P., Andre, E., Sen, S., and Frasson, C., editors, *Proceedings of the Fifth International Conference on Autonomous Agents (Agents 2001)*, pages 131–138.

Fikes, R. and Nilsson, N. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3–4):189–208.

Fitoussi, D. and Tennenholtz, M. (2000). Choosing social laws for multi-agent systems: Minimality and simplicity. *Artificial Intelligence*, 119(1–2):61–101.

Fornara, N. and Colombetti, M. (2006). Specifying and enforcing norms in artificial insitutions. In Omicini, A., Dunin-Keplicz, B., and Padget, J., editors, *Proceedings of the Fourth European Workshop on Multi-Agent Systems (EUMAS 2006).*

Gazen, B. C. and Knoblock, C. A. (1997). Combining the expressivity of UCPOP with the efficiency of GraphPlan. In Steel, S. and Alami, R., editors, *Proceedings of the Fourth European Conference on Planning (ECP 1997)*, pages 221–233.

Giunchiglia, E., Maratea, M., Tacchella, A., and Zambonin, D. (2001). Evaluating search heuristics and optimization techniques in propositional satisfiability. In Goré, R., Leitsch, A., and Nipkow, T., editors, *Proceedings of the First International Joint Conference on Automated Reasoning (IJCAR 2001)*, pages 347–363.

Griffiths, N. and Luck, M. (2010). Norm emergence in tag-based cooperation. In De Voc, M., Fornara, N., Pitt, J. V., and Vouros, G., editors, *Proceedings of the Ninth International Workshop on Coordination, Organization, Institutions and Norms in Multi-Agent Systems (COIN 2010)*, volume 6541 of *Lecture Notes in Artificial Intelligence*, pages 80–87.

Grizard, A., Vercouter, L., Stratulat, T., and Muller, G. (2006). A peer-to-peer normative system to achieve social order. In Vázquez-Salceda, J., Boella, G., Boissier, O., Dignum, V., Fornara, N., and Matson, E., editors, *Proceedings of the Fifth International Workshop on Coordination, Organization, Institutions and Norms in Multi-Agent Systems (COIN 2006)*, volume 4386 of *Lecture Notes in Artificial Intelligence*, pages 274–289.

Grossi, D., Aldewereld, H., and Dignum, F. (2007). Ubi lex, ibi poena: Designing norm enforcement in e-institutions. In Noriega, P., Vázquez-Salceda, J., Boella, G., Boissier, O., Dignum, V., Fornara, N., and Matson, E., editors, *Coordination, Organizations, Institutions, and Norms in Agent Systems II*, Lecture Notes in Artificial Intelligence, pages 101–114. Springer.

Grossi, D. and Dignum, F. (2005). From abstract to concrete norms in agent institutions. In Hinchey, M. G., Rash, J., Truszkowski, W., and Rouff, C., editors, *Proceedings of the Third NASA Workshop on Formal Approaches to Agent-Based Systems (FAABS III)*, volume 3228 of *Lecture Notes in Computer Science*, pages 12–29. Springer.

Hoffmann, J. and Nebel, B. (2001). The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14(1):253–302.

Hsu, C.-w. and Wah, B. W. (2008). The SGPlan planning system in IPC-6. In *Proceedings of the Sixth International Planning Competition Booklet*.

IPC (2011). International planning competition. `http://ipc.icaps-conference.org/`.

Jennings, N. R. (1993). Commitments and conventions: The foundation of coordination in multi-agent systems. *The Knowledge Engineering Review*, 8(3):223–250.

Jones, A. J. I. and Sergot, M. (1993). *On the characterization of law and computer systems: the normative systems perspective*, pages 275–307. John Wiley and Sons Ltd., Chichester, UK.

Kittock, J. E. (1993). Emergent conventions and the structure of multi-agent systems. In *Lectures in Complex Systems: Proceedings of the 1993 Complex systems summer school*, volume VI, pages 507–521. Addison-Wesley.

Koeppen, J. and López-Sánchez, M. (2010). Generating new regulations by learning from experience. In De Voc, M., Fornara, N., Pitt, J. V., and Vouros, G., editors, *Proceedings of the Ninth International Workshop on Coordination, Organization, Institutions and Norms in Multi-Agent Systems (COIN 2010)*, volume 6521 of *Lecture Notes in Artificial Intelligence*, pages 72–79.

Kollingbaum, M. J. (2005). *Norm-Governed Practical Reasoning Agents*. PhD thesis, University of Aberdeen.

Kollingbaum, M. J. and Norman, T. J. (2003). Norm adoption in the NoA agent architecture. In Rosenschein, J. S., Sandholm, T., Wooldridge, M., and Yokoo, M., editors, *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2003)*, pages 1038–1039.

Kvarnström, J. and Doherty, P. (2001). TALPlanner: A temporal logic based forward chaining planner. *Annals of Mathematics and Artificial Intelligence*, 30(1-4):119–169.

Levesque, H. J. (2005). Planning with loops. In Kaelbling, L. P. and Saffiotti, A., editors, *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI 2005)*, pages 509–515.

Lewis, D. (1969). *Convention: A Philosophical Study*. Harvard University Press, first edition.

López y López, F. and Arenas Marquez, A. (2004). An architecture for autonomous normative agents. In Baeza-Yates, R., Marroquin, J. L., and Chávez, E., editors, *Proeedings of the Fifth Mexican International Conference on Computer Science (ENC 2005)*, pages 96–103.

López y López, F. and Luck, M. (2002). Towards a model of the dynamics of normative multi-agent systems. In Lindermann, G., Moldt, D., Paolucci, M., and Yu, B., editors, *Proceedings of the International Workshop on Regulated Agent-Based Social Systems: Theories and Applications*, volume 318, pages 175–194.

López y López, F. and Luck, M. (2003). Modelling norms for autonomous agents. In Chávez, E., Favela, J., Mejía, M., and Oliart, A., editors, *Proceedings of the Fourth Mexican International Conference on Computer Science (ENC 2003)*, pages 238–245.

López y López, F., Luck, M., and d'Inverno, M. (2006). A normative framework for agent-based systems. *Computational and Mathematical Organization Theory*, 12(2-3):227–250.

Malone, T. W. and Croston, K. (1994). The interdisciplinary study of coordination. *ACM Computing Surveys*, 26(1):87–119.

McDermott, D. (2000). The 1998 AI planning systems competition. *Artificial Intelligence Magazine*, 21:35–55.

Meneguzzi, F. and Luck, M. (2009). Norm-based behaviour modification in bdi agents. In Decker, K. S., Sichman, J. S., Sierra, C., and Castelfranchi, C., editors, *Proceedings of the Eigth International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009)*, pages 177–184.

Meyer, J. J. C. (1988). A different approach to deontic logic: Deontic logic viewed as a variant of dynamic logic. *Notre Dame Journal of Formal Logic*, 29(1):109–136.

Milner, R. (2006). Ubiquitous computing: Shall we understand it? *The Computer Journal*, 49(4):383–389.

Mocha (2011). Mocha: Exploiting modularity in model checking. `http://mtc.epfl.ch/software-tools/mocha/`.

Moses, Y. and Tennenholtz, M. (1995). Artificial social systems. *Computers and Artificial Intelligence*, 14(6):533–562.

Nau, D., Ghallab, M., and Traverso, P. (2004a). *Automated Planning: Theory and Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, first edition.

Nau, D., Ghallab, M., and Traverso, P. (2004b). *Automated Planning: Theory and Practice*, pages 73–75. In Nau et al. (2004a), first edition.

Nebel, B. (2000). On the compilability and expressive power of propositional planning formalisms. *Journal of Artificial Intelligence Research*, 12(1):271–315.

Noriega, P. (1997). *Agent Mediated Auctions. The Fishmarket Metaphor.* PhD thesis, Universitat Autònoma de Barcelona.

Onn, S. and Tennenholtz, M. (1997). Determination of social laws for multi-agent mobilization. *Artificial Intelligence*, 95(1):155–167.

Parr, T. (2007). *The Definitive Antlr Reference: Building Domain-Specific Languages.* Pragmatic Bookshelp, 1st edition.

Pednault, E. (1987). Formulating multi-agent dynamic-world problems in the classical planning framework. In Georgeff, M. and Lansky, A., editors, *Proceedings of the 1996 Workshop on Reasoning about Actions and Plans*, pages 47–82.

Rao, A. S. (1996). AgentSpeak(L): BDI agents speak out in a logical computable language. In Van de Velde, W. and Perram, J. W., editors, *Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW 1996)*, pages 42–55.

Sagiv, M., Reps, T., and Wilhelm, R. (1999). Parametric shape analysis via 3-valued logic. In Appel, A. W. and Aiken, A., editors, *Proceedings of the Twenty-Sixth ACM Symposium on Principles of Programming Languages (POPL 1999)*, pages 105–118.

Saigol, Z. (2011). Pddl-driven graphplan implementation. `http://www.cs.bham.ac.uk/~zas/software/graphplanner.html`.

Savarimuthu, B. T. R. and Purvis, M. (2007). Mechanisms for norm emergence in multiagent societies. In Durfee, E., Yokoo, M., Huhns, M., and Shehory, O., editors, *Proceedings of the Sixth International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2007)*, pages 1104–1107.

Shoham, Y. and Tennenholtz, M. (1992a). Emergent conventions in multi-agent systems: Initial experimental results and observations (preliminary report). In Nebel, B., Rich, C., and Swartout, W. R., editors, *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning (KR 1992)*, pages 225–231.

Shoham, Y. and Tennenholtz, M. (1992b). On the synthesis of useful social laws for artificial agent societies. In Rosenbloom, P. and Szolovits, P., editors, *Proceedings* *of the Tenth National Conference on Artificial Intelligence (AAAI 1992)*, pages 276–281.

Shoham, Y. and Tennenholtz, M. (1995). On social laws for artificial agent societies: Off-line design. *Artificial Intelligence*, 73(1-2):231–252.

Somenzi, F. (2005). CUDD: CU Decision Diagram Package. `http://vlsi.colorado.edu/~fabio/CUDD/`.

Srivastava, S., Immerman, N., and Zilberstein, S. (2008). Learning generalized plans using abstract counting. In Cohn, A., editor, *Proceedings of the Twenty-Third National Conference on Artificial Intelligence (AAAI 2008)*, volume 2, pages 991–997.

Srivastava, S., Immerman, N., and Zilberstein, S. (2009). Challenges in finding generalized plans. In Fritz, C., McIlraith, S., Srivastava, S., and Zilberstein, S., editors, *Proceedings of the Workshop on Generalized Planning: Macros, Loops, Domain Control (ICAPS 2009)*, Thessaloniki, Greece.

Srivastava, S., Immerman, N., and Zilberstein, S. (2010). Computing applicability conditions for plans with loops. In Brafman, R., Geffner, H., Hoffmann, J., and Kautz, H., editors, *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling (ICAPS 2010)*, pages 161–168.

Stephan, W. and Biundo, S. (1996). Deduction-based refinement planning. In Drabble, B. and Tate, A., editors, *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems (AIPS 1996)*, pages 213–220.

Tuomela, R. and Bonnevier-Tuomela, M. (1995). Norms and agreement. *European Journal of Law, Philosophy and Computer Science*, 5:41–46.

Ullmann-Margalit, E. (1979). The emergence of norms. *Journal of Philosophy*, 54(209):575–587.

van der Hoek, W., Roberts, M., and Wooldridge, M. (2007). Social laws in alternating time: Effectiveness, feasibility, and synthesis. *Synthese*, 156(1):1–19.

Vázquez-Salceda, J. (2003). *The role of Norms and Electronic Institutions in Multi-Agent Systems applied to complex domains. The HARMONIA framework.* PhD thesis, Universitat Politécnica de Catalunya.

Vázquez-Salceda, J., Aldewereld, H., and Dignum, F. (2004). Norms in multiagent systems: Some implementation guidelines. In Ghidini, C., Giorgini, P., and van der Hoek, W., editors, *Proceedings of the Second European Workshop on Multiagent Systems (EUMAS 2004)*, pages 737–748.

Vázquez-Salceda, J. and Dignum, F. (2003). Modelling electronic organizations. In Mařík, V. and Pechouček, M., editors, *Proceedings of the Third Central and Eastern European Conference on Multi-Agent Systems*, pages 584–593.

Veloso, M. M. (1994). *Planning and Learning by Analogical Reasoning*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Verhagen, H. (2000). *Norm Autonomous Agents*. PhD thesis, Stockholm University.

von Wright, G. H. (1951). Deontic logic. *Mind*, 60(1–15).

Wagner, T., Lesser, V., Lesser, V., Decker, K., Decker, K., Carver, N., Carver, N., Garvey, A., Garvey, A., Neiman, D., Neiman, D., Prasad, M. N., and Prasad, M. N. (1998). Evolution of the GPGP domain-independent coordination framework. Technical report, University of Massachusetts.

Walker, A. and Wooldridge, M. (1995). Understanding the emergence of conventions in multi-agent systems. In Lesser, V. and Gasser, L., editors, *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS 1995)*, pages 384–389.

Weiß, G., editor (1999). *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, Cambridge, MA, USA.

Wooldridge, M. (2002). *An Introduction to MultiAgent Systems*. John Wiley and Sons.

Wooldridge, M. and Jennings, N. R. (1995). Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152.