

FORMALIZATION OF THE TRAFFIC
WORLD IN THE C ACTION LANGUAGE

A THESIS SUBMITTED TO
THE DEPARTMENT OF COMPUTER ENGINEERING
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Selim T. Erdoğan

July, 2000

QA
76-63
.E73
2000

FORMALIZATION OF THE TRAFFIC WORLD IN THE *C* ACTION LANGUAGE

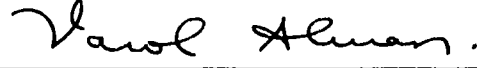
A THESIS SUBMITTED TO
THE DEPARTMENT OF COMPUTER ENGINEERING
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BİLKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Selim T. Erdoğan
July, 2000

SA
76-63
E73
2000

B 052997

I certify that I have read this thesis and that in my opinion it is fully adequate,
in scope and in quality, as a thesis for the degree of Master of Science.




Prof. Varol Akman (Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate,
in scope and in quality, as a thesis for the degree of Master of Science.




Asst. Prof. İlyas Çiçekli

I certify that I have read this thesis and that in my opinion it is fully adequate,
in scope and in quality, as a thesis for the degree of Master of Science.



Assoc. Prof. Nihan Kesim Çiçekli

Approved for the Institute of Engineering and Science:



Prof. Mehmet Baray
Director of the Institute

ABSTRACT

FORMALIZATION OF THE TRAFFIC WORLD IN THE \mathcal{C} ACTION LANGUAGE

Selim T. Erdođan

MS in Computer Engineering

Supervisor: Prof. Varol Akman

July, 2000

Reasoning about actions and effects of actions is an important task in Artificial Intelligence, with connections to knowledge representation and planning. Many formal methods for representing actions and inferring their effects have been developed over the years (e.g. action languages, fluent calculus, situation calculus). However, the examples formalized so far have been “toy” domains of very small sizes. Successful formalizations of scenarios of nontrivial size are needed in order to show that these methods are suitable for real applications and to assess the strong and weak sides of different methods. The \mathcal{C} action language is a logic programming language designed to represent the effects of actions on fluents. In this thesis we formalize the TRAFFIC scenario world — a domain of moderate size, specified at the Logic Modelling Workshop at <http://www.ida.liu.se/ext/etai/lmw/> — using the \mathcal{C} action language. Example planning problems using the formalization are successfully solved using the Causal Calculator — available at <http://www.cs.utexas.edu/users/tag/cc/> —, a program for planning and querying in action domains. The formalization is contrasted with previous work on the TRAFFIC world, namely the formalization of A. Henschel and M. Thielscher using the fluent calculus.

Keywords: Action languages, planning, causal reasoning, knowledge representation, \mathcal{C} , Causal Calculator, TRAFFIC, Logic Modelling Workshop.

ÖZET

TRAFFIC DÜNYASININ \mathcal{C} EYLEM DİLİNDE BİÇİMSELLEŞTİRİLMESİ

Selim T. Erdoğan

Bilgisayar Mühendisliği, Yüksek Lisans

Tez Yöneticisi: Prof. Dr. Varol Akman

Temmuz, 2000

Eylemler ve sonuçları hakkında akıl yürütme Yapay Zekâ açısından bilgi gösterimi ve planlama ile bağlantıları olan önemli bir iştir. Geçmiş yıllarda eylemlerin gösterimi ve sonuçlarının çıkarımı için pek çok biçimsel yöntem geliştirilmiştir (örneğin, eylem dilleri, akar hesabı, durum hesabı). Ne var ki, şimdiye kadar modellenmiş bütün örnekler çok küçük boyutlarda olan “oyuncak” dünyalardır. Bahsedilen yöntemlerin gerçek uygulamalar için uygun olduklarını göstermek ve değişik yöntemlerin güçlü ve zayıf yanlarını saptayabilmek için basit olmayan boyutlardaki senaryoların başarılı biçimselleştirmelerine gerek vardır. \mathcal{C} eylem dili eylemlerin akarlar üzerindeki etkilerini göstermek için tasarlanmış bir mantık programlama dilidir. Bu tezde TRAFFIC senaryo dünyası — <http://www.ida.liu.se/ext/etai/lmw/> adresindeki Logic Modelling Workshop tarafından tanımlanmış, orta büyüklükte bir dünya — \mathcal{C} eylem dili kullanılarak biçimselleştirilmektedir. Örnek planlama problemleri eylem dünyalarında planlamaya ve sorgulamaya yarayan Causal Calculator programını — bu program <http://www.cs.utexas.edu/users/tag/cc/> adresinden temin edilebilir — kullanarak başarılı bir biçimde çözülmektedir. Biçimselleştirme, TRAFFIC dünyasını biçimselleştirmek için daha önce A. Henschel ve M. Thielscher’in akar hesabı kullanarak yaptıkları çalışmayla karşılaştırılmaktadır.

Anahtar sözcükler: Eylem dilleri, planlama, nedensel akıl yürütme, bilgi gösterimi, \mathcal{C} , Causal Calculator, TRAFFIC, Logic Modelling Workshop.

Acknowledgements

I would like to express my gratitude to my supervisor Prof. Varol Akman for his guidance, useful suggestions, and invaluable encouragement throughout the development of this thesis. I am very fortunate to have had him as an advisor on academic and other matters.

The meetings I attended at Middle East Technical University in the early stages of this work were very helpful. I thank all who attended.

Lastly, I must say that the greatest support and help for all the good things that I have done so far, including this thesis, has come from my parents and my brother. Thanks from the depths of my heart.

To my family

Contents

| | | |
|----------|---|----------|
| 1 | Introduction | 1 |
| 1.1 | The Logic Modelling Workshop | 1 |
| 1.2 | Organization of the Thesis | 2 |
| 2 | Background Knowledge | 4 |
| 2.1 | Causal Theories | 4 |
| 2.1.1 | What Type of Causal Knowledge? | 5 |
| 2.1.2 | Syntax | 6 |
| 2.1.3 | Semantics | 7 |
| 2.1.4 | Definite Theories and Literal Completion | 7 |
| 2.2 | Action Language \mathcal{C} | 9 |
| 2.2.1 | About Action Languages | 9 |
| 2.2.2 | Syntax and Semantics of \mathcal{C} | 9 |
| 2.2.3 | Abbreviations | 10 |
| 2.2.4 | From \mathcal{C} Action Descriptions to Causal Theories | 13 |

| | | |
|----------|--|-----------|
| 2.3 | Satisfiability Planning | 14 |
| 2.3.1 | Satisfiability Planning with Causal Theories | 14 |
| 2.4 | The Causal Calculator | 15 |
| 3 | The Formalization of the TRAFFIC World | 16 |
| 3.1 | The TRAFFIC World | 16 |
| 3.1.1 | The Landscape | 16 |
| 3.1.2 | The Activities | 17 |
| 3.2 | Sorts | 17 |
| 3.2.1 | Constants | 18 |
| 3.2.2 | Variables | 18 |
| 3.3 | Fluents | 20 |
| 3.3.1 | Uniqueness | 22 |
| 3.3.2 | Using Macros Instead of Fluents | 23 |
| 3.4 | Actions | 24 |
| 3.4.1 | Effects of ENTER_SEGMENT | 25 |
| 3.4.2 | Execution Conditions for ENTER_SEGMENT | 26 |
| 3.5 | Automatic Speed Determination | 28 |
| 3.6 | Movement Along Segments and Arrival at Nodes | 31 |
| 3.6.1 | Effects of Being at a Node | 33 |
| 3.7 | Surrounding Traffic Restrictions | 34 |

| | |
|--|-----------|
| <i>CONTENTS</i> | x |
| 4 Example Planning Problems | 36 |
| 4.1 Constants | 37 |
| 4.2 Macros | 37 |
| 4.3 Problem 1: Getting from one node to another | 39 |
| 4.4 Problem 2: Getting from one node to another — concurrent change | 40 |
| 4.5 Problem 3: A blocked car | 42 |
| 5 Comparison with a Previous Formalization | 45 |
| 5.1 Similarities | 45 |
| 5.2 Differences | 46 |
| 5.3 Formalizing the Different Aspects in \mathcal{C} | 50 |
| 5.4 Fluent Calculus vs \mathcal{C} for the TRAFFIC World | 57 |
| 6 Conclusion | 59 |
| A The CCALC Program Listing | 61 |
| A.1 The World Description File | 61 |
| A.2 Scenario File for Problem 1 | 67 |
| A.3 Scenario File for Problem 2 | 69 |
| A.4 Scenario File for Problem 3 | 72 |
| B The TRAFFIC Scenario World | 75 |
| B.1 Introduction | 75 |

CONTENTS

xi

| | |
|---|----|
| B.2 Landscape Structure | 75 |
| B.3 Activities in the TRAFFIC world | 76 |

List of Figures

| | | |
|-----|--|----|
| 3.1 | The effects of ENTER_SEGMENT | 25 |
| 3.2 | Automatic speed determination when car1 is not blocked | 30 |
| 3.3 | Automatic speed determination when car1 is blocked | 31 |
| 3.4 | Movement of cars along segments | 32 |
| 3.5 | Arrival of cars at nodes | 33 |
| 4.1 | Landscape of the world in our planning problems | 38 |
| 4.2 | Initial state of problem 3 | 43 |
| 5.1 | Waiting areas for segments | 47 |
| 5.2 | Triggering the traffic light | 48 |

Chapter 1

Introduction

Reasoning about actions and effects of actions is an important task in Artificial Intelligence, with connections to knowledge representation and planning. Many formal methods for representing actions and inferring their effects have been developed over the years. However, the examples formalized so far have been “toy” domains of very small sizes. Successful formalizations of scenarios of nontrivial size are needed in order to show that these methods are suitable for real applications and to assess the strong and weak sides of different methods.

1.1 The Logic Modelling Workshop

The Logic Modelling Workshop (LMW) [23] is a part of the Reasoning about Actions and Change [32] research area¹. The Reasoning about Actions and Change area investigates formal methods, usually based on formal logic, for characterizing processes in terms of discrete-level state descriptions and events, actions, and activities which are capable of changing the current state [32]. The purpose of LMW is to provide an environment for researchers of action to communicate formalizations of scenario worlds of nontrivial size [23].

¹This is one of the research areas of the journal *Electronic Transactions on Artificial Intelligence* (ETAI) [8].

To this end, two scenario worlds have been specified at LMW: (i) ZOO, which is a world of a classical zoo with animals and cages, and (ii) TRAFFIC, a world of nodes and segments with cars traveling from node to node over the segments, obeying certain traffic rules.

The Logic Modelling Workshop challenges researchers to formalize the two scenario worlds in one of six well-known logics: action languages [11], cognitive robotics logic (CRL) [33], event calculus [31], fluent calculus [34], situation calculus (Toronto variant) [17], time and action logic (TAL) [7].

1.2 Organization of the Thesis

In this thesis we will formalize a discrete-time version of the TRAFFIC scenario world using the \mathcal{C} action language, an action language based on causal explanation. Our decision to choose the TRAFFIC world is based on the fact that it is closely related to important real life applications such as controlling real cars on real roads, and planning for automated driving. This implies that if the \mathcal{C} language can be successfully used to formalize the TRAFFIC world, it may later be extended to larger and more complicated worlds. We chose \mathcal{C} as the logic we would be using because of two reasons: it has many expressive possibilities (such as inertia, action preconditions, concurrency, nondeterminism), and there is a tool – the Causal Calculator [24] — for performing planning in domains expressed in \mathcal{C} .

In Chapter 2 we review the background knowledge that is needed to understand our work. This starts with a review of the causal theories of McCain and Turner [26], on which \mathcal{C} is based. The \mathcal{C} language and its abbreviations are reviewed after that. Satisfiability planning and how it may be done in domains expressed in \mathcal{C} are explained at the end of that chapter. The formalization of the TRAFFIC world is presented in Chapter 3. First we describe the TRAFFIC world and then all the aspects of the world are formalized. In Chapter 4 some planning problems and the solutions found are presented, demonstrating that our

formalization works. Chapter 5 contrasts our formalization with that of Henschel and Thielscher [13] which uses the fluent calculus. This is the only previous work on formalizing the TRAFFIC world. Similarities and differences of the formalizations are analyzed. We show that we can expand our formalization to include the aspects Henschel and Thielscher have formalized differently. Finally, we offer our conclusions and suggestions for future work in Chapter 6.

Chapter 2

Background Knowledge

We will formalize the TRAFFIC world using the \mathcal{C} action language [12] and then we will solve some example planning problems using the Causal Calculator [24]. \mathcal{C} is based on the theory of causal explanation and is closely related to the causal theories of McCain and Turner [26]. Satisfiability planning [14, 15] can be performed with \mathcal{C} using the Causal Calculator. In this chapter we first review causal theories and the \mathcal{C} language. After that, satisfiability planning and how the Causal Calculator can be used for action domains formalized in \mathcal{C} are explained.

2.1 Causal Theories

Causal theories of actions and change were introduced in 1997 by McCain and Turner [26]. Their previous work on a causal theory of ramifications and qualifications can be found in [25]. This review follows [26, 12].

2.1.1 What Type of Causal Knowledge?

Before we begin to describe causal theories we need to make more precise what kind of causal knowledge we will be dealing with.

There are two kinds of causal knowledge, one strong and one weak. In the strong case we know the causes of a fact. In the weak case, we do not know precisely what caused the fact but we know the conditions under which it is caused. We can express these two cases as follows:

- (i) The fact that ϕ causes the fact that ψ .
- (ii) Necessarily, if ϕ then the fact that ψ is caused.

Sentence (i) describes one of the causes of the fact ψ . Sentence (ii) only describes one of the conditions under which the fact ψ is caused. It should be noted that sentence (i) implies sentence (ii), but not vice versa. For example, the evacuation siren indicates that the explosion of the nuclear reactor in a few minutes is caused (type (ii)), but the cause of the explosion is not the siren (type (i)).

Having causal information about a certain domain enables an agent to rationally reason about that domain (this reasoning may be in the form of predictions, planning, or querying, just to name a few of the most common reasoning tasks). Given a set of facts Γ about the present and/or past which are known to be true, if there exists a causal sentence of type (i) or (ii) which shows that a fact ψ holds in the future, one may say that the fact ψ is part of a “causally possible world history¹.” If the fact could be shown to hold in every causally possible world history, then, given Γ we could safely predict ψ .

The causal theories that we will be discussing assume the “principle of universal causation.” This principle posits that the facts which hold in a causally possible world history are exactly the ones which are caused. In other words, every fact that is caused must hold, and every fact which holds must be caused.

¹In a causally possible world history, every fact that is caused holds.

Although this may look like a very strong ontological restriction, we show below how it may be bypassed in practice².

It is not difficult to see that when causally describing a domain in order to compute the causally possible world histories we may make use of information in the form of sentence (ii) just as well as information in the form of sentence (i). This allows us to have simpler and more homogenous representations. This idea is not something originally introduced by McCain and Turner. Previously, [9, 22] have also used knowledge of form (ii) in formalizing action domains.

2.1.2 Syntax

The language of causal theories is one of propositional³ logic. Its signature is given by a nonempty set of atoms. A *literal* is an atom (e.g. A) or a negated atom (e.g. $\neg A$). The expression *True* is a zero-place logical connective which is a tautology and *False* stands for $\neg True$.

A *causal law* is an expression of the form

$$\psi \leftarrow \phi \tag{2.1}$$

where ϕ and ψ are formulas⁴ of the propositional language⁵. The *antecedent* and *consequent* of (2.1) are ϕ and ψ , respectively.

Expression (2.1) is read as “Necessarily, if ϕ then the fact that ψ is caused.” This is causal knowledge of type (ii). As mentioned above, knowledge of type (i) can also be correctly (for our purposes) represented in this form.

A *causal theory* is a finite set of causal laws.

²We almost never hold all facts exempt from the principle of universal causation. Usually only the initial facts, and occasionally some others, benefit from this exemption.

³Only propositional causal theories will be dealt with in this thesis. For a reformulation of causal theories in predicate logic the reader is referred to [18].

⁴By a formula we mean any well-formed combination of atoms using the five logical connectives of propositional logic (\neg , \wedge , \vee , \supset , \equiv).

⁵An alternative syntax for (2.1), used in [26] is $\phi \Rightarrow \psi$.

2.1.3 Semantics

Let D be a causal theory and I be an interpretation⁶. For every D and I , let D^I be the following set

$$D^I = \{\psi : \text{for some } \phi, \psi \leftarrow \phi \in D \text{ and } I \models \phi\}.$$

D^I is the set of consequents of the causal laws in D whose antecedents are true in I . In other words, the formulas in D^I are exactly those which are caused to be true in I according to the causal theory D .

Main definition. Let D be a causal theory. An interpretation I is *causally explained* according to D if I is the unique model of D^I .

This can also be stated as follows. An interpretation I is causally explained according to D if and only if for every formula ϕ ,

$$I \models \phi \text{ iff } D^I \models \phi.$$

This means that I is causally explained according to D if and only if the formulas true in I are exactly those caused to be true according to D . This is equivalent to what we had stated earlier as the principle of universal causation. Everything which holds is caused and everything that is caused holds. Thus, each causally explained interpretation is a causally possible world history.

A formula ϕ is called a *consequence* of a causal theory D if ϕ is true in every interpretation which is causally explained according to D .

2.1.4 Definite Theories and Literal Completion

Let D be a causal theory in which

⁶By an interpretation I of a propositional language we mean a set of literals of the language such that this set includes exactly one literal for each atom.

- the consequent of every causal law $\psi \leftarrow \phi$ is either a literal or *False*,
- every literal is the consequent of finitely many causal laws.

Such a causal theory is said to be *definite*. Definite theories are important because there is a procedure to translate a definite causal theory into a classical propositional theory.

The *literal completion* of a definite causal theory is the classical propositional theory obtained by applying a modification of the Clark completion method [6]. The procedure is as follows:

- For each literal L in the language of D
 - Find all the causal laws $(L \leftarrow \phi_1, \dots, L \leftarrow \phi_n)$ having L as the consequent, and add the formula $L \equiv (\phi_1 \vee \dots \vee \phi_n)$,
 - If no causal law has the consequent L , add $L \equiv \text{False}$,
- For each causal law $\text{False} \leftarrow \phi$, add $\neg\phi$.

Proposition 2.1 *Let D be a definite causal theory. An interpretation I is causally explained according to D if and only if I is a model of the literal completion of D .*

The proof of this proposition can be found in [28, 27]. Now that we have a translation method from causal theories into classical propositional theories and we also know Proposition 2.1, the task of finding causally explained interpretations (and thus causally possible world histories) of a causal theory D is reduced to finding models of its literal completion.

2.2 Action Language \mathcal{C}

2.2.1 About Action Languages

Action languages are formal models of parts of the natural language that are used for talking about the effects of actions [11]. They are one of the many formal methods developed to represent actions and effects of actions.

The first action language, \mathcal{A} , was developed in 1993 [10]. Later on, more action languages were developed, the most recent one of which is the language \mathcal{C} [12]. \mathcal{C} is a language with many desirable properties. Nondeterminism, concurrent actions, action preconditions, inertial and noninertial fluents may all be conveniently expressed in \mathcal{C} .

In the following sections we review \mathcal{C} since it is the language which we will use to formalize the TRAFFIC world. Our review will follow [12, 11]. ([11] has general definitions of action languages.) More information about action languages may be found in [10, 3, 4, 35, 19, 21].

2.2.2 Syntax and Semantics of \mathcal{C}

There are two sets of propositional symbols in \mathcal{C} : σ^{fl} is the set of fluent names, and σ^{act} is the set of action names. The union of the sets σ^{fl} and σ^{act} is σ . An *action* is an interpretation of σ^{act} . What this means is that there are *elementary actions* which comprise the set σ^{act} (e.g. $\sigma^{act} = \{\text{load_the_gun}, \text{aim_gun_at_enemy}, \text{pull_the_trigger}, \text{smile_with_satisfaction}\}$) and an action a which specifies a group of elementary actions to be executed concurrently⁷ (e.g. an interpretation of σ^{act} which assigns the value *True* to `pull_the_trigger` and `smile_with_satisfaction` and the value *False* to `load_the_gun` and `aim_gun_at_enemy`).

⁷So we might imagine the set of actions as the power set of σ^{act} .

There are two types of *propositions* in \mathcal{C} :

$$\text{caused } F \text{ if } G, \quad (2.2)$$

$$\text{caused } F \text{ if } G \text{ after } H. \quad (2.3)$$

In the above propositions (and in the rest of this chapter) F and G are formulas of σ^{fl} and H is a formula of σ . Propositions of form (2.2) are *static laws* and propositions of form (2.3) are *dynamic laws*. In both kinds of propositions F is called the *head*.

An *action description* is a finite set of propositions.

Let D be an action description. A *state* is an interpretation of σ^{fl} which satisfies $G \supset F$ for every static law (2.2) in D . A *transition* is any triple $\langle s, a, s' \rangle$ where s, s' are states and a is an action; s is the *initial* state of the transition, and s' is its *resulting* state. A formula F is *caused* in a transition $\langle s, a, s' \rangle$ if it is

- the head of a static law (2.2) from D such that s' satisfies G , or
- the head of a dynamic law (2.3) from D such that s' satisfies G and $s \cup a$ satisfies H .

A transition is *causally explained* according to D if its resulting state s' is the only interpretation of σ^{fl} which satisfies all formulas caused in this transition.

2.2.3 Abbreviations

Although the only laws in \mathcal{C} are (2.2) and (2.3), many abbreviations of those two laws are used both to shorten action descriptions and to make them more comprehensible. We present the abbreviations which we will be using (in the abbreviations below U is a formula of σ^{act}).

- (i) An expression of the form

$$\text{caused } F \text{ after } H \quad (2.4)$$

stands for the dynamic law

caused F if $True$ after H .

(ii) An expression of the form

inertial F (2.5)

stands for a dynamic law of the form

caused F if F after F .

This means that if a formula holds in two consecutive states, the fact that it holds is caused by virtue of its persistence. Making formulas inertial solves the frame problem [30] for them.

(iii) An expression of the form

inertial F_1, F_2, \dots, F_n (2.6)

stands for the group of expressions

inertial F_1 ,

inertial F_2 ,

inertial F_n .

(iv) An expression of the form

U causes F if G (2.7)

stands for the dynamic law

caused F if $True$ after $U \wedge G$.

(v) An expression of the form

U causes F (2.8)

stands for the dynamic law

caused F if $True$ after U .

(vi) An expression of the form

$$\mathbf{nonexecutable } U \text{ if } F \quad (2.9)$$

stands for the dynamic law

$$\mathbf{caused } False \text{ if } True \text{ after } U \wedge F.$$

This states an “action precondition”. The action specified by U cannot be executed in states where F holds.

(vii) An expression of the form

$$\mathbf{default } F \quad (2.10)$$

stands for the static law

$$\mathbf{caused } F \text{ if } F.$$

Sometimes we would like to say that a fact holds whenever it has not been prevented by executing an action. For example, in a domain about tossing rocks in the air, we would make the fluent representing the rock being on the ground a default fluent (instead of inertial).

(viii) An expression of the form

$$\mathbf{never } F \quad (2.11)$$

stands for the static law

$$\mathbf{caused } False \text{ if } F.$$

(ix) An expression of the form

$$U \text{ may cause } F \text{ if } G \quad (2.12)$$

stands for the dynamic law

$$\mathbf{caused } F \text{ if } F \text{ after } U \wedge G.$$

This law enables us to specify nondeterministic effects of actions. For example, the outcome of a coin tossing action (U) may be formalized using two laws like (2.12) with different outcomes, one to cause tails (F) and one to cause heads ($\neg F$).

2.2.4 From \mathcal{C} Action Descriptions to Causal Theories

Let D be an action description. Then for any positive integer n , we may describe the translation $ct_n(D)$ into causal theories as follows. The signature σ_n of $ct_n(D)$ consists of $n + 1$ disjoint copies σ_i^{fl} ($0 \leq i \leq n$) of the fluent signature σ^{fl} and of n disjoint copies σ_i^{act} ($0 \leq i < n$) of the action signature σ^{act} . For any formula F of the original signature σ , F_i is the formula obtained by replacing every atom in F by the corresponding atom from σ_i^{fl} or σ_i^{act} .

Intuitively, the subscript i represents time. For a fluent symbol P , the atom P_i expresses that P holds at time i . For any action symbol A , the atom A_i expresses that the elementary action A is executed between times i and $i + 1$.

As the causal laws of the causal theory $ct_n(D)$, we have, for each static law of form (2.2) in D , the laws

$$F_i \leftarrow G_i \quad (0 \leq i \leq n),$$

and for all dynamic laws of form (2.3) in D , the laws

$$F_{i+1} \leftarrow G_{i+1} \wedge H_i \quad (0 \leq i < n).$$

Recall that all facts in a causally explained interpretation must be caused. This rule includes actions and initial states also so we must include laws for these to be caused. It is typically assumed that these facts are *exogenous* to the causal theory. Therefore, the following laws are added (where A , F , and t are meta-variables for action, fluent, and time names, respectively).

$$A_t \leftarrow A_t$$

$$\neg A_t \leftarrow \neg A_t$$

$$F_0 \leftarrow F_0$$

$$\neg F_0 \leftarrow \neg F_0$$

Now, given an action description, we have shown a translation to obtain the corresponding causal theory. In Section 2.1.4 we had shown how, given a definite

causal theory, we can use literal completion to translate it into a theory of classical propositional logic. What remains is how to apply this propositional theory to planning. This is the subject of the next section.

2.3 Satisfiability Planning

Approaching planning as a satisfiability problem was first proposed by Kautz and Selman [14]. There are indeed attractive properties of planning as satisfiability: Instead of just stating facts about the initial and goal states, arbitrary facts about any state of the world may be stated. Of course, this freedom is not limited to states. We may state constraints on actions, like having a certain action occur at a certain time.

As Kautz and Selman explain in [15], a plan corresponds to any model which satisfies a set of logical constraints that represent the initial state, the goal state and the domain axioms. Time consists of a fixed number of instances. Constraints specify conditions on fluents holding and actions being executed at particular times. When there are general constraints in the domain description, these are instantiated for all the time instances of the problem. This means that there is a limit for the maximum plan length. If the plan length is not known in advance, a binary search on instantiations with various time limits can find the solution.

2.3.1 Satisfiability Planning with Causal Theories

The idea to apply satisfiability planning to causal theories comes from (not surprisingly) McCain and Turner [28]. They define a special type of causal theories: causal action theories. The form of a causal action theory is exactly like the translation $ct_n(D)$ obtained from an action description D .

An *initial state description* S_0 is a set of fluent literals which refer to time 0, including exactly one literal for each atom in σ_0^{fl} . A *time-specific goal* G is a fluent formula. A *plan* P is a set of actions such that it includes at most one

literal of each atom in σ_i^{act} ($0 \leq i < n$).

In a domain description D , a plan P is a *causally possible* plan for achieving G in S_0 if $S_0 \cup P \not\vdash_D \neg G$. A *deterministic* plan is one in which for every time t , the history up to and including time t together with the actions performed at time t entail the state of the world at time $t + 1$. It is proven in [28] that every causally possible and deterministic plan is *valid* (i.e. the actions in P are sufficient for achieving G in S_0 and furthermore, we can always execute the plan).

2.4 The Causal Calculator

The Causal Calculator (CCALC) [24] is a model checker for the language of causal theories which runs in Prolog. Propositions in the action language \mathcal{C} may be translated into the language of causal theories by using rewrite rules. This way, planning and querying may be done for action descriptions in \mathcal{C} .

Given a file with a domain description in \mathcal{C} , CCALC first translates this into schemas in the language of causal theories. These schemas are then instantiated with respect to the language signature and translated into propositional logic to obtain a set of clauses. When a planning problem is given along with a set of times (the integers from 0 to the time of the latest goal) the set of clauses is used to obtain another set of clauses whose models correspond to causally possible world histories over the time range. Finally, this set of clauses is combined with clauses describing the initial state and clauses describing the desired goals, and a satisfiability checker⁸ is called to search for a model of all the clauses. If a model is found, the action literals in it correspond to the plan.

The plan found will be causally possible, but not necessarily valid. CCALC also gives the user the option to verify the plans found (i.e. check whether the initial state was completely specified and whether the plan is deterministic). If the plan is verified, it is valid.

⁸Two satisfiability checkers can be used with CCALC: *relnsat* [5] or *sato* [36].

Chapter 3

The Formalization of the TRAFFIC World

3.1 The TRAFFIC World

The TRAFFIC World is described at the Logic Modelling Workshop [23]. The goal is to represent vehicles traveling from town to town (or from intersection to intersection, depending on the interpretation) along roads, with the vehicles respecting speed limits and other cars they encounter on the road. The specification consists of the landscape and the activities¹.

3.1.1 The Landscape

There are two types in TRAFFIC: *nodes* and *segments*. These may be thought of as road crossings (or towns) and roads which connect the nodes, respectively. There is a start node, an end node, and a length for each segment. All the nodes and segments are fully known and can be assigned unique names.

¹A verbatim copy of the full specification from the Logic Modelling Workshop can be seen in Appendix B.

3.1.2 The Activities

The only activities in TRAFFIC are done by *cars*. Cars drive along the segments from node to node.

At each point in time, each car has a position which is composed of two parts: The segment on which the car is traveling and the distance traveled along the segment. However, cars can travel in both directions along segments so (although it is not specified at LMW) there needs to be a third component of the position, namely the direction in which the car is traveling.

Cars traveling in opposite directions along the same segment may pass by each other without any problems.

Overtaking is not allowed. When a car driving along a segment reaches another car traveling in the same direction, it must stay behind the car until they reach the next node. This is called the *surrounding traffic restriction*. There is a fixed safety distance *varsigma*. The car behind is never allowed to be closer than *varsigma* to the car in front, so its speed must be adjusted when it gets close to the car in front.

There is a top speed for each car and a speed limit for each segment. At each point in time, the speed of the car is the maximum allowed by its own top speed, the speed limit of the segment on which it is traveling, and the surrounding traffic restriction.

3.2 Sorts

We use six sorts in the formalization:

car, node, segment, distance, speed, direction

The first three sorts are for representing the vehicles, nodes, and segments, respectively. The sort *distance* is for representing the distances between nodes (i.e. the lengths of segments) and the distances which vehicles have traveled along segments. The speeds of the vehicles are represented by *speed*. An alternative formalization could use a single sort *number* in place of these two sorts. We chose to use two sorts because it increases the efficiency of planning since there are fewer rules when the generic rules are grounded (this will be explained in the section about variables). The last sort *direction* is for representing the direction in which the vehicle is traveling on a segment².

3.2.1 Constants

In order to describe domains and problems, we need to specify the constants of each sort. These will be the node and segment names, the car names, the numbers we use for the speeds and distances, and the direction names.

All of these constants, except for the direction names, are problem-dependent and are specified with the problems. Only the direction constants, listed below, are common to all problems:

backward, forward : *direction*

3.2.2 Variables

When stating *general* static and dynamic laws, variables of the appropriate sort are used in fluents instead of the actual constants we have in our scenario world. This saves us from writing the same rule many times only changing the arguments of the fluents or actions.

²Another alternative formalization may choose to omit this sort and allow negative distances. However, there doesn't seem to be any advantage gained by this and it doesn't make the formalization any clearer. In fact, the number of ground rules would increase, making planning much less efficient.

We have the following variables for each sort:

| | |
|-------------------|--------------------|
| C, C1, C2 | : <i>car</i> |
| N, N1 | : <i>node</i> |
| S, S1 | : <i>segment</i> |
| D, D1, D2, D3 | : <i>distance</i> |
| Sp, Sp1, Sp2, Sp3 | : <i>speed</i> |
| Dir, Dir1 | : <i>direction</i> |

At the time of grounding, the variables in the general fluents and the general laws are made ground using all the constants of the matching sorts. For example if we had a fluent $\text{foo}(C)$, and we had the constants bar1 and bar2 , then we would have the atoms $\text{foo}(\text{bar1})$ and $\text{foo}(\text{bar2})$ in our actual action description. The same kind of replacement happens with the variables in the laws.

This is why we have the two sorts *distance* and *speed* instead of *number*. If we had a single sort *number*, the constants of this sort would be all the integers from 0 to the largest number in the domain, which is either the largest speed or the largest distance. All the fluents which had *number* as their argument (the fluents which take a *number* argument to represent speed or distance) would be instantiated for all these integers. In contrast, when we have the two sorts *distance* and *speed* then we have two sets of integers from 0 to the largest integer of the sort. Usually, we would expect the largest speed to be much less than the largest distance. So when we ground the fluents, we get fewer instantiations of fluents having an argument of sort *speed*.

We also have some variables of a special sort *computed* which are used to increase the efficiency.

| | |
|---------------|-------------------|
| X, X1, X2, X3 | : <i>computed</i> |
|---------------|-------------------|

Unlike with the variables of other sorts, CCALC does not need to select values for *computed* variables during grounding since they are pre-computed by looking at the grounded values of the other arguments of the fluent or action in which

they appear. So instead of having all the versions of a causal law for all the possible combinations of the values the arguments of the fluents and actions may take, we have only the versions in which the arguments not of *computed* sort take all the possible combinations.

3.3 Fluents

The LMW specification says that at any point in time cars have a position composed of two components: the segment on which the car is traveling, and the distance the car has traveled along that segment. In addition to these components, we have also included a third component for times when a car is at a node. In this case, instead of simultaneously being on all the segments which meet at that node (at a distance of 0), the car is simply at the node³. The three fluents for the position of the car are:

```
on_segment(car,segment)
at_distance(car,distance)
at_node(car,node)
```

Of these three fluents, we want `on_segment` and `at_node` to be inertial when they are true. This means that once a car is at a node, it can stay at that node and once it is on a segment it will stay on that segment until it gets off. However, `at_distance` does not need to be inertial since a unique positive literal which corresponds to the distance is always caused to hold by one of the laws which we will introduce shortly. So

```
inertial on_segment(C,S).
inertial at_node(C,N).
```

³This is what we would naturally expect if we think of nodes as towns or some other kind of area where the cars may stop for any amount of time. In the case where nodes are intersections with traffic lights, it is not that intuitive.

We could have chosen to have one fluent `position(car,segment,distance)` instead of the two fluents `on_segment` and `at_distance`. We preferred to have two because it makes planning more efficient. Imagine that there are n constants of each sort. If we had chosen the fluent with three arguments, then the ground version of our fluent would have n^3 atoms but since we have the two fluents declared above we get $2n^2$ atoms.

Cars also have a velocity at each point in time. This means that they have a certain speed and they are traveling in a certain direction along the segment on which they are. There are two fluents for representing the velocity:

```
at_speed(car,speed)
in_direction(car,direction)
```

The speed is determined at all times by the causal laws but the direction isn't. Therefore `in_direction` should be made inertial when true so that the direction stays the same as long as the car is traveling on the segment.

```
inertial in_direction(C,Dir).
```

The LMW specification forbids cars to “make a U-turn” at a node and go back on the segment they just traveled on. To solve this problem we have a fluent which is true if and only if a car has just traveled on a certain segment:

```
last(car,segment)
```

The fluent `last` needs to be inertial for both the true and the false cases. This is because of cases where we might have a car at a node in the initial state and that car may stay there for a while. We would not want to restrict this car from traveling on any segments once it decides to travel so both `¬last` and `last` would need to be caused.

```
inertial last(C,S), ¬last(C,S).
```

Sometimes a car will be blocked by another car in front of it traveling in the same direction and it will modify its speed in order to meet safety measures. Hence, a fluent notifying whether a car is blocked is needed:

`blocked(car)`

When a car is on a segment it will most likely not be blocked. We only have a law causing `blocked`, so we should have a way to have `¬blocked` caused if the car is not blocked. The best way to do this is to make it a default false fluent:

`default ¬blocked(C).`

3.3.1 Uniqueness

A car cannot be at more than one node at a given time. Likewise, it cannot be on more than one segment at a time either. In fact, if one looks at all the fluents above, one can see that a car cannot have more than one distance, speed, direction, or segment on which it last traveled. Therefore, there must be laws which cause the fluents with different arguments to be false when fluents with certain values hold. This is formalized in the following static laws:⁴

`caused ¬at_node(C,N) if at_node(C,N1) && ¬(N=N1).`
`caused ¬on_segment(C,S) if on_segment(C,S1) && ¬(S=S1.)`
`caused ¬at_distance(C,D) if at_distance(C,D1) && ¬(D=D1).`
`caused ¬at_speed(C,Sp) if at_speed(C,Sp1) && ¬(Sp=Sp1).`
`caused ¬in_direction(C,Dir) if in_direction(C,Dir1) && ¬(Dir=Dir1).`
`caused ¬last(C,S) if last(C,S1) && ¬(S=S1).`

⁴We use '&&' to denote logical conjunction.

3.3.2 Using Macros Instead of Fluents

Some relations between sorts stay the same throughout our plans. For example, the lengths of segments and the start node and end node of each segment do not change. Likewise, the speed limit of a segment and the maximum speed of a car do not change either.

Declaring these as fluents would not only add a lot of fluents to our domain but it would also force us to have a very large number of uniqueness laws and this would decrease the efficiency considerably. (The number of laws about the speed limits and the maximum speeds would be especially large due to the large number of constants — all the integers from 0 to the largest integer.)

The Causal Calculator allows users to declare rewrite rules called *macros*. Using these, we can have a certain expression transformed into a Prolog expression. So, instead of declaring as fluents, we use macros for the following:

```
startnode(segment,node)
endnode(segment,node)
length(segment,distance)
speed_limit(segment,speed)
max_car_speed(car,speed)
```

These macros are used just like fluents in static or dynamic laws. What each macro stands for is dependent on the problem. During execution, if CCALC encounters one of the above macros it replaces it with the corresponding substitution. (Example macro declarations can be seen in Section 4.2.)

In addition to the problem-dependent macros, we have macros which we use for representing mathematical functions:

```
sum(#1,#2,#3)           - > #1 is min((#2)+(#3), maxInteger)
absdiff(#1,#2,#3)       - > #1 is abs((#2)-(#3))
diff(#1,#2,#3)          - > #1 is max((#2)-(#3), 0)
```

`minimum(#1,#2,#3)` \rightarrow #1 is $\min(\#2,\#3)$

These are also used just like fluents. They hold when the expression on the right side of the declaration holds. The `sum` and `diff` macros are for addition and subtraction in the $0..\text{maxInteger}$ range⁵. `maxInteger` is the largest integer in the program (usually the length of the longest segment). `absdiff` is for finding the absolute value of the difference and `minimum` is for finding the smaller of two numbers⁶. The arguments of these macros are always *speed* and *distance* variables. (They are the only numerical variables.)

As a final comment on macros, we may say that (like variables) everything we do with them could be done without them but would be much more difficult.

3.4 Actions

There is only one action⁷:

`ENTER_SEGMENT(car,segment)`

This action causes a car at a node to get onto a segment. Once a car is on a segment, the speed is uniquely determined by the road conditions so there is no need for an action to change the speed.

⁵The `sum` and `diff` macros and the idea of representing mathematical functions with macros were taken from [20].

⁶Actually, these macros are not necessarily for *finding* the said values. Only when the argument which is “found” is a variable of computed sort can they be said to find since they limit the number of ground rules. Their real effect in the causal laws in which they appear is just like fluents.

⁷From now on, when we say “action”, we mean an elementary action (i.e. a single element of σ^{act}).

3.4.1 Effects of ENTER_SEGMENT

Executing the ENTER_SEGMENT action causes a car to be positioned on the segment at the next instant and the distance traveled is caused to be zero, independent of the node from which the car enters the segment.

ENTER_SEGMENT(C,S) causes on_segment(C,S).

ENTER_SEGMENT(C,S) causes at_distance(C,0).

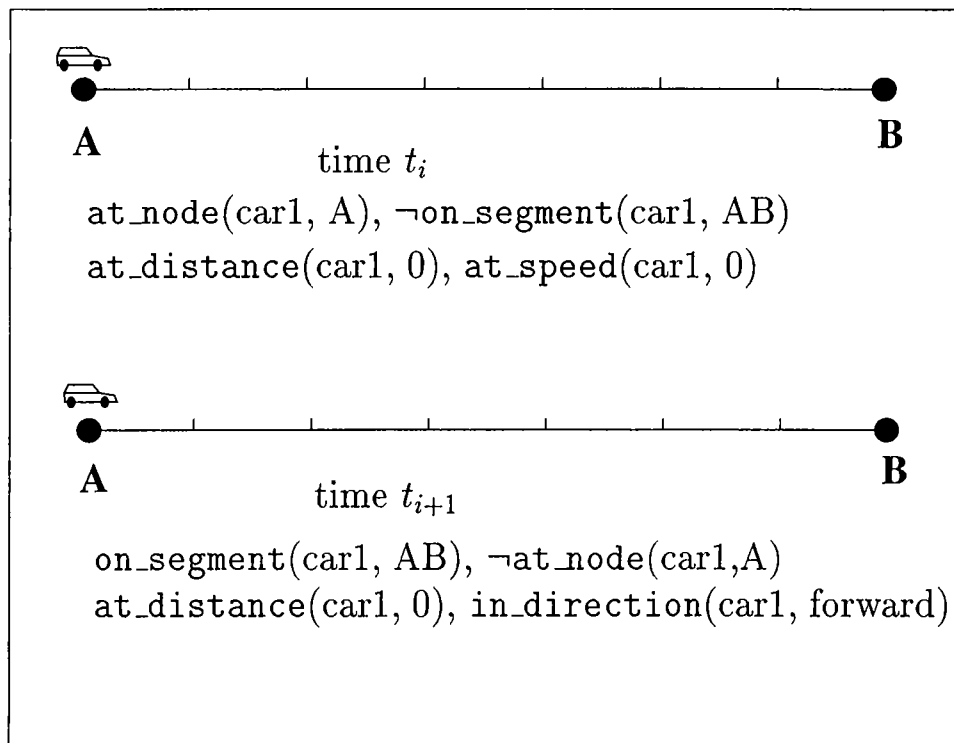


Figure 3.1: The effects of ENTER_SEGMENT

The direction a car is going in depends on the node from which the car enters the segment. If it is at the start node of the segment when the action is executed, it is caused to be going in the forward direction. If it is at the end node of the segment when the action is executed, it is caused to be going in the backward direction.

```

ENTER_SEGMENT(C,S) causes in_direction(C,forward)
                        if startnode(S,N) && at_node(C,N).
ENTER_SEGMENT(C,S) causes in_direction(C,backward)
                        if endnode(S,N) && at_node(C,N).

```

Once a car enters a segment, it is no longer at a node. We could formalize this with a dynamic rule involving the action `ENTER_SEGMENT`. However, we might like to have problem descriptions with cars already on segments (instead of all cars being at nodes initially). To allow this we include a static law in which the car is caused not to be at any node if it is on a segment.

```

caused ¬at_node(C,N) if on_segment(C,S).

```

The same thing goes for the fluent `last`. A static law states that a segment is caused to be the segment on which the car has most recently traveled if the car is on that segment.

```

caused last(C,S) if on_segment(C,S).

```

Note that we could represent the two cases above as effects of the action `ENTER_SEGMENT` if we state the value of all the fluents in the initial state. Our formalization makes it easier to state problems. It seems more natural to say that a car is not at a node because it is on a segment. In relation to the `ENTER_SEGMENT` action, these effects are represented as indirect effects, i.e. ramifications.

3.4.2 Execution Conditions for `ENTER_SEGMENT`

When describing an action domain usually there are some action preconditions which must be satisfied before the related actions can take place. Therefore, we must include in our domain description information about what these preconditions are.

A car can only enter a segment if it is at a node. This means that the action `ENTER_SEGMENT` cannot be executed if the car is not at any node⁸.

nonexecutable `ENTER_SEGMENT(C,S)` **if** $\neg(\ \backslash/N: \text{at_node}(C,N))$.

The law above said that a car must be at a node to be able to execute the `ENTER_SEGMENT` action, but this is not enough. The node at which it is must be the start node or the end node of the segment which it is trying to enter. So a car cannot enter a segment which does not have that node as its start or end node:⁹

nonexecutable `ENTER_SEGMENT(C,S)`
if `at_node(C,N) && $\neg(\ \text{startnode}(S,N) ++ \text{endnode}(S,N))$` .

Looking at the two laws above one may, at first sight, think that including only the second one in our domain description would suffice and that the first one is redundant. This is not true. If we carefully inspect this second law, we see that if the car is not at any node then the first part of the conjunction (`at_node(C,N)`) fails and there is nothing preventing the execution of `ENTER_SEGMENT`. We include the first law to prevent the execution of the action in such cases.

Cars at nodes are not allowed to go back on the segments on which they came to that node. The law that a car cannot enter a segment on which it most recently drove is formalized as follows:

nonexecutable `ENTER_SEGMENT(C,S)` **if** `last(C,S)`.

⁸We use $\backslash/$ in front of a variable to mean the logical disjunctions of the following expression replaced with constants of the same sort as the variable. It is similar to the existential quantifier in First Order Logic.

⁹We use ‘++’ to denote logical disjunction.

3.5 Automatic Speed Determination

The speed of a car at each point in time is uniquely determined as the maximum speed it can drive at without violating the safety measures. The safety measures say that no car may get closer than a fixed distance *varsigma* to the car traveling in front of it.

The maximum speed that the car may travel at when there is no car in front restricting its speed is the smaller of two limits: the speed limit of the road on which it is traveling and its own top speed. The maximum speed that the car may travel at when there is a car blocking it is the speed which will bring it to a position exactly *varsigma* behind the car in front. So we see that there are two distinct cases for which we must design causal laws to have the speed of the car caused. What we need now is a way to know when which causal law should be applied to cause the speed.

This is where the fluent `blocked` comes into play. If we can design a causal law to cause `blocked` to be true at times when we need to adjust the speed to be just *varsigma* behind the car in front, and have `blocked` be caused to be false when the road is free to go at the smaller of the two limits, then the case is solved.

Fortunately there is a quite straightforward way to do this. Ordinarily we would expect that a car goes at the smaller of its own top speed and the speed limit of the segment. Since we know the current position and the smaller of these two limits we can imagine for a moment where the car would be if it indeed went at this speed. We also know the positions and the current speeds of the cars in front of it on the same segment. So we can check whether it will be closer than *varsigma* to (or even ahead of) any of the cars in front. This way we can detect when the surrounding traffic restriction would be violated if the car doesn't adjust its speed as if it were blocked, and we can prevent such situations. This is formalized in the law below which causes the fluent `blocked` to be true for a car when there is another car traveling in front of it and it will get closer than *varsigma* to that car (or possibly even pass it) if it goes at the smaller of its two

speed limits:

```

caused blocked(C)
  if on_segment(C,S) && on_segment(C1,S) && ¬(C=C1)
    && in_direction(C,Dir) && in_direction(C1,Dir)
    && at_distance(C,D) && at_distance(C1,D1) && (D1>D)
    && speed_limit(S,Sp1) && max_car_speed(C,Sp2)
    && minimum(X,Sp1,Sp2) && sum(X1,X,D)
    && at_speed(C1,Sp3) && sum(X2,Sp3,D1)
    && diff(X3,X2,varsigma) && (X1 > X3).

```

An example where this law causes `blocked` to be true is shown in Figure 3.3.

Recall that the fluent `blocked` was caused to be false by default in Section 3.3. This means that when it is not caused to be true by the law above it is caused to be false. This is what we would like to have in the usual case. If there is no other car in front blocking the way, then, since `blocked` is false, the car goes at the smaller of its two speed limits (the speed limit of the road and its own maximum speed):

```

caused at_speed(C,X)
  if ¬blocked(C) && on_segment(C,S) && speed_limit(S,Sp1)
    && max_car_speed(C,Sp2) && minimum(X,Sp1,Sp2).

```

An example where this law causes the speed of a car which is not blocked is shown in Figure 3.2.

When a car is blocked, it has to adjust its speed so that it will be traveling at the fastest possible speed without getting closer than *varsigma* to the car in front of it. Clearly, this speed will be the speed that brings it to a distance of exactly *varsigma* behind the car it is following. Writing a causal law which causes the speed of the car to be adjusted according to the car in front of it may seem to be a good solution. However, there is a subtlety here. Among the cars which are on

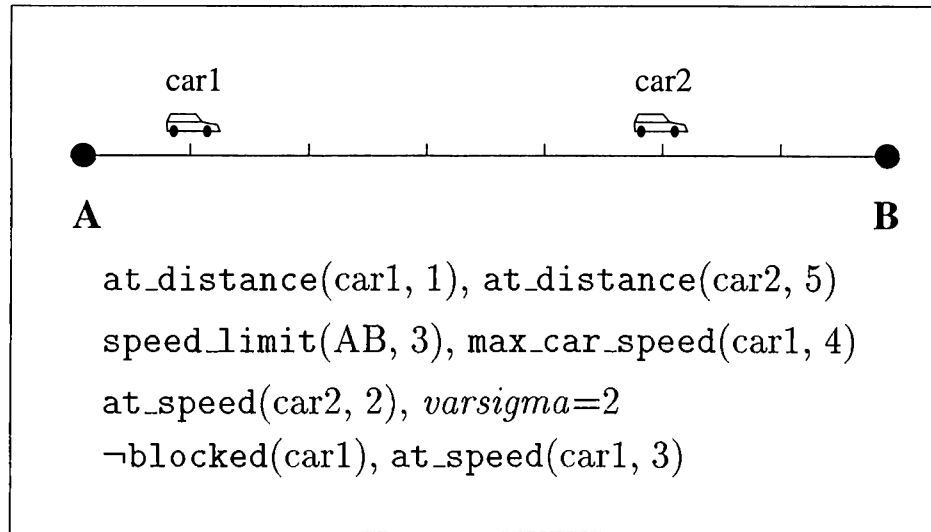


Figure 3.2: Automatic speed determination when car1 is not blocked

the same segment with the car for which we are trying to determine the speed, there may be more than one which is in front of it. Such a causal law would cause more than one speed which would be nonsense and of course would lead to no plans being found. So we must make sure that the speed is adjusted according to the *closest* car in front. This can be done by making the causal law so that the speed is adjusted according to a particular “blocking” car in front if (i) there are no cars on the segment and going in the same direction other than the blocked and the “blocking” car, or (ii) the other cars on the segment which are going in the same direction are at distances which are greater than the distance at which the “blocking” car is. The adjustment causes the speed to be such a value that at the next time instant the car will be exactly *varsigma* behind the point at which the blocking car will be. This is true even when the car in front reaches a node (in this case the speed will take the car to a distance of *varsigma* away from the node). We formalize the adjustment law as follows:¹⁰

caused at_speed(C,Sp1)

¹⁰We use \wedge in front of a variable to mean the logical conjunctions of the following expression replaced with constants of the same sort as the variable. It is similar to the universal quantifier in First Order Logic.

```

if blocked(C) && on_segment(C,S) && on_segment(C1,S)
&& ¬(C=C1) && in_direction(C,Dir) && in_direction(C1,Dir)
&& at_distance(C,D) && at_distance(C1,D1) && (D1>D)
&& at_speed(C1,Sp) &&
(\/C2: ( ¬on_segment(C2,S) ++ (C2=C) ++ (C2=C1)
++ ¬in_direction(C2,Dir)
++(at_distance(C2,D2) && (D2 > D1)) ) )
&& diff(X1,D1,D) && sum(X2,X1,Sp)
&& diff(Sp1,X2,varsigma).

```

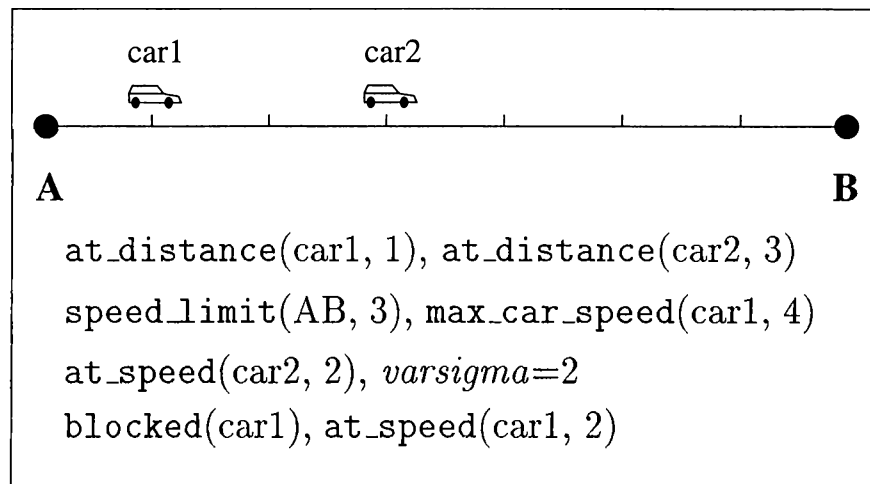


Figure 3.3: Automatic speed determination when car1 is blocked

3.6 Movement Along Segments and Arrival at Nodes

Movement in the TRAFFIC world is accomplished without actions. After executing the ENTER_SEGMENT action the car is on the segment and its speed is determined automatically by the laws in the previous section so everything needed for movement is ready. As would be expected, a car is caused to be at a distance equal to the sum of its speed and the current distance it has already traveled

(Figure 3.4). Of course, if this distance is greater than the length of the segment it is traveling on, it will be reaching the next node so the law should not cause the fluent `at_distance` in such cases.

caused `at_distance(C,X)`
 after `at_distance(C,D1) && at_speed(C,Sp) && sum(X,Sp,D1)`
`&& on_segment(C,S) && length(S,D2) && (X<D2)`.

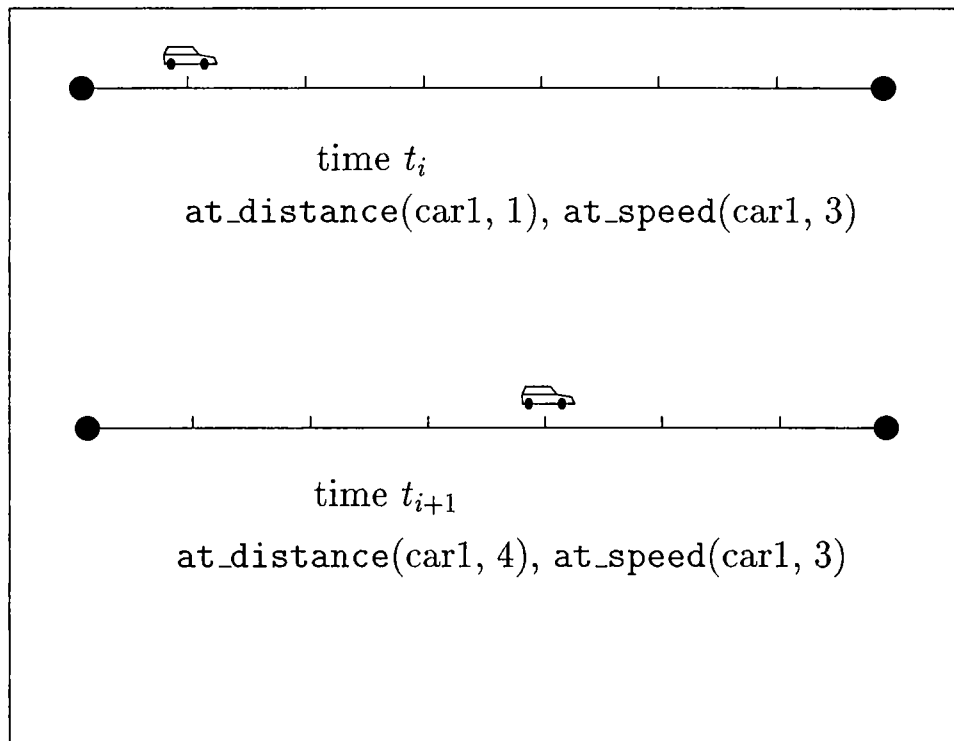


Figure 3.4: Movement of cars along segments

If the point to which the car would advance with its current speed is equal to or more than the length of the segment, this means that the car should arrive at the next node at the following time instant (Figure 3.5). The fluent `at_node` is caused to be true by this dynamic law:

caused `at_node(C,N)`
 after `at_distance(C,D1) && at_speed(C,Sp)`

```

&& sum(X,Sp,D1) && on_segment(C,S)
&& length(S,D2) && (X>=D2)
&& ( (in_direction(C,forward) && endnode(S,N))
    ++ (in_direction(C,backward) && startnode(S,N)) ).

```

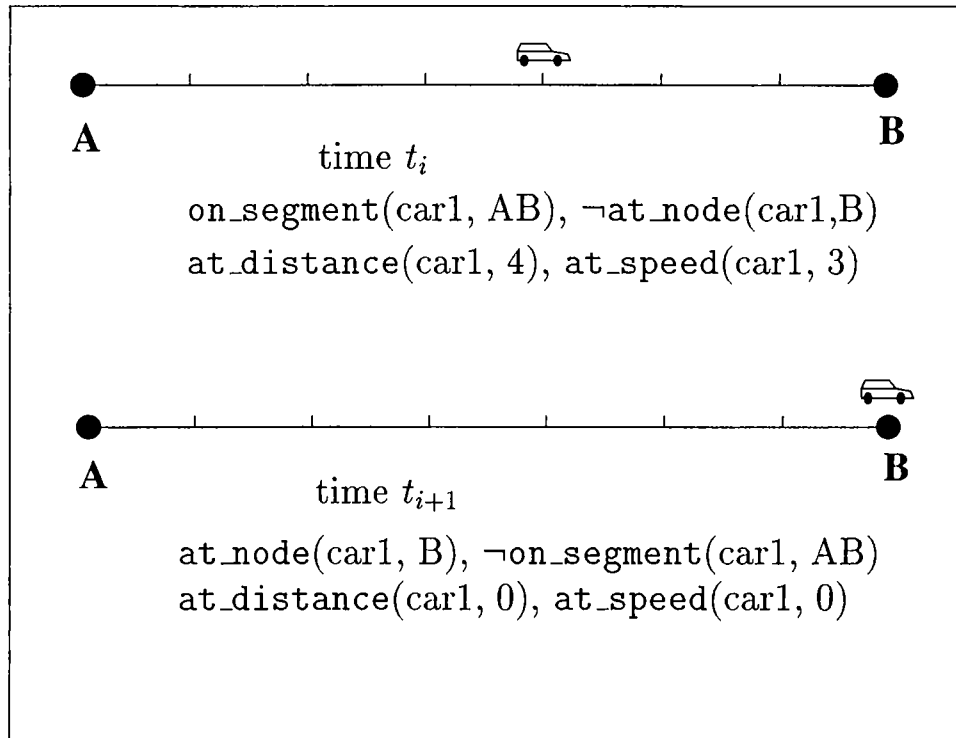


Figure 3.5: Arrival of cars at nodes

3.6.1 Effects of Being at a Node

A car which arrives at a node is not on any segment anymore. Of course, not being on any segment, there is no direction in which it is traveling either. A car which is at a node is caused to be at a distance of 0. We could cause it to be at no distance at all (by causing the fluents with all the values of the distances to be false) but that wouldn't make any difference. The speed is also caused to be 0.

Finally, a car at a node is clearly not blocked¹¹. All these points are formalized by the following static laws:

caused \neg on_segment(C,S) if at_node(C,N).
 caused at_distance(C,0) if at_node(C,N).
 caused \neg in_direction(C,Dir) if at_node(C,N).
 caused at_speed(C,0) if at_node(C,N).
 caused \neg blocked(C) if at_node(C,N).

3.7 Surrounding Traffic Restrictions

The surrounding traffic restrictions state that (i) two cars traveling in the same direction on the same segment cannot be closer than *varsigma* and that (ii) cars cannot pass other cars in front of them.

We have already guaranteed by the laws in Section 3.5 that cars already on segments will adjust their speeds to obey the restrictions. However, we must also consider two other cases: in the first one there are two cars at the same node and they want to enter the same segment. With the causal laws listed so far they will both be allowed to execute the ENTER_SEGMENT action and enter the segment, which will cause them both to be at the same distance. (Of course the same goes for more than two cars also.) In the second case, there is a car at a node and there is another car which is already on the segment which the first car wants to enter. If the second car hasn't reached a distance greater than *varsigma* by the time the first car has completed the execution of ENTER_SEGMENT, the restrictions will be violated. These two cases need to be prevented. This is easily taken care of by the following law which forbids two cars which are on the same segment and traveling in the same direction to be closer than *varsigma*:

never on_segment(C,S) && on_segment(C1,S) && \neg (C=C1)

¹¹We could easily omit this last law if we specify in our problem descriptions whether cars at nodes in the initial state are blocked or not.

$$\begin{aligned} &\& \text{in_direction}(C, \text{Dir}) \ \& \text{in_direction}(C1, \text{Dir}) \\ &\& \text{at_distance}(C, D) \ \& \text{at_distance}(C1, D1) \\ &\& \text{absdiff}(X, D, D1) \ \& (X < \text{varsigma}). \end{aligned}$$

Overtaking is prevented by the laws which automatically determine the speed. A car on a segment can never have a speed which would take it to a point farther than *varsigma* behind the car in front of it. Cars cannot violate the overtaking restriction when they are entering segments either, since the action ENTER_SEGMENT causes all cars to be at distance 0 when they first enter the segment.

Having in our domain description both the law above and the law which states that ENTER_SEGMENT causes cars to be at a distance of 0 when they enter a segment puts an implicit condition on actions: two cars cannot enter the same segment at the same time.

Chapter 4

Example Planning Problems

The formalization of the TRAFFIC world presented in the previous chapter has been implemented as a domain description file for the Causal Calculator¹. Example planning problems were devised to illustrate that the causal laws indeed work as desired. In this chapter, we explain each planning problem and the plans found. The planning problems and the solutions found are presented in the original CCALC format, which is easy to understand.

All of the problems were tested using CCALC version 1.23 which ran in SICStus Prolog version 3 #5. The satisfiability checker was *relnsat* [5]. This setup was used on a Sun UltraSPARC-II running SunOS 5.6. Although the computer had 1GB of memory, the version of SICStus we were running had a limit of 64MB for the memory it can use. Due to this limitation, we were not able to run large planning problems.

¹The CCALC program listing for the world and problem descriptions can be seen in Appendix A.

4.1 Constants

The causal laws in the action description were written using variables. To get the actual causal laws which govern our problems we need to state the constants of each sort over which the variables will be instantiated.

Most of the constants in our planning problems are common. These are

```

car1, car2      :car
a, b, c        :node
road_ab, road_bc :segment
0..maxDistance :distance
0..maxSpeed     :speed
backward,forward :direction

```

4.2 Macros

In Chapter 3, we mentioned that there are some relations between sorts that are constant in the domain, like lengths of segments, top speeds of cars, and so on. In the interest of efficiency we have represented these constant relations as macros instead of as fluents. Here we list the macros we use in most of the planning problems we will examine.

```

startnode(#1,#2)      - > ( (#1=road_ab && #2=a)
                        ++(#1=road_bc && #2=b))
endnode(#1,#2)       - > ( (#1=road_ab && #2=b)
                        ++(#1=road_bc && #2=c))
length(#1,#2)        - > ( (#1=road_ab && #2=10)
                        ++(#1=road_bc && #2=10))
speed_limit(#1,#2)   - > ( (#1=road_ab && #2=4)
                        ++(#1=road_bc && #2=4))
max_car_speed(#1,#2) - > ( (#1=car1 && #2=2)

```

```
++(#1=car2 && #2=4))
```

The first four of the above macros define the landscape of the world (Figure 4.1). The last one defines the top speeds of the cars.

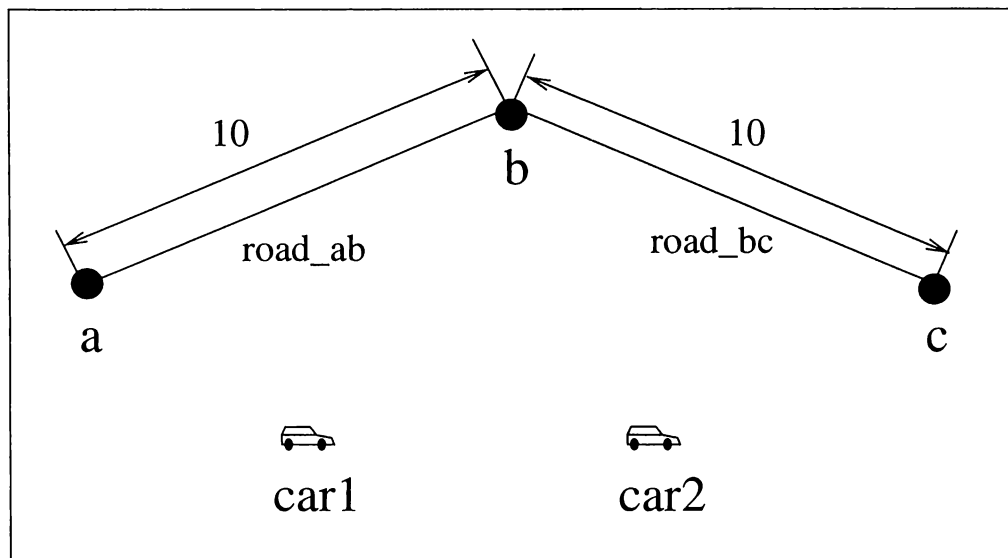


Figure 4.1: Landscape of the world in our planning problems

The next group of macros is used in defining other macros or for setting limits on integer constants. `maxSpeed` and `maxDistance` are the largest values speed and distance can take in our problems. `maxInteger` is the largest integer we may encounter in the arithmetic operations.

| | |
|--------------------------|------------------------|
| <code>varsigma</code> | <code>- > 1</code> |
| <code>maxSpeed</code> | <code>- > 4</code> |
| <code>maxDistance</code> | <code>- > 10</code> |
| <code>maxInteger</code> | <code>- > 10</code> |

4.3 Problem 1: Getting from one node to another

This problem was designed to test whether the basic parts of the formalization work. Only `car1` and `road_ab` were used; `car2`, `road_bc`, and the node `c` were left out. In the process of leaving a node and going to another, a car must successfully and correctly execute an `ENTER_SEGMENT` action. Once on the segment, the speed must be determined automatically as the smaller of its own top speed and the speed limit of the segment. The laws causing the car to move along a segment must function properly. And finally, the law causing arrival at a node needs to work correctly.

The planning problem given to CCALC was:

```
:- plan
facts ::
  0: at_node(car1,a),
  0: /\C: /\S: -last(C,S);
goal ::
  6: at_node(car1,b).
```

The plan found is shown below. Note that we set CCALC so that it would show only the fluents which are true at each state and the actions executed between states.

```
0.  at_distance(car1,0)  at_node(car1,a)  at_speed(car1,0)

ACTIONS: enter_segment(car1,road_ab)

1.  at_distance(car1,0)  at_speed(car1,2)  in_direction(car1,forward)
last(car1,road_ab)  on_segment(car1,road_ab)

2.  at_distance(car1,2)  at_speed(car1,2)  in_direction(car1,forward)
```

```

last(car1,road_ab) on_segment(car1,road_ab)

3. at_distance(car1,4) at_speed(car1,2) in_direction(car1,forward)
last(car1,road_ab) on_segment(car1,road_ab)

4. at_distance(car1,6) at_speed(car1,2) in_direction(car1,forward)
last(car1,road_ab) on_segment(car1,road_ab)

5. at_distance(car1,8) at_speed(car1,2) in_direction(car1,forward)
last(car1,road_ab) on_segment(car1,road_ab)

6. at_distance(car1,0) at_node(car1,b) at_speed(car1,0)
last(car1,road_ab)

```

4.4 Problem 2: Getting from one node to another — concurrent change

This second problem is for checking whether concurrent change is handled without any problems. It involves two cars at the outer nodes in our landscape which want to meet at the node in the middle. The cars must leave their respective nodes, travel along the two segments and arrive at the middle node.

The planning problem was:

```

:- plan
facts ::
0: at_node(car1,a),
0: at_node(car2,c),
0: /\C: /\S: -last(C,S);
goal ::
6: at_node(car1,b),
6: at_node(car2,b).

```

The plan found was:

0. at_distance(car1,0) at_distance(car2,0) at_node(car1,a)
at_node(car2,c) at_speed(car1,0) at_speed(car2,0)

ACTIONS: enter_segment(car1,road_ab) enter_segment(car2,road_bc)

1. at_distance(car1,0) at_distance(car2,0) at_speed(car1,2)
at_speed(car2,4) in_direction(car1,forward) in_direction(car2,backward)
last(car1,road_ab) last(car2,road_bc) on_segment(car1,road_ab)
on_segment(car2,road_bc)

2. at_distance(car1,2) at_distance(car2,4) at_speed(car1,2)
at_speed(car2,4) in_direction(car1,forward) in_direction(car2,backward)
last(car1,road_ab) last(car2,road_bc) on_segment(car1,road_ab)
on_segment(car2,road_bc)

3. at_distance(car1,4) at_distance(car2,8) at_speed(car1,2)
at_speed(car2,4) in_direction(car1,forward) in_direction(car2,backward)
last(car1,road_ab) last(car2,road_bc) on_segment(car1,road_ab)
on_segment(car2,road_bc)

4. at_distance(car1,6) at_distance(car2,0) at_node(car2,b)
at_speed(car1,2) at_speed(car2,0) in_direction(car1,forward)
last(car1,road_ab) last(car2,road_bc) on_segment(car1,road_ab)

5. at_distance(car1,8) at_distance(car2,0) at_node(car2,b)
at_speed(car1,2) at_speed(car2,0) in_direction(car1,forward)
last(car1,road_ab) last(car2,road_bc) on_segment(car1,road_ab)

6. at_distance(car1,0) at_distance(car2,0) at_node(car1,b)
at_node(car2,b) at_speed(car1,0) at_speed(car2,0) last(car1,road_ab)
last(car2,road_bc)

4.5 Problem 3: A blocked car

This is the most complicated problem. In the first problem we had tested whether the causal laws for the automatic determination of speed worked as desired for a lone car. Here we test the law causing the fluent `blocked` and the law causing a blocked car to adjust its speed to meet the surrounding traffic restrictions.

The law causing the speed of a blocked car involves not only the blocked and blocking cars but also pays attention not to mistakenly adjust the speed according to other cars in front of the blocking car. In order to test this we added a third car to the world. Since the number of rules explodes with each new car, we weren't able to run the test with the world we described initially so we needed to make some changes to our world.

In this problem, there is only one segment (`road_ab`) and the length of that segment is 6. Of course, this means that `maxDistance` and `maxInteger` are also 6. The third car is named 'car3' and its top speed is 1. The initial state of the problem is shown in Figure 4.2.

The problem we gave to CCALC was:

```
:- plan
facts ::
0: on_segment(car1,road_ab),
0: on_segment(car2,road_ab),
0: on_segment(car3,road_ab),
0: in_direction(car1,forward),
0: in_direction(car2,forward),
0: in_direction(car3,forward),
0: at_distance(car1,2),
0: at_distance(car2,0),
0: at_distance(car3,5),
0: at_speed(car1,2),
0: at_speed(car3,1),
```



```

0: -blocked(car1),
0: -blocked(car3);
goal ::
4: at_node(car1,b),
4: at_node(car2,b),
4: at_node(car3,b).

```

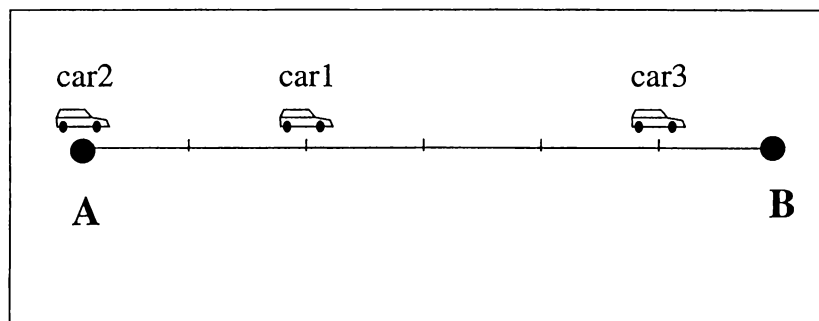


Figure 4.2: Initial state of problem 3

The solution plan returned was:

```

0. blocked(car2) at_distance(car1,2) at_distance(car2,0)
at_distance(car3,5) at_speed(car1,2) at_speed(car2,3) at_speed(car3,1)
in_direction(car1,forward) in_direction(car2,forward)
in_direction(car3,forward) last(car1,road_ab) last(car2,road_ab)
last(car3,road_ab) on_segment(car1,road_ab) on_segment(car2,road_ab)
on_segment(car3,road_ab)

1. blocked(car2) at_distance(car1,4) at_distance(car2,3)
at_distance(car3,0) at_node(car3,b) at_speed(car1,2) at_speed(car2,2)
at_speed(car3,0) in_direction(car1,forward) in_direction(car2,forward)
last(car1,road_ab) last(car2,road_ab) last(car3,road_ab)
on_segment(car1,road_ab) on_segment(car2,road_ab)

2. at_distance(car1,0) at_distance(car2,5) at_distance(car3,0)

```

```

at_node(car1,b) at_node(car3,b) at_speed(car1,0) at_speed(car2,4)
at_speed(car3,0) in_direction(car2,forward) last(car1,road_ab)
last(car2,road_ab) last(car3,road_ab) on_segment(car2,road_ab)

```

```

3. at_distance(car1,0) at_distance(car2,0) at_distance(car3,0)
at_node(car1,b) at_node(car2,b) at_node(car3,b) at_speed(car1,0)
at_speed(car2,0) at_speed(car3,0) last(car1,road_ab) last(car2,road_ab)
last(car3,road_ab)

```

```

4. at_distance(car1,0) at_distance(car2,0) at_distance(car3,0)
at_node(car1,b) at_node(car2,b) at_node(car3,b) at_speed(car1,0)
at_speed(car2,0) at_speed(car3,0) last(car1,road_ab) last(car2,road_ab)
last(car3,road_ab)

```

Notice that car2 is blocked in the initial state, so its speed is adjusted so that it will be exactly *varsigma* behind car1 at time 1. car3 affects neither car1 nor car2. At time 1, car2 is still blocked but now it is exactly *varsigma* behind car1. This time its speed is adjusted to simply match that of car1. At time 2, car1 gets off the segment. Not blocked anymore, the speed of car2 is changed to the maximum speed of 4.

Chapter 5

Comparison with a Previous Formalization

The only previous work on formalizing the TRAFFIC world has been done by Henschel and Thielscher [13]. They use the fluent calculus to axiomatize a slightly modified version of the original specification. For information about the fluent calculus the reader is referred to [34].

In this chapter we first note the parts of their formalization which are similar to ours (for convenience, we will be calling their formalization “the H-T formalization” from now on). Then we examine in detail what they have chosen to do differently. We show how these different aspects can be formalized using \mathcal{C} and combined with our formalization. Finally, we discuss the various advantages and disadvantages of choosing the \mathcal{C} action language or the fluent calculus to formalize the TRAFFIC world.

5.1 Similarities

Most parts of the H-T formalization are similar to ours, of course, due to its being based on the same specification given at LMW.

The world in their formalization is also made up of nodes and segments. Each segment connects two nodes, has a fixed length, and a fixed speed limit. All nodes and segments have unique names.

The activities are done by cars with each car having a fixed top speed. The cars cannot go faster than their top speeds or the speed limits of the segments they are driving on and they cannot violate the surrounding traffic restrictions.

At each point in time each car is either on a segment or at a node¹.

Cars are not allowed to make a U-turn and go back on the segment they just traveled on.

The minimum of two numbers is found using a function which is similar to our macro minimum.

5.2 Differences

Instead of having all cars obey the same rules, the H-T formalization separates cars into two groups: *deliberative* cars, and *non-deliberative* cars. Deliberative cars are under our control (i.e. we say what actions they will execute and when they will execute them) and non-deliberative cars are not.

A non-deliberative car must go at the maximum speed allowed by its own top speed, the speed limit of the segment it is on, and the surrounding traffic restrictions. In contrast, a deliberative car is free to go at any speed it likes, as long as it does not exceed the two limits and does not violate the surrounding traffic restrictions. There is an action *ChangeVel* for a deliberative car to change its speed.

In our formalization we had allowed cars to stay at nodes. In the H-T formalization, this right is reserved only for deliberative cars. Non-deliberative cars must turn into (enter, in our terminology) a segment as soon as they arrive at

¹There is a small catch to this which will be explained in the next section.

a node (this is denoted by the action *ArriveAt*). They are not allowed to stop. Deliberative cars which arrive at nodes (denoted by action *Arriveat_D*) may turn into segments when they want to (of course, they need to be at the start or end node of that segment). The action which accomplishes this is *Turn*.

Having non-deliberative cars turning into segments immediately and deliberative cars turning when they want to is likely to lead to violations of the surrounding traffic restrictions. To overcome this problem, a waiting area is used for each segment (Figure 5.1). When a car turns into a segment at a node, be it deliberative or not, instead of starting to move, it is placed in the next free spot in the waiting area of that segment-node pair. If there is another car on the segment at a distance less than *varsigma*, it will wait. Otherwise it is immediately set in motion. There is a counter at the waiting area which is incremented or decremented as each car enters or leaves the waiting area, respectively. Each car entering the waiting area is assigned a number in the queue (which is the current number of the counter). The counter is represented by the fluent *Counter* taking nodes, segments and numbers as arguments. The fluent with the number argument n is true when the next free spot available is the n^{th} one.

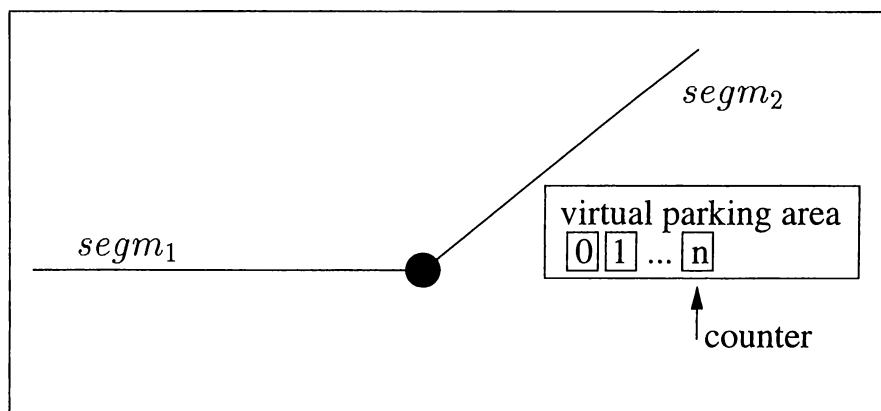


Figure 5.1: Waiting areas for segments²

For each node, priorities are assigned to segments which connect to that node. When more than one car tries to turn into a segment, they are placed into the

²This figure has been drawn by us based on a figure which appears in [13].

waiting area in order of decreasing priority of the segment they are coming from. This is just like the right-of-way rule at real life intersections.

Allowing the release of cars from the waiting area onto the segment is organized as follows: There is a virtual traffic light. When the car which has most recently moved onto the segment reaches a distance of $varsigma$, the traffic light is triggered and the next car in the queue is allowed to go (Figure 5.2).

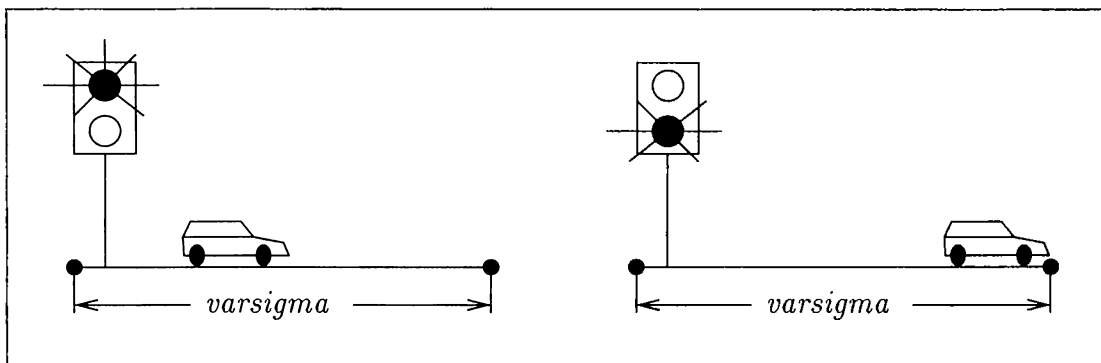


Figure 5.2: Triggering the traffic light³

The virtual traffic light is formalized as a fluent *Blocked* (with 2 arguments: a node and a segment) which is true when there is a car waiting at spot 0 (the first spot) in the waiting area for that segment-node pair. The action *Unblock* makes the fluent *Blocked* false when a car which left that node and got on the segment reaches a distance of $varsigma$.

A major difference between the two formalizations is with respect to time. In \mathcal{C} , there are states and transitions between states. Each causal law expresses a condition which causally explained transitions must satisfy. Intuitively, each distinct point in time is represented by a state⁴ and this is the interpretation we have used in the TRAFFIC formalization. In contrast, in the fluent calculus, time

³This figure has been drawn by us based on a figure which appears in [13].

⁴Although this is the most natural interpretation which is strongly suggested by the use of the word **after** in dynamic causal laws, it does not necessarily have to be the case. Recently, we have heard that work is being done on representing time as a fluent with two arguments, the numerator and denominator of a rational number [16].

is a sort, the elements of which are the real numbers along with their standard interpretation [13, Appendix A], so the H-T formalization uses time values as arguments in the fluents.

When a car is moving on a segment, instead of representing the distance it has traveled so far, the distance the car had left behind at the time of the last change of velocity is represented, along with the time of the last change of velocity. The distance the car has traveled at the current time can then be calculated using these two values.

Segments don't have a start node and an end node. Instead, a predicate with three arguments describes that the segment connects two nodes and for each such predicate which holds, a predicate with the node arguments interchanged holds. Since segments do not have a direction, the information about which way the car is going is represented as the node it is heading for.

At each point in time a car is in exactly one of three states: moving, waiting, or at a node (only deliberative cars can be in this last state). This is formalized as three fluents: *Movement* (with arguments for the car, distance, velocity, time, segment, and node), *Waiting* (with arguments for the car, node, segment, and number), *AtNode* (with arguments for the car, node, and segment).

In the section where we mention the similarities between the two formalizations, we said that a car is always either at a node or on a segment. This is because we consider waiting cars as being on the segments for which they are waiting. Once a car turns into that segment it is just a matter of time before it starts traveling on it. It cannot change its mind and later choose another segment.

In addition to the two actions *ChangeVel* and *Turn* which deliberative cars can execute, there are four “natural” actions. They are *ArriveAt*, *ArriveAt_D*, *Approach*, and *Unblock* which denote a non-deliberative car arriving at a node, a deliberative car arriving at a node, a car reaching the safety distance of the car in front, and a segment leaving a node becoming unblocked, respectively. These are actually not “actions which are executed” but “events which take place” when the

preconditions are satisfied. Nonetheless, they have certain effects on the state of things. The effects of all except *Approach* were described above. *Approach* makes a car adjust its speed to match the speed of the car in front.

Arithmetical operators like addition, subtraction and multiplication are part of the language and not fluents (nor macros).

5.3 Formalizing the Different Aspects in \mathcal{C}

The sections of the H-T formalization which are different from ours can be formalized as causal laws using \mathcal{C} and incorporated into our formalization.

Let's start with separating cars into deliberative and non-deliberative cars. This is easily done by adding a fluent which indicates whether a car is deliberative or not:

```
deliberative(car)
```

Since the fact that a car is deliberative or not does not change, we make both `deliberative` and `¬deliberative` inertial.

```
inertial deliberative(C), ¬deliberative(C).
```

Non-deliberative cars must turn into a segment in the next time instant after they arrive at a node. They cannot stay at a node.

```
caused False if ¬deliberative(C) && at_node(C,N)
      after at_node(C,N).
```

Notice that non-deliberative cars still need to execute an action to turn into a segment.

We add a new action for deliberative cars to change their speed:

`CHANGE_SPEED(car,speed)`

This action can only be performed by deliberative cars.

nonexecutable `CHANGE_SPEED(C,Sp)` **if** `¬deliberative(C)`.

The effect of executing `CHANGE_SPEED` is to set the speed to a new value.

`CHANGE_SPEED(C,Sp)` **causes** `at_speed(C,Sp)`.

No deliberative car can change its speed to go faster than the speed limit of the segment it is on or its own top speed.

nonexecutable `CHANGE_SPEED(C,Sp)`
 if `on_segment(C,S) && speed_limit(S,Sp1) && (Sp > Sp1)`.
nonexecutable `CHANGE_SPEED(C,Sp)`
 if `max_car_speed(C,Sp1) && (Sp > Sp1)`.

Cars which are at nodes cannot change their speeds because this would conflict with the law causing speed to be 0 at a node.

nonexecutable `CHANGE_SPEED(C,Sp)` **if** `at_node(C,N)`.

A car cannot change its speed to its current speed (this would not violate anything, but it can be confusing to see a `CHANGE_SPEED` action be executed without the speed changing).

nonexecutable `CHANGE_SPEED(C,Sp)` **if** `at_speed(C,Sp)`.

The speeds of non-deliberative cars are determined automatically by the laws given. We only need to make sure that the laws are only for non-deliberative

cars. This is done by adding '&& \neg deliberative(C)' to the end of each of the three laws in Section 3.5.

We would like to have a waiting area for each segment-node pair. When a car at a node enters a segment, it should be placed in this area in the appropriate place in a queue. The appropriate place is determined by the priority of the segment the car came to the node on. To formalize all this, we will need a new sort *number* to represent the position in the queue and the priority, and two new fluents:

`waiting(car,segment,node,number)`
`priority(node,segment,number)`

We will use the variables Nu, Nu1, Nu2, Nu3 for the sort *number*.

The fluent *priority* will need to be made inertial since the priority of a segment does not change, and the uniqueness laws should be included:

inertial `priority(N,S,Nu)`.
caused \neg `priority(N,S,Nu)` if `priority(N,S,Nu1)` && \neg (Nu=Nu1).

The causal law which stated that entering a segment causes the car to be on that segment should be removed. The new effect of ENTER_SEGMENT is to place the car in the waiting area.

The waiting area imposes many different constraints on cars:

- Each car in it must be in a unique position in the queue.
- Multiple cars entering the area at the same instant must be placed into the queue according to the priority of the segment they are coming from (and they should be placed behind all the cars which are already waiting).
- The order in the queue should be preserved from one time instant to the next.

- There should be no gaps in the queue. (This means that no car is behind an empty spot in the queue.)

In order not to violate the surrounding traffic restrictions, the first car in the queue does not get on the segment (which now is different from entering the segment) until the road is clear up to a distance of *varsigma*. And as soon as the road is clear it moves onto the segment.

This means that a car may leave the waiting area at the same time that some other cars are entering the area. This and all the constraints listed above suggest that calculating new values for the position of cars in a queue and causing fluents to be true at these calculated values is very difficult⁵. Interestingly (but not really surprisingly), the hint to a succesful solution is hidden in that last sentence which seems to imply failure. Constraints is the keyword here. We can make ENTER_SEGMENT affect the fluent *waiting* nondeterministically (by a rule like 2.12 in Section 2.2.3) and then formalize all the constraints so that the only fluents which can be chosen nondeterministically will be the ones which satisfy the constraints. We want a car to be waiting at a unique position after ENTER_SEGMENT(C,S). This can be expressed by three causal laws:

```

caused waiting(C,S,N,Nu) if waiting(C,S,N,Nu)
    after ENTER_SEGMENT(C,S) && at_node(C,N).
caused False if (/Nu:( ¬waiting(C,S,N,Nu) ) )
    after ENTER_SEGMENT(C,S) && at_node(C,N).
caused ¬waiting(C,S,N,Nu) if waiting(C,S,N,Nu1) && ¬(Nu=Nu1).

```

Executing ENTER_SEGMENT(C,S) also causes (indirectly) the car to no longer be at a node.

```

caused ¬at_node(C,N) if waiting(C,S,N,Nu).

```

⁵If not impossible! We spent many fruitless hours trying to devise such laws but in the end discovered a different way based on constraints.

When a car is waiting in the waiting area of a segment leaving a node, it cannot be waiting anywhere else.

```
caused  $\neg$ waiting(C,S,N,Nu)
      if waiting(C,S1,N1,Nu1) &&  $\neg$ ( (S=S1) && (N=N1) ).
```

Now we formalize the constraints. Two cars entering a segment at the same time must be placed in positions respecting the priority values of the segments they came on:

```
caused False if priority(N,S,Nu) && priority(N,S1,Nu1)
      && waiting(C,S2,N,Nu2) && waiting(C1,S2,N,Nu3)
      && (Nu>Nu1) && (Nu2>=Nu3)
      after last(C,S) && last(C1,S1).
      && at_node(C,N) && at_node(C1,N)
      && ENTER_SEGMENT(C,S2) && ENTER_SEGMENT(C1,S2).
```

Any car which just entered the waiting area must be behind all the other cars which were already in the queue at the previous time:

```
caused False if waiting(C,S,N,Nu)
      && waiting(C1,S,N,Nu1) && (Nu>=Nu1)
      after waiting(C,S,N,Nu2)
      && ( $\forall$ Nu3: (  $\neg$ waiting(C1,S,N,Nu3) ).
```

Cars already in the waiting queue must preserve their order:

```
caused False if waiting(C,S,N,Nu)
      && waiting(C1,S,N,Nu1) && (Nu>=Nu1)
      after waiting(C,S,N,Nu2) && waiting(C1,S,N,Nu3)
      && (Nu2<Nu3).
```

There are no gaps in the queue. That is, for all cars in the queue except the first car, there is another car in the spot immediately in front of it.

```

caused False if waiting(C,S,N,Nu)
      && diff(1,Nu,Nu1) &&  $\neg(\exists C1: \text{waiting}(C1,S,N,Nu1) )$ 
      &&  $\neg(\text{Nu}=0)$ .

```

In a transition from one state to the next, two things may happen to cars in the queue. They will either move up a spot (if the first car gets on the segment) or they may stay at their current places. Also if a car is not in any queue then it will probably still not be in a queue so there should be a law which causes this. These last two cases are inertial.

```

caused waiting(C,S,N,Nu) if waiting(C,S,N,Nu)
      after waiting(C,S,N,Nu1) && diff(1,Nu1,Nu).
inertial waiting(C,S,N,Nu),  $\neg$ waiting(C,S,N,Nu).

```

Now we have all the numbering of and entering into the waiting area under control. So when do cars leave the waiting area? Simply when there is no car on the segment which has traveled a distance less than *varsigma*. Indication of whether there is such a car is done by the help of a fluent:

```

blocked_segment(segment,node)

```

Typically, the segment is expected not to be blocked:

```

default  $\neg$ blocked_segment(S,N).

```

The law to cause `blocked_segment` is

```

caused blocked_segment(S,N)
      if on_segment(C,S) && at_distance(C,D)
      && ( (in_direction(C,forward) && startnode(S,N))
          ++ (in_direction(C,backward) && endnode(S,N)) )
      && (D < varsigma) &&  $\neg$ waiting(C,S,N,0).

```

Notice that if the only car on the segment closer than $varsigma$ is the first car in the queue, then the road is not blocked. This is because we want the first car to be on the segment if it is not blocked.

```
caused on_segment(C,S)
  if waiting(C,S,N,0) && ¬blocked_segment(S,N).
```

Once the first car gets on the segment it will not be in the queue at the next state.

```
caused ¬waiting(C,S,N,0)
  after on_segment(C,S) && waiting(C,S,N,0).
```

The above laws secure the correct management of waiting and queueing. The surrounding traffic restrictions were already formalized in our original formalization and they work for non-deliberative cars. However, for deliberative cars, we need to add two extra laws to prevent overtaking. This is because non-deliberative cars set their own speeds and if the speed difference between two cars is more than $2 \times varsigma$, then the car behind at one time instant may be ahead at the next time instant and we want to prevent this. The first law below prevents overtaking when the cars would both still be on the segment after the overtaking. The second law prevents the case where the overtaking car arrives at a node (and its speed drops to 0).

```
caused False if on_segment(C,S) && on_segment(C1,S) && ¬(C=C1)
  && in_direction(C,Dir) && in_direction(C1,Dir)
  && at_distance(C,D) && at_distance(C1,D1) && (D1>D)
  after at_distance(C,D2) && at_distance(C1,D3)
  && (D2>D3).

caused False if at_node(C,N) && on_segment(C1,S) && ¬(C=C1)
  after on_segment(C,S) && on_segment(C1,S)
  && in_direction(C,Dir) && in_direction(C1,Dir)
  && at_distance(C,D) && at_distance(C1,D1)
  && (D1>D).
```

We complete the formalization of the different aspects by making the fluents `at_distance` and `at_speed` inertial when true. This is because we need to have some value of these fluents caused while waiting in the queue.

`inertial at_distance(C,D), at_speed(C,Sp).`

5.4 Fluent Calculus vs \mathcal{C} for the TRAFFIC World

In our view, the biggest difference between fluent calculus and \mathcal{C} is that the former allows continuous, real-valued time, whereas the latter does not. Due to the discrete nature of \mathcal{C} which is based on transition systems, we were only able to formalize a discrete-time version of the TRAFFIC world whereas the H-T formalization is for continuous time.

Having continuous time makes the problem more realistic. On the other hand, one may argue that continuous change is a matter of granularity. The smaller the time steps get, the more realistic the model becomes.

We showed that all the aspects of the H-T formalization which differ from our formalization (except continuous time) could be formalized using \mathcal{C} . This basically shows that in this domain, \mathcal{C} is no less expressive than the fluent calculus.

Furthermore, formalizing the differences required almost no modification of the laws in our original formalization. Only one causal law needed to be removed from our formalization and a conjunction was added to the end of three other laws. This is quite an interesting and even impressive fact. We may say that our original formalization has proven to be robust and easily extendable. This is an important property for formalizations, which is desired by the Logic Modeling Workshop. It also shows that our formalization is “elaboration tolerant” [29] to a certain extent⁶.

⁶Previously, [20] discussed the input language of the Causal Calculator with respect to the problem of elaboration tolerance.

Using \mathcal{C} to model TRAFFIC has one huge advantage over fluent calculus. To our knowledge, there is no tool which does for fluent calculus, what CCALC does for \mathcal{C} . Having a well-functioning *computational* tool to test the formalization is invaluable. While we were working on formalizing TRAFFIC, with each small change, we were able to test whether we had overlooked any details. Without such a tool, a formalization may look correct but fail due to some small mistake.

Unlike toy problems, the TRAFFIC domain has strong ties to the real world. Having a computational tool to do planning with a formalization is a valuable resource for applications, besides being a helper for testing formalizations. We think that this, together with the fact that we were able to formalize everything the H-T formalization did, makes \mathcal{C} a better choice than fluent calculus for the TRAFFIC World, despite the fact that we cannot represent continuous time.

Chapter 6

Conclusion

Our aim in this thesis was to formalize the TRAFFIC world using the \mathcal{C} action language. This is an important goal because of its implications for logic-based formal knowledge representation methods. Although many logic-based methods have been defined, the problems typically used to show their expressiveness are of small size. It is time, especially with funding agencies nowadays tending to favor research leading to practical applications, for researchers in logic-based methods to demonstrate that the formalisms they invent are useful for more than just toy problems.

The formalization of the TRAFFIC world presented shows that it is possible to formalize domains of nontrivial size in \mathcal{C} . The TRAFFIC world models most of the important properties of real traffic: cars moving on a well-defined landscape of nodes and segments, traveling at well-defined speeds, and coping safely with the surrounding traffic. Being able to formalize these important properties of real world traffic points out to us the possibility of some future work which might be built upon the current formalization. The domain specification may be made more detailed and more in correspondence with the rules governing real life traffic.

Test problems were run to demonstrate that the formalization functioned as desired when applied to planning problems. We postulate that, because the basic rules are the same no matter how many cars, nodes, or segments we have, bigger

planning problems should present no challenge to the formalization. However, to say that the formalization is scalable is *not* to say that the implementation is also scalable. It was observed that, as the values of the largest integers for the speeds of the cars or the lengths of the roads were increased, the number of rules exploded. This is because numbers require a large number of uniqueness laws and also because domains involving numbers usually require laws involving multiple number variables (for arithmetic operations). The number of ground rules of laws with multiple variables of a single sort increases exponentially as the number of the constants of that sort increases. This suggests another area of future research: representing numbers in action languages and other logic-based approaches in ways which would yield efficient implementations.

The formalization was contrasted with the formalization of Henschel and Thielscher which uses the fluent calculus [13]. It was shown that, when new specifications were added to the domain, we were able to extend the existing formalization without changing much. Such modularity and suitability for successive development is an important property of the formalization. We showed that in the TRAFFIC domain, except for being unable to represent time continuously, \mathcal{C} is no less expressive than the fluent calculus.

Appendix A

The CCALC Program Listing

A.1 The World Description File

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
% Last update: July 5, 2000  
% Author: Selim T. Erdogan  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
:- include 'C.t'.
```

```
:- sorts
```

```
    car;  
    node;  
    segment;  
    distance;  
    speed;  
    direction.
```

```
:- variables
```

```
    C,C1,C2          :: car;
```



```

% a car cannot be at two nodes at once.
caused -at_node(C,N) if at_node(C,N1) && -(N=N1).

% a car cannot be on two segments at once.
caused -on_segment(C,S) if on_segment(C,S1) && -(S=S1).

% a car cannot be at two points on a segment at once.
caused -at_distance(C,D) if at_distance(C,D1) && -(D=D1).

% a car cannot be traveling at two different speeds at once
caused -at_speed(C,Sp) if at_speed(C,Sp1) && -(Sp=Sp1).

% a car cannot be travelling in two different directions at once
caused -in_direction(C,Dir) if in_direction(C,Dir1) && -(Dir=Dir1).

% the segment that each car most recently travelled on is unique
caused -last(C,S) if last(C,S1) && -(S=S1).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Action: enter_segment(car,segment)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

enter_segment(C,S) causes on_segment(C,S).

enter_segment(C,S) causes at_distance(C,0).

enter_segment(C,S) causes in_direction(C,forward) if startnode(S,N)
&& at_node(C,N).

enter_segment(C,S) causes in_direction(C,backward) if endnode(S,N)
&& at_node(C,N).

```

```

% once a car enters a segment, it is no longer at any node.
caused -at_node(C,N) if on_segment(C,S).

% the segment most recently travelled on is recorded
caused last(C,S) if on_segment(C,S).

% a car cannot enter a new segment unless it is at a node
nonexecutable enter_segment(C,S) if -( \N: at_node(C,N) ).

% a car cannot enter a segment while not at the startnode or
% endnode of that segment
nonexecutable enter_segment(C,S) if at_node(C,N)
                                && -( startnode(S,N) ++ endnode(S,N) ).

% a car cannot return on a segment it most recently drove on
nonexecutable enter_segment(C,S) if last(C,S).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Movement of cars which are on segments
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% the distance that a car has travelled along a segment is incremented
% by the amount of its speed
caused at_distance(C,X) after at_distance(C,D1) && at_speed(C,Sp)
                                && sum(X,Sp,D1) && on_segment(C,S)
                                && length(S,D2) && (X<D2).

caused at_node(C,N) after at_distance(C,D1) && at_speed(C,Sp)
                                && sum(X,Sp,D1) && on_segment(C,S)
                                && length(S,D2) && (X>=D2)
                                && ( (in_direction(C,forward) && endnode(S,N))
                                ++(in_direction(C,backward) && startnode(S,N)) ).

```



```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% a car is blocked if traveling at the maximum speed allowed by the
% road and its own top speed will make it get closer than varsigma
% to a car in front of it
```

```
caused blocked(C) if on_segment(C,S) && on_segment(C1,S) && -(C=C1)
    && in_direction(C,Dir) && in_direction(C1,Dir)
    && at_distance(C,D) && at_distance(C1,D1) && (D1>D)
    && speed_limit(S,Sp1) && max_car_speed(C,Sp2)
    && minimum(X,Sp1,Sp2) && sum(X1,X,D)
    && at_speed(C1,Sp3) && sum(X2,Sp3,D1)
    && diff(X3,X2,varsigma) && (X1>X3).
```

```
% cars which are not blocked travel at the maximum speed that the speed
% limit of the road and the top speed of the car allow
```

```
caused at_speed(C,X) if -blocked(C) && on_segment(C,S)
    && speed_limit(S,Sp1) && max_car_speed(C,Sp2)
    && minimum(X,Sp1,Sp2).
```

```
% a blocked car adjusts its speed in a way so that it is exactly
% varsigma away from the car blocking it (the car blocking is the
% closest one in front of it)
```

```
caused at_speed(C,Sp1) if blocked(C) && on_segment(C,S)
    && on_segment(C1,S)
    && -(C=C1) && in_direction(C,Dir)
    && in_direction(C1,Dir) && at_distance(C,D)
    && at_distance(C1,D1) && (D1>D) && at_speed(C1,Sp)
    && (/ \C2: ( -on_segment(C2,S) ++ (C2=C) ++ (C2=C1)
    ++ -in_direction(C2,Dir)
    ++ (at_distance(C2,D2) && (D2>D1)) ) )
```



```

length(#1,#2) -> (   (#1=road_ab && #2=10)
                    ++(#1=road_bc && #2=10) );

speed_limit(#1,#2) -> (   (#1=road_ab && #2=4)
                          ++(#1=road_bc && #2=4) );

% change this macro to add cars
max_car_speed(#1,#2) -> (   (#1=car1 && #2=2)
                            ++(#1=car2 && #2=4) ).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Macros for arithmetic
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
:- macros
    sum(#1,#2,#3) -> #1 is min((#2)+(#3), maxDistance); % addition

    absdiff(#1,#2,#3) -> #1 is abs((#2)-(#3));

    diff(#1,#2,#3) -> #1 is max((#2)-(#3), 0);

    minimum(#1,#2,#3) -> #1 is min(#2,#3).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Domain description
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- include 'new_traffic1.t'.

:- constants
    car1      :: car;
    a         :: node;

```



```

% surrounding traffic restriction parameter
varsigma -> 1;

% Largest values for the speeds and distances
maxSpeed -> 4;
maxDistance-> 10;

% change the following four macros to add nodes and segments
startnode(#1,#2) -> ( (#1=road_ab && #2=a)
                      ++(#1=road_bc && #2=b) );

endnode(#1,#2) -> ( (#1=road_ab && #2=b)
                    ++(#1=road_bc && #2=c) );

length(#1,#2) -> ( (#1=road_ab && #2=10)
                   ++(#1=road_bc && #2=10) );

speed_limit(#1,#2) -> ( (#1=road_ab && #2=4)
                        ++(#1=road_bc && #2=4) );

% change this macro to add cars
max_car_speed(#1,#2) -> ( (#1=car1 && #2=2)
                          ++(#1=car2 && #2=4) ).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Macros for arithmetic
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
:- macros
sum(#1,#2,#3) -> #1 is min((#2)+(#3), maxDistance); % addition

absdiff(#1,#2,#3) -> #1 is abs((#2)-(#3));

```

```

diff(#1,#2,#3) -> #1 is max((#2)-(#3), 0);

minimum(#1,#2,#3) -> #1 is min(#2,#3).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Domain description
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- include 'new_traffic1.t'.

:- constants
    car1      :: car;
    car2      :: car;
    a         :: node;
    b         :: node;
    c         :: node;
    road_ab   :: segment;
    road_bc   :: segment;

    0..maxDistance  :: distance;
    0..maxSpeed     :: speed.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Problem description
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- plan
facts ::
    0: at_node(car1,a),
    0: at_node(car2,c),
    0: /\C: /\S: -last(C,S);

```

```
goal ::
  6: at_node(car1,b),
  6: at_node(car2,b).
```

A.4 Scenario File for Problem 3

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Last update: July 5, 2000
% Author: Selim T. Erdogan
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% macros for constants
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- macros

    % surrounding traffic restriction parameter
    varsigma -> 1;

    % Largest values for the speeds and distances
    maxSpeed -> 4;
    maxDistance-> 6;

% change the following four macros to add nodes and segments
    startnode(#1,#2) -> (#1=road_ab && #2=a);

    endnode(#1,#2) -> (#1=road_ab && #2=b);

    length(#1,#2) -> (#1=road_ab && #2=6);
```

```

    speed_limit(#1,#2) -> (#1=road_ab && #2=4);

% change this macro to add cars
    max_car_speed(#1,#2) -> (    (#1=car1 && #2=2)
                                ++(#1=car2 && #2=4)
                                ++(#1=car3 && #2=1) ).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Macros for arithmetic
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
:- macros
    sum(#1,#2,#3) -> #1 is min((#2)+(#3), maxDistance); % addition

    absdiff(#1,#2,#3) -> #1 is abs((#2)-(#3));

    diff(#1,#2,#3) -> #1 is max((#2)-(#3), 0);

    minimum(#1,#2,#3) -> #1 is min(#2,#3).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Domain description
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- include 'new_traffic1.t'.

:- constants
    car1      :: car;
    car2      :: car;
    car3      :: car;
    a         :: node;
    b         :: node;

```

```
road_ab      :: segment;

0..maxDistance  :: distance;
0..maxSpeed     :: speed.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Problem description
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- plan
facts ::
  0: on_segment(car1,road_ab),
  0: on_segment(car2,road_ab),
  0: on_segment(car3,road_ab),
  0: in_direction(car1,forward),
  0: in_direction(car2,forward),
  0: in_direction(car3,forward),
  0: at_distance(car1,2),
  0: at_distance(car2,0),
  0: at_distance(car3,5),
  0: at_speed(car1,2),
  0: at_speed(car3,1),
  0: -blocked(car1),
  0: -blocked(car3);
goal ::
  4: at_node(car1,b),
  4: at_node(car2,b),
  4: at_node(car3,b).
```


Appendix B

The TRAFFIC Scenario World

B.1 Introduction

The TRAFFIC scenario world is intended to capture simple hybrid phenomena: vehicles moving continuously with well defined velocities along roads with well defined lengths, respecting speed limits and other restrictions on the vehicle's behaviors.

B.2 Landscape Structure

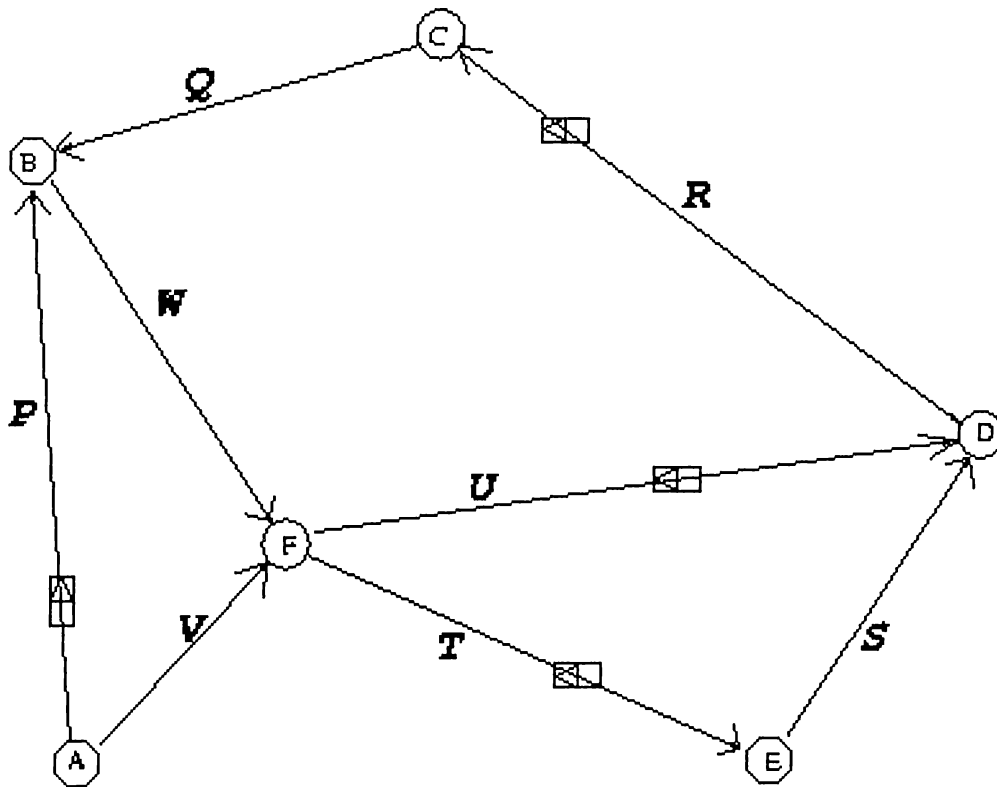
The landscape in the TRAFFIC Scenario World uses the following two types:

- *Nodes*, which can be thought of as road crossings without any particular structure (no lanes, etc)
- *Segments*, which can be thought of as road segments each of which connects two nodes.

The set of nodes and the set of segments are both considered as fully known, and all nodes and segments can be assigned individual names.

Each segment has exactly one *start node* and exactly one *end node*. It also has a *length*, which is a real number (or rational number, if preferred). This is all the structure there is.

Figure 1 shows an example of a traffic landscape that may be used for specific scenarios.



B.3 Activities in the TRAFFIC world

The activity structure in the TRAFFIC world uses only one sort:

- *Cars*, which are intuitively thought of as driving along the arcs in the TRAFFIC landscape structure.

Each car has a *position* at each point in time. The position is indicated as a pair consisting of the segment where the car is located, and the distance travelled along the segment. The distance travelled is a number between 0 and the segment's length.

Each car has a *top speed*, and each road segment has a *speed limit*. The actual velocity of a car at each point in time is the maximum velocity allowed by the following three conditions:

- The speed limit of the road segment where it is driving
- Its own top speed
- Surrounding traffic restrictions

Cars drive at piecewise constant velocity, and can change velocity discontinuously. (A more refined variant, TRAFFIC2, will require cars to change their velocity continuously, and assumes piecewise constant acceleration/ deceleration). When a car arrives at a node (“intersection”) then it may continue on any segment that connects to that node, except the one it is arriving at.

Cars can drive in both “directions” along a segment, that is, they can move both from the start node to the end node, and vice versa.

Cars can not overtake — if two cars go in the same direction on the same road segment, and one catches up with the other, then it has to stay behind at least until they arrive to the next node, where possibly the second car can choose another direction onwards. Cars going in opposite directions on the same segment can meet without difficulty, however.

The surrounding traffic restriction says that a car is never allowed to be closer than a fixed safety distance *varsigma* to the car in front of it, and it may never get itself into a situation where that could happen. This means, first of all, that when it gets to a distance of *varsigma* to a car moving in front of it on the same segment and in the same direction, then it must reduce speed to match the speed of the car in front of it. Also when getting close to a node (= an intersection),

a car must reduce its speed in a way that takes into consideration all other cars that are just approaching or leaving the same node.

Bibliography

- [1] *Proc. of IJCAI-95*, 1995.
- [2] *Proc. of AAAI-97*, 1997.
- [3] Chitta Baral and Michael Gelfond. Reasoning about effects of concurrent actions. *Journal of Logic Programming*, 31:85–118, 1997.
- [4] Chitta Baral, Michael Gelfond, and Alessandro Provetti. Reasoning about actions: Laws, observations and hypotheses. *Journal of Logic Programming*, 31:201–244, 1997.
- [5] Roberto Bayardo and Robert Shrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proc. of AAAI-97* [2], pages 203–208.
- [6] Keith Clark. Negation as failure. In Herve Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.
- [7] Patrick Doherty, Joakim Gustafsson, Lars Karlsson, and Jonas Kvarnström. Tal: Temporal action logics language <> Specification and tutorial. *Linköping Electronic Articles in Computer and Information Science*, 3(15), 1998. <http://www.ep.liu.se/ea/cis/1999/015/>.
- [8] *Electronic Transactions on Artificial Intelligence*. <http://www.ida.liu.se/ext/etai>.
- [9] Hector Geffner. Causal theories for nonmonotonic reasoning. In *Proc. of AAAI-90*, pages 524–530, 1990.

- [10] Michael Gelfond and Vladimir Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming*, 17:301–322, 1993.
- [11] Michael Gelfond and Vladimir Lifschitz. Action languages. *Linköping Electronic Articles in Computer and Information Science*, 3(16), 1998. <http://www.ep.liu.se/ea/cis/1998/016/>.
- [12] Enrico Giunchiglia and Vladimir Lifschitz. An action language based on causal explanation: Preliminary report. In *Proc. of AAAI-98*, pages 623–630, 1998.
- [13] Andreas Henschel and Michael Thielscher. The LMW traffic world in the fluent calculus. *Linköping Electronic Articles in Computer and Information Science*, 4 (number not yet determined), 1999. <http://www.ep.liu.se/ea/cis/1999/>¹
- [14] Henry Kautz and Bart Selman. Planning as satisfiability. In *Proc. of ECAI-92*, pages 359–379, 1992.
- [15] Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proc. of AAAI-96*, pages 1194–1201, 1996.
- [16] Joohyung Lee, June 2000. Personal correspondence.
- [17] Hector Levesque, Fiora Pirri, and Ray Reiter. Foundations for the situation calculus. *Linköping Electronic Articles in Computer and Information Science*, 3(18), 1998. <http://www.ep.liu.se/ea/cis/1999/018/>.
- [18] Vladimir Lifschitz. On the logic of causal explanation. *Artificial Intelligence*, 96:451–465, 1997.
- [19] Vladimir Lifschitz. Two components of an action language. *Annals of Mathematics and Artificial Intelligence*, 21:305–320, 1997.

¹The article is not yet available at the URL above. Currently it may be accessed at <http://www.ida.liu.se/ext/etai/lmw/TRAFFIC/001/index.html> .

- [20] Vladimir Lifschitz. Missionaries and cannibals in the causal calculator. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Seventh International Conference*, pages 85–96, 1999.
- [21] Vladimir Lifschitz and Hudson Turner. Representing transition systems by logic programs. In *Proc. of the Fifth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99)*, pages 92–106, 1999.
- [22] Fangzhen Lin. Embracing causality in specifying the indirect effects of actions. In *Proc. of IJCAI-95* [1], pages 1985–1991.
- [23] Logic Modelling Workshop, 1999. <http://www.ida.liu.se/ext/etai/lmw>.
- [24] Norman McCain. *Using the Causal Calculator with the C Input Language*, 1999. <http://www.cs.utexas.edu/users/tag/cc/>.
- [25] Norman McCain and Hudson Turner. A causal theory of ramifications and qualifications. In *Proc. of IJCAI-95* [1], pages 1970–1984.
- [26] Norman McCain and Hudson Turner. Causal theories of action and change. In *Proc. of AAAI-97* [2], pages 460–465.
- [27] Norman McCain and Hudson Turner. On relating causal theories to other formalisms. Unpublished manuscript. <http://www.d.umn.edu/~hudson/poct3.ps> , 1997.
- [28] Norman McCain and Hudson Turner. Satisfiability planning with causal theories. In *Proc. of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR98)*, pages 212–223, 1998.
- [29] John McCarthy. Elaboration tolerance. In progress. <http://www-formal.stanford.edu/jmc/elaboration.html>, 1999.
- [30] John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969.

- [31] Rob Miller and Murray Shanahan. The event calculus in classical logic — alternative axiomatisations. *Linköping Electronic Articles in Computer and Information Science*, 4(16), 1999. <http://www.ep.liu.se/ea/cis/1999/016/>.
- [32] Electronic Colloquium on Reasoning About Actions and Change. <http://www.ida.liu.se/ext/etai/rac>.
- [33] Erik Sandewall. Cognitive robotics logic and its metatheory: Features and fluents revisited. *Linköping Electronic Articles in Computer and Information Science*, 3(17), 1998. <http://www.ep.liu.se/ea/cis/1998/017/>.
- [34] Michael Thielscher. Introduction to the fluent calculus. *Linköping Electronic Articles in Computer and Information Science*, 3(14), 1998. <http://www.ep.liu.se/ea/cis/1998/014/>.
- [35] Hudson Turner. Representing actions in logic programs and default theories: A situation calculus approach. *Journal of Logic Programming*, 31:245–298, 1997.
- [36] Hantao Zhang. Sato: an efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction (CADE-97)*, pages 272–275, 1997.