Multi-Agent Planning Using an *Abductive* Event Calculus

Christoph G. Jung*, Klaus Fischer, Alastair Burt

* *Graduiertenkolleg Kognitionswissenschaft*, Universität des Saarlandes

Acknowledgements

We would like to thank Prof. Dr. Jörg Siekmann for supervising the development of the planning system that is described in this report.

We appreciate the work of Prof. Dr. Robert Kowalski and his group at the 'Imperial College', London. They created the original formulation of the EVENT CALCULUS and were the first to apply it to planning. The immediate inspiration for our approach, however, has been the work of Lode Missiaen, Maurice Bruynooghe, and Marc Denecker from the 'KU', Leuven. Besides their planning system CHICA, they have also developed the foundations of SLDNFA$^+$, the basic *proof procedure* used in our implementation, EVE.

Further acknowledgements are due to the Programming Systems Lab of the DFKI GmbH and its head Prof. Dr. Gert Smolka for the design and the implementation of the wonderful OZ calculus, the *constraint-based* implementation platform of EVE. The group, especially Christian Schulte, were always ready to give assistance. Without Christian's backing and the discussions about constraint-based programming and *encapsulated search*, EVE would have never come to life (... and what would have ADAM said to that ?).

Gero Vierke has been a valuable help by proof-reading this report.

Christoph G. Jung would also like to thank the *Graduiertenkolleg Kognitionswissenschaft* at the Universität des Saarlandes for their unique support.

Contents

1 Preface **9**

2 Introduction **11**
2.1 Logic . . . 11
2.2 Planning . . . 13
2.3 EVE's Architecture . . . 16

3 Theory of Time and Action: the Event Calculus **18**
3.1 Requirements . . . 18
3.2 The Situation Calculus . . . 19
3.3 The Event Calculus by Kowalski & Shanahan . . . 20
3.4 The Event Calculus by Missiaen . . . 22
3.5 The Simple Event Calculus of EVE . . . 24
3.6 The Extended Event Calculus of EVE . . . 25
3.7 Remarks . . . 25
3.8 Summary . . . 26

4 Theorem Proving with Abduction: SLDNFA+ **29**
4.1 Resolution: SLD . . . 29
4.2 Negation As Failure: SLDNF . . . 30
4.3 Constructive Negation . . . 33
4.3.1 SLDCNF . . . 33
4.3.2 SLDNF$^+$ . . . 33
4.3.3 Efficient Negation in SLDNF$^+$ . . . 35
4.4 Abduction: SLDA . . . 35
4.5 Negation and Abduction . . . 39
4.5.1 SLDNFA . . . 39
4.5.2 SLDCNFA . . . 39
4.5.3 SLDNFA$^+$ . . . 41
4.6 Remarks . . . 43
4.7 Summary . . . 43

5 The Event Calculus under SLDNFA$^+$ **45**
5.1 Planning by Theorem Proving . . . 45
5.1.1 Plan Analysis and Evaluation . . . 45
5.1.2 Plan Synthesis and Modification . . . 45
5.2 Abductive Predicates and their Maintenance . . . 46
5.3 Treatment of Negations . . . 47
5.4 SLD-Rule, Choice Points and Heuristics . . . 47
5.5 Execution of the Simple Event Calculus of EVE . . . 48
5.6 Execution of the Extended Event Calculus of EVE . . . 49
5.7 Summary . . . 50

6 SLDNFA$^+$ by Constraint Logic Programming **52**
6.1 The Oz calculus . . . 52
6.2 Resolution in Oz . . . 53
6.3 Negation As Failure in Oz . . . 55
6.4 Constructive Negation in Oz . . . 56
6.5 Abduction in Oz . . . 56
6.6 Remarks . . . 59
6.7 Summary . . . 63

7 The Constraint-Based Event Calculus **65**
7.1 Meta-Programming . . . 65
7.2 Heuristics . . . 65
7.3 Object-Oriented Abduction . . . 66
7.4 The default Abducibles and the Knowledge Base Interface . . . 67
7.5 Summary . . . 67

8 Object-Oriented Representation **68**
8.1 Representation of Properties . . . 68
8.2 The Planning Service Class: `EVE` . . . 68
8.3 The Event Class: `UrEvent` . . . 69
8.4 The Default Abducibles . . . 72
8.5 The Knowledge Base Object . . . 72
8.6 Summary . . . 73

9 EVE in the Multi-Agent Blocksworld Domain **74**

10 EVE in a Multi-Agent System **76**
10.1 The InteRRaP Architecture . . . 76
10.2 EVE within the Local Planning Layer . . . 78
10.3 EVE in the Loading Dock Domain . . . 79
10.4 Summary . . . 81

11 Related Work **82**
11.1 Planning with a logical framework . . . 82
11.1.1 Situation Calculus-Based Systems . . . 82
11.1.2 Planning using Temporal Logics . . . 82
11.1.3 Event Calculus-Based Systems . . . 82
11.2 Procedural, Nonlinear Planning . . . 83
11.3 Plan Graph Analysis . . . 83
11.4 Multi-Agent Planning . . . 83
11.5 Summary . . . 84

12 Future Research **85**
12.1 Planning in a Multi-Agent Architecture . . . 85
12.2 Extensions to the Event Calculus . . . 86
12.2.1 Maintenance . . . 86
12.2.2 Events with Duration . . . 86
12.2.3 Context-Dependent Conditions . . . 87
12.2.4 Probability in Planning . . . 87
12.2.5 Sensing Actions . . . 87
12.2.6 Hierarchical Planning . . . 87
12.2.7 Generic Resources . . . 88
12.3 Heuristics . . . 89
12.4 Abduction & constraint logic programming . . . 89
12.5 Summary . . . 90

13 Conclusion **91**

A EVE: Installation **93**
A.1 Dependencies and Compilation . . . 93
A.2 A Counting Example . . . 93
A.3 Summary . . . 95

B Axiomatisation of the Multi-Agent Blockworld **97**

C Axiomatisation of the Loading Dock Scenario **99**

List of Figures **104**

References **106**

Index **110**

1 Preface

Since its early beginnings, Artificial Intelligence research in the field of *planning* [32] has had to cope with the *frame problem* [44; 14]. The first logic-based systems using the SITUATION CALCULUS [17] showed that this *theory* of *time* and *action* has not been able to solve the frame problem efficiently, because, at least in its original formulation, it is restricted to a *linear* form of planning.

Later approaches, however, have shown that a theory originally designed for calculating database updates and narrative understanding, the EVENT CALCULUS [36], is capable of improving on these disappointing results, promising expressive *nonlinear* temporal reasoning, and thus nonlinear planning.

In this report, we first focus on the *clause*-based evolution from the SITUATION CALCULUS to the EVENT CALCULUS [30; 23] and derive two *axiomatisations* of the EVENT CALCULUS that are suitable for nonlinear planning using a common *action representation* based around *preconditions* and *postconditions* (Section 3). The development is guided by the exploration of *soundness* and *completeness* issues with respect to *strong* nonlinear plans as solutions.

The *theorem proof* procedure (Section 4) that underlies the theories of time and action has to supply *resolution* [38] and *negation as failure* [20] in order to *analyse* and *evaluate* plans. Plan *synthesis* and *modification* can be achieved through a form*hypothetical reasoning* introduced by the *abduction* principle [8; 41]. We therefore use a new proof procedure, SLDNFA$^+$, derived from a description in [25] that correctly merges the above three *inference* techniques with an extension based on *constructive negation* [7]. Its completeness concentrates on a *least commitment* class of *hypotheses* and solution *substitutions* with *clause programs* restricted to allow only *finite domain* variables within negated goals. The main problems that this algorithm is mainly faced with include the high interference between *nonmonotonic* negation and abduction. In addition, we demonstrate how to improve the treatment of negation in SLDNFA$^+$ for most calculi so that the finite domain requirement can be dropped. This holds, in particular, for proofs with the axiomatisations of EVENT CALCULUS we use. We investigate this matter closely with respect to the capabilities of SLDNFA$^+$ in Section 5.

Based on the implementation platform OZ [11], a *higher-order constraint-based* programming language offering many features, such as *concurrency, abstraction, encapsulated search* [4], and *object orientation*, we show how to realize the SLDNFA$^+$ procedure by syntactical *transformation functions* (Section 6) that turn clause-based programs into constraints [6]. The *selection rule* is hereby handed out to the reduction strategy of OZ and is can be influenced by many constructs of the language. Treatment of disjunctive *choice points* depends on the *driver* procedure that guides the encapsulated search process. Again, transformation techniques that guide the dynamic creation of choice points from within the *computation spaces* give us the power to implement different heuristics. We exemplary show such a mechanism within the *abduction step*.

A discussion of some special issues of our implementation, EVE [5], follows in Sections 7 and 8. EVE represents a generic planning module written in OZ that can be used in variety of Artificial Intelligence *architectures*. Object-oriented techniques help to model the necessary concepts in planning and furthermore provide comfortable extensions like *object-oriented abduction* and an improved constraint-based translation of the EVENT CALCULUS. Furthermore, the concurrent embedding of the temporal reasoning process is investigated.

After a first evaluation of the EVE system in the multi-agent blocksworld domain (Section 9), we explore its prototypical application within the agent architecture InteRRaP (Section 10). Multi-agent systems are usually found in domains that have real need for strong nonlinear planning services as offered by EVE. The

layered architecture InteRRaP deals with temporal reasoning on several levels of abstraction but the focus of this section is the evaluation of EVE in the local planning layer. We use the loading dock scenario, the main research environment of InteRRaP, to discuss this initial integration.

A comparison between EVE and related planning approaches with respect to efficiency and expressiveness (Section 11) follows and is continued with the presentation of the future research, especially in the field of *distributed*, *multi-agent* planning. Besides the architectural considerations within multi-agent systems, we describe possible extensions to the Event Calculus, an approach based *generic resources*, and *hierarchical planning* through *plan abstraction* (Section 12). Furthermore, some ideas that try to bring the notion of abduction and constraint-based logical programming more closely together are described. A conclusion in Section 13 summarises the overall contribution of our work and an appendix gives a short installation guide (Section A) and presents the axiomatisations of the scenarios we used (Sections B and C),

2 Introduction

2.1 Logic

Logic can be perhaps best defined as a class of *truth* languages. Major *syntactical* units, the *formulae*, denote (*represent*) truth values and from these values we construct an *equivalence semantics*, $\models\!\models$. Whereas *classical* logic restricts to the common truth values **true**/$\top$ and **false**/$\bot$, there exist today many extensions allowing different conceptions of truth (*four-valued* logic, *fuzzy* logic, etc.).

Prime syntactical elements of a logical language are *propositions* that act like *statements* uttered by humans, and logical *connectives*, e.g., *conjunction* $\wedge$, *disjunction* $\vee$, *equivalence* $\equiv$, *implication* $\supset$, *negation* $\neg$, etc. The semantics of the connectives is reasonably defined by in terms of the *meta-logic* level, e.g., $F_1 \supset F_2 \models\!\models \top \Leftrightarrow F_1 \models F_2$.

What makes logic so attractive from a computational point of view is that, once given a *procedure* that investigates the truth of a formula (*proof* - $\vdash$), this does not only yield a way of programming but also some machinery that allows to make *inferences* close to human *reasoning*. It is not obvious to assume the existence of such algorithms and much research has been spent to finally obtain the encouraging result that truth within the propositional and first-order logic framework is computationally observable ($\vdash\!\Leftrightarrow\!\models$). Depending on the expressivity of logical languages however, restrictions to this statement (like on *decidability*) are unavoidable.

Provable formulae are called *theorems* of the appropriate proof algorithm. A way to classify theorem proof procedures is via *soundness* and *completeness*. Soundness, i.e., *correctness*, guarantees that a formula confirmed by the proof scheme is indeed a *valid*, i.e. true, formula, ($\vdash\!\Rightarrow\!\models$). Completeness is only reached if every valid formula is also a theorem ($\vdash\!\Leftarrow\!\models$). In contrast to validity, one speaks of an *inconsistency*, if a formula denotes $\bot$.

The logic best explored so far is *first-order logic* or *predicate logic*. *Entities* (members) of a denoted *domain* or *universe* (set) are described by *atoms* (basic entities), *terms* (functions over entities), and *variables* (representatives for entities). The already introduced propositions are expressed by *predicates* that represent *relations* over entities. Finally, the combination of propositions with the logical connectives forms the class of first-order logic formulae. A special class of connectives remains to explain: *universal*($\forall$) and *existential* ($\exists$) *quantifiers*. *Quantification* allows extended formulae involving variables and introduces the abbreviations $\breve{\forall}F, \tilde{\exists}F$ that describe the quantification of all *free* variables of the *sub formula* F, i.e., variables that are not in the *scope* (any sub formula) of some quantifier regarding F alone.

The universe and an appropriate *interpretation* of the atoms (into the members of the universe), the terms (into the functions over the universe), and the predicates (into the relations over the universe) build a *structure*. A structure is linking syntax (formulae) to semantics (truth values) by defining *solutions* of a formula as the set of replacements of variables with entities that render the formula true. If the solution set of a formula covers all possible replacements, the structure is said to be a *model* of the formula. A formula is called *satisfiable*, if there exists a structure such that the solutions are not empty. Given a set of *closed* formulae, i.e., containing no free variables, one speaks of a *theory*. It can also be equivalently described by the behaviour of its solutions in all structures.

Sound and complete proof procedures for first-order logic have been presented. The computational power is equivalent to the *Turing Machine* concept. *Undecidability* results from the *Halting Problem* therefore carry over to predicate logic. In this report, we use a normalised *clause* syntax (Figure 1 uses *Backus-Naur Form* to describe it), derived from the *completion* form [20], to present the *calculi* (calculus=abstract computation flow), to describe the proof procedures and

variables	$V : \mathtt{X}, \mathtt{Y}, \ldots$	
constants	$A : \mathtt{a}, \mathtt{b}, \ldots$	
terms [a]	$s, t : A(\bar{s}), A, V$	
predicates	$P : A(\bar{s}), s{=}t, \top, \bot$	
conjunctions	$K, I : K \wedge I, P$	
goals	$D, E : D \vee E, \exists \bar{V}\,(K)$	$\tilde{D}, \tilde{E} : \tilde{D} \vee \tilde{E}, \exists \bar{V}\,(K), \neg\exists \bar{V}\,(K)$
clauses	$C : \forall \bar{V}\,(A(\bar{V}) \equiv D), \top \equiv D$	$\tilde{C} : \forall \bar{V}\,(A(\bar{V}) \equiv \tilde{D}), \top \equiv \tilde{D}$

[a] $\bar{s}$ is the type of tuples over type s. Depending on the context of s, this could also denote concatenation, conjunction, or even a set.

Figure 1: Normalised Clauses C,$\tilde{C}$

to define *transformation functions* on. Clause programs are conjunctions over $C, \tilde{C}$ with each clause having a different *head* predicate (left part of the equivalence $\equiv$). Variables never occur free; they have to reside in the scope of some quantifier. Undefined predicates are represented by clauses with $\bot$ as the *body* (right part of the equivalence $\equiv$). In general, one assumes that the set of the atom, function, and predicate symbols is infinite.

With negation, $\tilde{C}$, clause syntax is nevertheless as expressive as first-order logic since *equivalence transformations* exist [22]. These transformations are rewritings of formulae that preserve their semantics and thus form an important part of proof algorithms. Procedures that are designed to work specifically with clause programs require a special form of query, because they decide about the truth of $P \supset G (\Leftrightarrow P \models G)$ for a given program P (also called theory) and a *goal* $G = (\top \equiv D)$. Examples are SLD (*resolution*) and SLDNF (including negation treatment). Extending the principles to *hypothetical reasoning*, SLDNFA$^+$ even allows the generation of *fact* clauses $Ab : \forall \bar{V}\,(\, A(\bar{V}) \equiv \bigwedge_{\mathtt{X} \in \bar{V}} \mathtt{X}{=}s \,)$ such that $(P \wedge Ab) \models G$.

To obtain the functionality of programming languages that compute output values, proof procedures are usually not confined to verifying the existence of entities which satisfy a goal. These algorithms also return information about the form of the entities that satisfy the goal G. An important concept is *equality* on the syntactic (the predicate =) as well as on the semantic level. Theorem proving handles the equality semantics via *substitutions*, replacements of variables with terms, and *unification*, a way of producing *unifying substitutions* that render the given terms S and t equal. The orthogonality of substitution and equality can be seen from the fact that for each satisfiable conjunction of equality terms, there exists at least one substitution which, when applied to the formula, turns it into a conjunction of trivial equalities $\bigwedge s = s$. Furthermore, each substitution represents per se a formula $\bigwedge \mathtt{X}{=}s$; its application to F is equivalent to the conjunction $F \wedge \bigwedge \mathtt{X}{=}s$.

Soundness and completeness can now be extended to the answer substitution set. This set is in general infinite, but many of the answer substitutions share a

common structure. Thus the introduction of the *most general* unifier (MGU) that forms a base for the answer substitution space of a unification improves the computational result without losing completeness, provided that the proof procedure is *fair*, i.e., it regularly expands competing parts (introduced by disjunctive *choice* and conjunctive *selection*) of the proof. Only minimal, structurally different solutions are generated with the MGU.

To describe a *denotational* semantics of first-order logic in a uniform way, formulae are associated with an appropriate *transition* function over ordered structures. Fix-points of the function turn out to be the models of the formula and the notions of *monotonicity* and *non-monotonicity* are derived from the transition function to the appropriate connectives and reasoning principles.

Modern approaches try to integrate the logical paradigm into all sorts of computational tasks instead of leaving it as a playground for mathematicians and logicians. Efficiency and flexibility is mainly gained by restrictions to a logical sub language or a special domain. One such approach is the use of *constraint logic systems*, where the basic formulae are logical *constraints*.

Of course this section is not intended to fully[1] introduce the interested reader into the world (or better universe) of logic that has been developed over 2000 years. It only presents the key concepts that are used throughout this report and motivate an important part of the approach that is used in EVE. For a complete overview about first-order logic and its realisation through automation, [27] may be a good choice. A perspective on constraint logical programming and its declarative semantics can be found in [13]. This paper also extends the unification process to rational trees and develops an appropriate abstract machine that covers most of its concepts. These concepts are also the base for the Oz calculus, the constraint-based implementation platform of EVE.

2.2 Planning

Computational research in *planning*, or *general problem solving* (GPS), started in the late 50's [17]. Since then, it has grown to a rather large discipline in *Artificial Intelligence* research. GPS systems aim to arrange parts of a given *resource specification* into a *plan* that meets a given *goal* (Figure 2). With todays knowledge in *complexity theory* it is clear that solving this 'problem of problem solving' is necessarily NP-hard[2] and thus *exponentially* worse. Such a general approach will therefore only be able to work efficiently in certain classes of planning *domains*[3] The more general the algorithm, the more efficiency has to be incorporated from additional domain-specific knowledge, so called *heuristics*, to guide the *search* for the solution plan. These guidelines are a general tool borrowed from *graph theory* and help to find optimal *paths* in a search *tree* based on the *cost* of the already collected paths and an underestimation of the cost to the nearest goal *leaf*.

Whereas *scheduling* problems focus on possibly optimal, temporal coordination of the given resources, common planning systems are faced with a more general formulation born in *means-end* analysis research. Starting with the concept of a *state* or *situation* usually represented in logic, (e.g., clause-based as in Figure 1), a plan consists of the collection and arrangement of *actions* or *events*. The *types* of the events are also defined in logic in terms of *conditions* that each instance 'places' onto

[1] and thus accurately, as the description of several expressions is vague because of some lacking details.

[2] It is straightforward to reduce the *travelling salesman* task to a planning problem.

[3] There are several expressions, such as domain, model, soundness, completeness, goal, etc., that appear in this report in several, slightly different meanings due to the logical foundation of planning. Indeed in the approach of EVE (see Section 2.3) that has a planning theory on top of a theorem proof layer, there is even a strong dependency between their meaninings. The actual meaning, however, should be clear from the respective context.

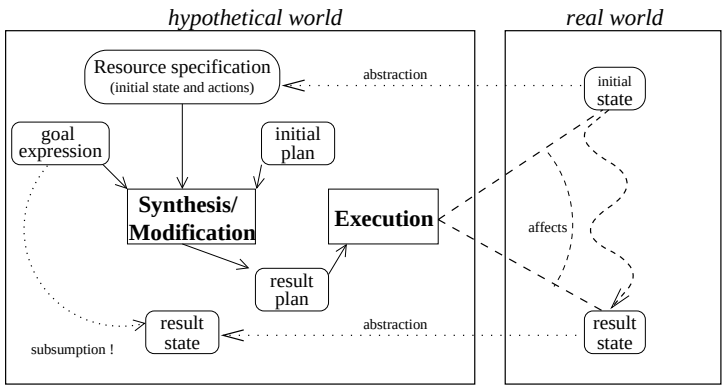


Figure 2: The Task of Plan Generation

the state representation, if it is *executed*. Solution plans for the planning problem are those that turn the *start* situation into some situation that is satisfying the given logical goal expression. The resource specification in the planning case is given by a combination of the start situation and the collection of the allowed action types which build the planning domain description[4]. Pure *plan synthesis*, or *planning from scratch*, as shown in Figure 2 starts with an empty plan description — *modification* receives an input plan to complete and is computationally just as complex [3]. This is not surprising as NP-hard, exponentially worse problems do not allow the assumption that each solution for a *subgoal* is extendable to solve the whole task.

Situations are often represented by a *knowledge base* (KB) whose prime logical elements are called *facts, propositions* or *properties* analogue to logical terminology. Based on the propositional syntax used in the KB, the conditions of event types can now be distinguished into *preconditions*, properties that have to be *subsumed*, i.e., implicated, by the situation at execution time to yield *success* of the execution, and *postconditions* or *effects*, properties that have changed, i.e. they are deleted or retracted to obtain the result situation of execution. To gain expressivity, action types can carry *roles* that are parameters of their conditions. Besides establishing a type *hierarchy* (from the type with the least to the type with the most specified roles), roles further extend a plan to specify their instantiations. Also the goals are defined in terms of the propositions from the KB which makes the success of planning dependent on subsumption. Examples of a situation, an action type, a goal and a plan are given in Figure 3.

A finite representation can only handle finitely many details. The result of this fact comnbined with the complexity of the real world is called the *qualification problem*, which states that for each solution plan there are still uncountably many exceptions that may lead to failure of its execution.

Planning systems can be classified in several ways. First, soundness and completeness with respect to solution plans are an important means of classification. Soundness guarantees in this case that the generated solution plans are correct with respect to the goal and the chosen execution model. Completeness is only guaranteed if the planning procedure is capable of generating all solution plans for the given goal that are sound. It depends on the cost, here the relative amount of execution effort, of the events whether solution plans are optimal. If no such cost information is available, the plan with the least number of steps is, of course, the

[4] As you will see in section 3, this is a weak distinction. Depending on the perspective of different theories, there are intersections between the notions of a plan, the start situation, and the domain description.

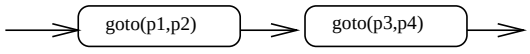
Situation	`standing(p1)` $\wedge$ `free(p2)` $\wedge$ `free(p3)` $\wedge$ `reach(p1,p2)` $\wedge$ `reach(p2,p3)`
Action Type	`goto(From,To)`
Preconditions:	`standing(From)` $\wedge$ `free(To)` $\wedge$ `reach(From,To)`
Postconditions (added):	`standing(To)` $\wedge$ `free(From)`
Postconditions (retracted):	`standing(From)` $\wedge$ `free(To)`
Goal	`standing(p3)`
Plan	

Figure 3: Situation, Action Type, Goal, and Plan

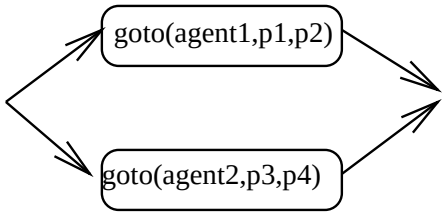


Figure 4: Nonlinear Multi-Agent Plan

prime candidate for the optimum.

Another distinction is introduced by the kind of treatment of the goal. Systems that broadly expand the start situation by applying actions and in this way try to reach the goal are called *forward* planning systems. Other approaches prune the state space by a goal-driven expansion — *backward* planning. Both cases are unified if one chooses the space of plans rather than the space of states as the base for the search. The choice between forward and backward planning is then a question of heuristics.

The concept of time also divides planning approaches into several classes. Whereas plans with a serial execution concept are called *linear*, there are application domains, such as *multi-agent* domains (see Section 10), in which it is convenient to have *nonlinear* plans (see Figure 4) with *concurrent* or even *parallel* actions. Nonlinearity is obviously more general than linearity and, as we will explain in Section 3, it can be modelled by *incomplete* knowledge about the temporal ordering of events. A further advantage of nonlinear to linear planning is the reduced search space of plans. Each nonlinear plan is a representative of its *linearisations* (completion to a total, temporal order) and the planning process tries to keep the number of temporal restrictions as small as possible. The opportunities for conflicts between nonlinear branches, however, are increased. It depends on the appropriate planning domain whether a linear or a nonlinear approach will be the better choice.

Sound nonlinear plans could be characterised by having at least one sound linearisation (*weak* nonlinearity). *Strong* nonlinearity goes even further and demands that all possible linearisations of a correct nonlinear solution be sound in the linear meaning.

Adaption of human *abstraction* methods leads to *hierarchical* extensions. Planning problem solvers that support such methods do not only allow actions as primitives for plans. They also allow whole plan structures as events as presented in

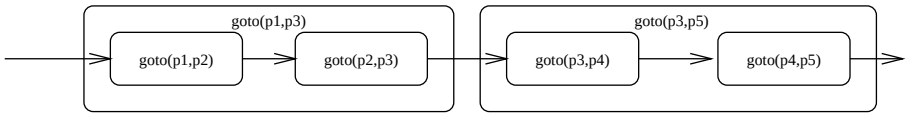


Figure 5: Hierarchical Plan

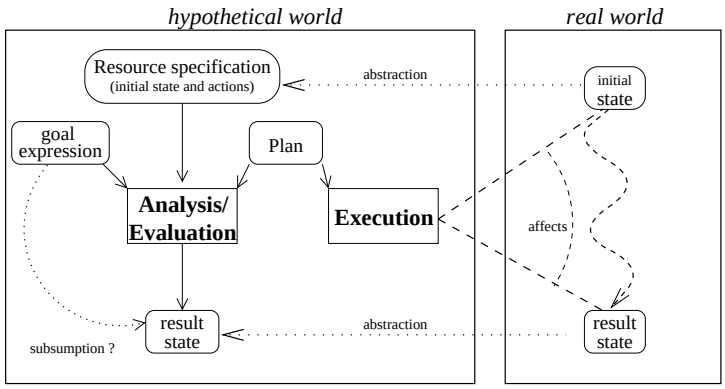


Figure 6: The Task of Plan Analysis

Figure 5. This can be accomplished either by a concept of plan abstraction, as Section 12.2 explains, or by a planning process that is carried out in several stages according to goal hierarchy (situation abstraction).

Finally, planning procedures differ in the programming paradigm they use. Early systems often relied on procedural algorithms that are implementable in most programming languages. The approach of logic programming however is to use a logic itself as the basis for a declarative programming language. It therefore requires some theorem proof procedure as an 'interpreter', but key concepts, such as unification, are already defined and the resulting framework tends to be more flexible and extendable. With EVE's approach, it is even possible to obtain plan *analysis* and *evaluation* (see Figure 6) from the same underlying framework. Together with *modification*, these tasks cannot be solved with most fixed procedural approaches without much rewriting effort.

2.3 EVE's Architecture

From a computational point of view, the *architecture* of the planning module (Figure 7) described in this report, EVE, can be conceptually divided into three layers due to its logical approach. A theory of time and action, the Event Calculus, resides on the top level (layer 1) and is designed to handle incomplete temporal knowledge and thus reasoning about nonlinear plans (see Section 3).

Since EVE provides two versions of the calculus with different computational costs (simple and extended), their execution (proof) with an underlying theorem proving procedure (layer 2) is parametrisable in several ways. This layer supports resolution, negation, and abduction in order to obtain plan analysis as well as synthesis (see Sections 4 and 5). In particular the nontrivial dependency of soundness/completeness of the theory layer (with respect to solution plans) and of the proof layer (with respect to goals, solution substitutions and hypotheses) will be investigated in Section 5.1). Finally, the implementation base (layer 3) for these two upper levels is represented by the Oz calculus [11;

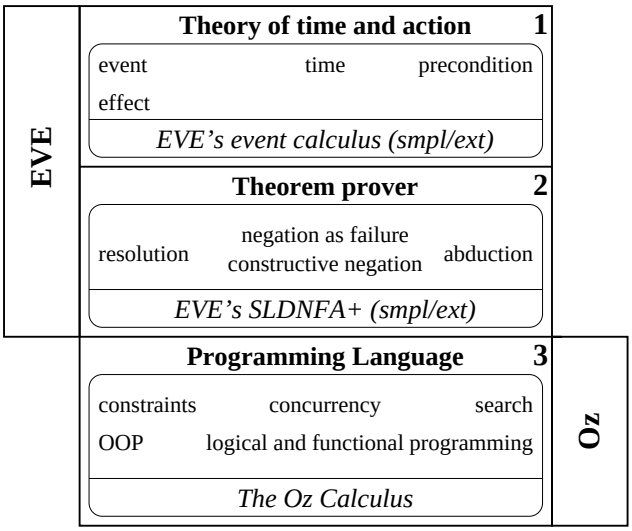


Figure 7: Computational Architecture of EVE

12], realised in EVE as a higher-order, constraint-based abstract machine supporting *concurrency*, *object orientation*, and *encapsulated search*.

As our approach has been to use as many logical resources from this constraint calculus as possible, layer 2 is totally subsumed in the result of the sound transformation functions (Section 6) that link the layers 1 and 3. There are furthermore many features, such as object orientation, that improve customisation and the embedding of EVE into various applications (Sections 7 and 8).

`initiates(pickup(Agent,Block),Term)` $\equiv$ `Term=holding(Agent,Block)`
`terminates(pickup(Agent,Block),Term)` $\equiv$ `Term=handempty(Agent)` $\vee$ `Term=ontable(Block)`
`precondition(pickup(Agent,Block),Term)` $\equiv$ `Term=handempty(Agent)` $\vee$ `Term=ontable(Block)`

Figure 8: Action Type Definition/Planning Domain Axiomatisation

3 Theory of Time and Action: the Event Calculus

We first present the basic properties that a theory of time and action has to provide to allow reasoning over collections of actions and thus be suitable for planning purposes. Starting with the well-known Situation Calculus, it then progresses chronologically, guided by the investigation soundness and completeness issues, to describe two versions of the Event Calculus that are suited for nonlinear planning. The logical syntax employed is a slightly looser but still equivalent one for clause programs of Figure 1. The axioms presented rely furthermore on some predefined terms ($|$ in *infix* notation, $\top^*$[5], $\bot^*$, `nil`) and predicates (`member()`, $\neg^*()$[6]) to incorporate *lists* and *boolean* arguments.

3.1 Requirements

The following considerations describe the basic skills that a theory of time and action has to preserve:

- **Representation of property, situation, action types, actions and time:** A theory suitable for logical planning has to define the concepts of properties, situations, action types, actions as their instances, and time. Whereas properties and action types are immediately defined in terms of logic, thus forming the planning domain axiomatisation (see Figure 8), the representation of situation and time is not necessarily *explicit*. A plan consists of action instance choices and their temporal order.

- **Sound and possibly complete reasoning over plans.** Assisted from a theorem proof procedure, the used theory T should be able to correctly (with respect to its execution model) calculate the overall effects E of a given plan P in the planning domain D:

  $$T \wedge D \wedge P \vdash E$$

  Given some special effect as a goal, it should be furthermore able to accept as many as possible plans that establish this effect.

- **Nonmonotonicity:** State transition caused by action execution is nonmonotonic, because properties can be added as well as retracted. Techniques to circumvent inconsistent representations are necessary. Closely related is the task of dividing the properties into the ones that change and the ones that remain untouched, the so-called *frame problem* [14; 44] this is manageable using techniques for nonmonotonic reasoning[9].

[5] These are special atoms that represent a *worst case* flag (see Section 3.6). They have to be distinguished from the 0-ary predicates $\top$ and $\bot$.

[6] This is a special, unary predicate to invert the value of a worst case argument. It has to be distinguished from the connective $\neg$.

- **Incomplete temporal knowledge:** Nonlinearity can be modelled by partial temporal order and thus we have incomplete knowledge. Whenever there is no temporal relation between two actions, they should be executable in any order. With strong nonlinearity, they are even ideally allowed to interleave each other introducing concurrency or parallelism. A nonlinear theory of time and action has to cope with this lack of information and nevertheless guarantee soundness with respect to its execution model. *Worst case assumptions* therefore play an important role.

- **Sound and possibly complete hypothetical reasoning:** The solution plan P for goal E has to be constructed as a hypothetical result of a proof, starting from some initial P_1 with $P \supset P_1$:

  $$T \wedge D \wedge P_1 \vdash_P E$$

  Again, the treatment of incomplete knowledge is required since P is derived incrementally and proof steps that were taken when P is partially constructed may later have to be retracted. Solutions should furthermore cover a reasonable range of correct plans.

- **Knowledge base:** Finally, a KB containing the start situation to the planning problem has to be interfaced. Integration of the propositions collected there depends on the planning theory and the planning domain representation.

3.2 The Situation Calculus

The Situation Calculus has been one of the first paradigms in planning to solve the frame problem correctly [17]. Since its original formulation had some drawbacks with respect to the representation, we present a revised proposal from [35] that is more elegant and allows better comparison with the theories we present later. Furthermore, the axiomatisations are in accordance with the concept of preconditions being a function on action types within the planning domain description.

The basic idea of the Situation Calculus (Figure 9) is the representation of explicit situations `S` and their properties `P` through the `holds` predicate: `holds(P,S)`. Following the action type definition scheme from the beginning of this section, explicit events are associated with a type (`act()`). Only time is left as being defined *implicitly* by the order of the situations. Reasoning about properties is therefore situation-driven. If a situation `S` subsumes the preconditions of the type `Ty` of an event `E`, the application of `E` to the situation will succeed and produce a result situation: `result(S,E)`.

The explicit situation representation performs the nonmonotonic state transition by coupling the propositions with the situation terms. The solution to the frame problem, the *frame axiom*, is expressed using negation to decide about property propagation from situation to situation (Figure 10). Negation can even simplify the definition of the precondition check within the `fails()` axiom.

This formulation, however, is not capable of dealing with incomplete temporal knowledge. Holes in the plan structure, i.e. undefined intermediate situations, prevent any reasoning about following states. Plan analysis as well as plan synthesis are therefore restricted to linear planning using a *linearity assumption*. This assumption postulates that conjunctive goals have to be solved in a sequence of complete subgoal solutions. Optimal plans, however, require interleaved subgoal treatment, as in the case of the famous *Sussman Anomaly*.

An interface to a knowledge base is easily provided as an equivalence to the start situation.

$$\forall P,S,E\ (\ \texttt{holds}(P,\texttt{result}(S,E)) \equiv$$
$$\exists Ty(\texttt{holds}(P,S) \wedge \texttt{act}(E,Ty) \wedge \neg\texttt{terminates}(Ty,P)) \vee$$
$$\exists Ty(\texttt{act}(E,Ty) \wedge \texttt{initiates}(Ty,P) \wedge \neg\texttt{fails}(E,S)))$$

$$\forall S,E\ (\ \texttt{fails}(E,S) \equiv$$
$$\exists Pd,Ty\ (\texttt{act}(E,Ty) \wedge \texttt{precondition}(Pd,Ty) \wedge \neg\texttt{holds}(Pd,S)))$$

Figure 9: The Situation Calculus

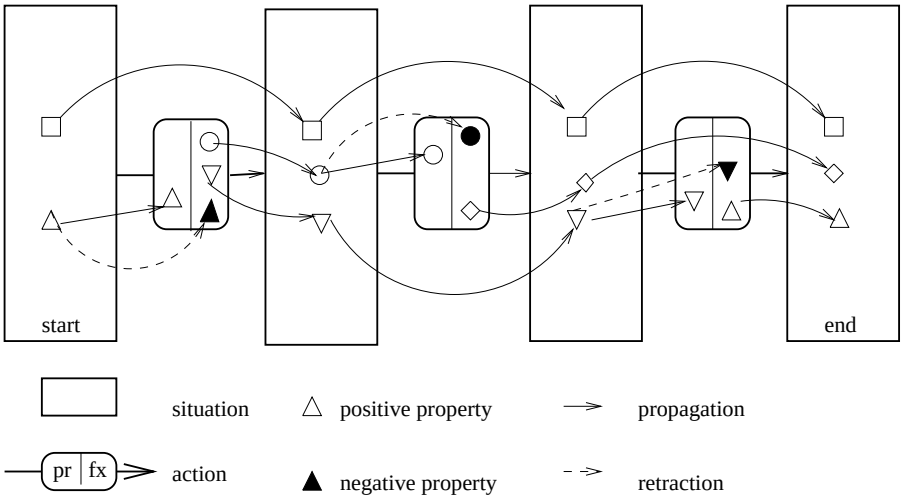


Figure 10: Propagation of Properties in the Situation Calculus

Although it is not always obvious, the Situation Calculus is present in many planning approaches including STRIPS. Typical characteristics of these systems are their solution to the frame problem and the linearity assumption. Even linear *regression* techniques that describe situation classes rather than concrete situations in backward planning belong to these systems.

3.3 The Event Calculus by Kowalski & Shanahan

A far more flexible approach to the frame problem is provided by a theory originally designed for maintenance of temporal databases and narrative understanding: the Event Calculus [36]. An adaption of the theory to the planning problem and the commonly used representation of actions is presented in [30] (Figure 11).

The focus is no longer on an explicit representation of situations, but on action type instances, that is the actions or events introduced by `happens()`. Furthermore, there are terms describing points in time (`start()`,`end()`) with respect to these events and a predicate denoting their temporal relation (`before()`). Within this this report, the simplified assumption of events having ideally no *duration* is used (`start(E)`$\approx$`end(E)`), which is a common assumption throughout the literature.

The introduction of time points supports the task of modelling the change of state (Figure 12) that is computed by a search for possible *initiators* of properties and a more general solution to the frame problem: the *persistence* axiom represented in terms of the `clipped()` relation. Again, negation is a key technique that allows us to formulate the following idea in a logical background:

A property holds true at a certain time point `Tp` if it is introduced by

∀P,Tp (holds(P,Tp) ≡ ∃E,Ty (happens(E) ∧ act(E,Ty)∧ initiates(Ty,P) ∧ before(end(EI),Tp)∧ ¬fails(E) ∧ ¬clipped(P,end(E),Tp)))
∀E (fails(E) ≡ ∃Ty,P (act(E,Ty) ∧ precondition(Ty,P)∧ ¬holds(P,start(E))))
∀Tp1,Tp2,P (clipped(P,Tp1,Tp2) ≡ ∃E,Ty (happens(E) ∧ act(E,Ty)∧ terminates(Ty,P) ∧ ¬fails(E)∧ in(end(E),Tp1,Tp2)))
∀Tp1,Tp2,Tp3 (in(Tp1,Tp2,Tp3) ≡ before(Tp2,Tp1) ∧ before(Tp1,Tp3))

Figure 11: The EVENT CALCULUS by Shanahan

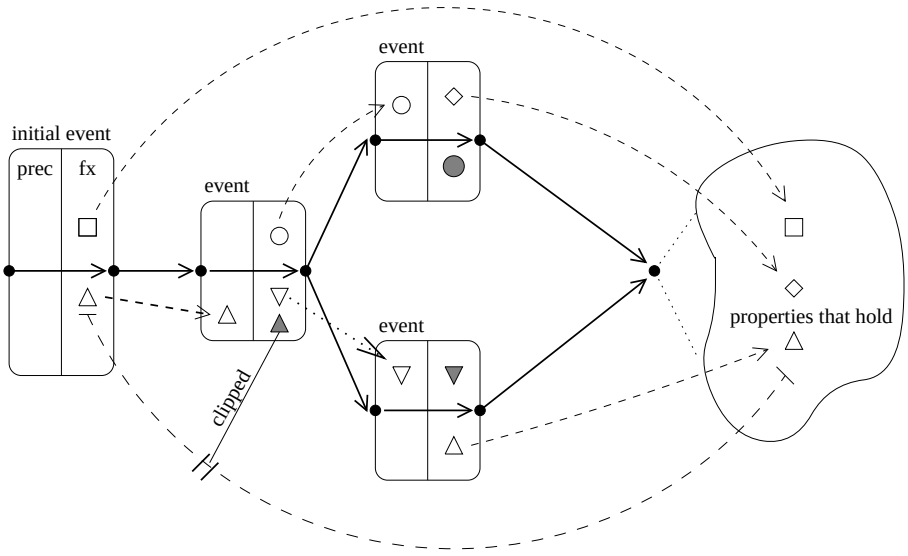


Figure 12: Propagation of Properties in the EVENT CALCULUS

some successful event `E` known to happen before `Tp`, and if it is not terminated by some other successful event between `end(E)` and `Tp`.

Compared with the SITUATION CALCULUS, this is an equivalent approach to linear planning. However, persistence further extends the original frame axiom by reasoning about incomplete temporal knowledge and thus partially ordered events in nonlinear planning.

But this version of the persistence axiom has a problem with strong nonlinearity. It recognises only those persistence *destroyers* that are provable to be inside (`in()`) the time interval within which the property must be preserved. Further completions of the temporal order could thus add further such destroyers. The solution plans are therefore characterised as being extendible to a correct, linear plan in at least one way (weak nonlinearity). Strong nonlinearity as it is required in EVE's intended application domains (see Section 10) only accepts those plans that are correct in all possible temporal extensions (Figure 13).

When proceeding from a situation-based to an event-based representation, the

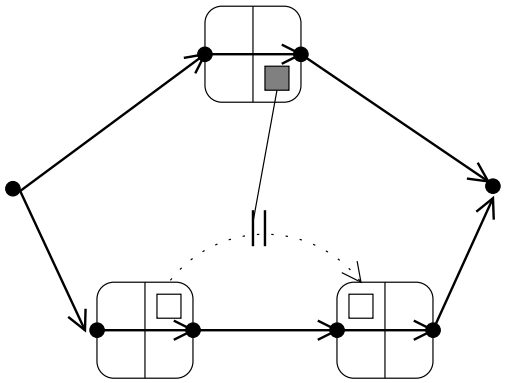


Figure 13: Strong Nonlinear Incorrectness

∀P,Tp (holds(P,Tp) ≡ ∃E,Ty (happens(E) ∧ act(E,Ty)∧ initiates(Ty,P) ∧ before(end(E),Tp)∧ ¬fails(E) ∧ ¬clipped(P,end(E),Tp)
∀E (fails(E) ≡ ∃P,Ty (act(E,Ty) ∧ precondition(Ty,P)∧ ¬holds(P,start(E))))
∀Tp1,Tp2,P (clipped(P,Tp1,Tp2) ≡ ∃EC,TyC (happens(EC) ∧ act(EC,TyC)∧ terminates(TyC,P) ∧ ¬fails(EC)∧ ¬out(end(EC),Tp1,Tp2)))
∀Tp1,Tp2,Tp3 (out(Tp1,Tp2,Tp3) ≡ before(Tp1,Tp2) ∨ before(Tp3,Tp1) ∨ Tp3=Tp1)

Figure 14: EVENT CALCULUS by Missiaen

interface to a knowledge base is no longer straightforward. The metaphor of events can nevertheless be kept because of a special `initial` action associated with a special type `initial`* that is able to initiate all properties contained in the KB.

3.4 The EVENT CALCULUS by Missiaen

A reformulation of the persistence axiom is given in [23]. Again, negation plays an important role in the sound treatment of incomplete temporal knowledge :

> A property holds true at a certain time point `Tp` if it is introduced by some successful event `E` known to happen before `Tp`, and if it is not terminated by some other successful event that is not provable to be outside the interval specified by `end(E)` and `Tp`.

Persistence destroyers are no longer events provable to reside inside the appropriate time interval. They are rather defined as not being provable to happen outside (`out()`) the given range (Figure 14). The reasoning ranges, therefore, over all possible linearisations of the current, incomplete plan. A certain event is not able to destroy the persistence of its own preconditions, thus the assumed identity of `start(E)`≈`end(E)` is only approximately true. The definition of `out()` handles this fact by operating on an interval which is open to its right.

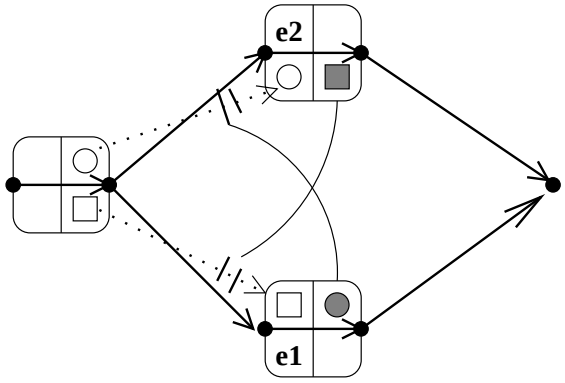


Figure 15: Nontermination

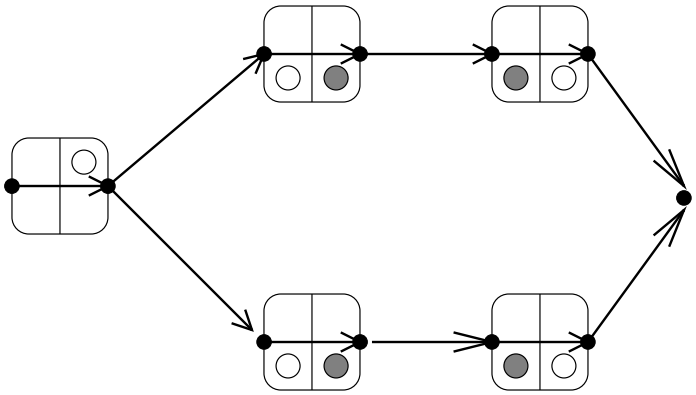


Figure 16: A Not Strong Nonlinear Solution

Reasoning about nonlinear structures is now sound even for the stronger notion of nonlinearity but is faced with a *termination* problem that is shown in Figure 15. Both events e_1,e_2 are mutual destroyers of each others preconditions. A proof of $\neg$`fails(`e_1`)` thus depends on $\neg$`fails(`e_2`)`, and vice versa, causing an infinite loop within the computation. The situation of two interacting and concurrent actions can occur both in plan analysis and plan synthesis tasks. Initial attempts to solve the problem tried to recognise and linearise the events involved. This requires some periodical check together with a decision about which linearisation to take.

The solutions that we propose for EVE try to put this knowledge into the calculus itself, thus providing a 'natural' treatment. In each case, however, the sound result for strong nonlinearity is achieved through a loss of some solutions (Figure 16). In this example, each linearisation of the plan is correct, but the property '$\circ$' does not provably hold at the end of the plan. It is common in the literature to exclude such cases from the class of strong nonlinear solutions. A characterisation of this class of solutions is yet to be formulated, but it seems like they require several identical event streams. These special cases are not likely to be very important in practical planning domains.

∀P,Tp (holdstrue(P,Tp) ≡ ∃EI,TyI (happens(EI) ∧ act(EI,TyI)∧ initiates(TyI,P) ∧ before(end(EI),Tp)∧ ¬fails(EI) ∧ ¬clippedtrue(P,end(EI),Tp)))
∀P,Tp (holdsfalse(P,Tp) ≡ ∃ET,TyT (happens(ET) ∧ act(ET,TyT)∧ terminates(TyT,P) ∧ before(end(ET),Tp)∧ ¬fails(ET) ∧ ¬clippedfalse(P,end(ET),Tp)))
∀E (fails(E) ≡ ∃Ty,PD (act(E,Ty)∧ (posprecondition(Ty,PD) ∧ ¬holdstrue(PD,start(E)))∨ (negprecondition(Ty,PD) ∧ ¬holdfalse(PD,start(E)))))
∀Tp1,Tp2,P (clippedtrue(P,Tp1,Tp2) ≡ ∃EC,TyC (happens(EC) ∧ act(EC,TyC) terminates(TyC,P) ∧ ¬out(end(EC),Tp1,Tp2)))
∀Tp1,Tp2,P (clippedfalse(P,Tp1,Tp2) ≡ ∃EC,TyC (happens(EC) ∧ act(EC,TyC)∧ initiates(TyC,P) ∧ ¬out(end(EC),Tp1,Tp2)))
∀Tp1,Tp2,Tp3 (out(Tp1,Tp2,Tp3) ≡ before(Tp1,Tp2) ∨ before(Tp3,Tp1) ∨ Tp3=Tp1)

Figure 17: The Simple Event Calculus of EVE

3.5 The Simple Event Calculus of EVE

We first present a very simple change to the Event Calculus from Section 3.4 to avoid nontermination. It is based on the investigation that actions that are allowed to fail should not necessarily appear while planning from scratch. Whenever the planner introduces some event `E`, there is always the intention to establish a certain goal that afterwards depends on the success of `E`. Furthermore, if the plan execution mechanism does not involve any precondition check and always tries to execute some portion of the action until it gets a failure report back, the failed events cannot just be ignored because some of their effects could have taken place. Solution plans therefore should be restricted respectively.

To incorporate these considerations into the calculus, it is necessary to implement a worst case assumption about the behaviour of actions. It is certainly obvious that a failed action is not guaranteed to introduce any property the planner likes to have installed. However, it is still able to serve as a possible destroyer of persistence. Our redefinition of the persistence axiom therefore omits the destroyer success check. This strategy is reasonable as analysis (evaluation) appropriately restricts the accepted class of plans and synthesis (modification) tries to push even failed events out of a possible destroyer role. In the case of Figure 15, a linearisation takes place in order to derive the success of e_1 or e_2.

Figure 17 presents the derived calculus together with the additional handling of *negative* properties that are now dealt with symmetrically to the *positive* ones. Preconditions can be distinguished into positive preconditions (`pospreconditions()`) that should hold (`holdstrue()`) at the start time of execution and negative preconditions (`negpreconditions()`) that should not hold (`holdsfalse()`) to prove the success. Negative properties can even be accessed in the start event/knowledge base. Knowledge bases that contain purely positive facts can be accessed via negation (*as failure*).

As Section 5.5 discusses, our formulation is even suited to simplify complicated parts of a proof procedure that otherwise runs into problems caused by nonmono-

tonicity. Applicable domains, however, are restricted to those that do not rely on disablement of actions, e.g., *cooperative* domains and planning from scratch tasks.

3.6 The Extended Event Calculus of EVE

A restriction to cooperative planning or planning from scratch as in the case of EVE's simple calculus is not always wanted, e.g., in negotiation planning or generally non-cooperative domains. Since it could be of use to intentionally render the preconditions of an action unsatisfied in order to get rid of the unwanted effects of this action[7], a success check within the `clipped()` axiom would still be needed. With this support, the calculus is able either to render unwanted events unexecutable or at least to recognise their failure.

As we have already mentioned, the infinite-loop situations should be recognised and solved within the calculus itself. A detection is easy to accomplish by the addition of a *list* parameter `L` to `fails()`, `holds()`, and `clipped()`. This parameter collects the events in whose nested success check the current proof branch is residing. If `fails()` is recognising events that already contained in `L`, this will indicate the infinite-loop situation.

What is the appropriate action on detection of a loop? The worst case assumption from our simpler approach has to be slightly modified in order to let the axiomatisation do the work. If one introduces a parameter `W` that indicates the worst case either to be the success (`W=`$\top^*$) or the failure (`W=`$\bot^*$) of the current `fails()` call, the proof of the clause could just behave accordingly ($\top^* \approx \top \quad \bot^* \approx \bot$). Since each negation sign inverts the worst case, the parameter also swaps its value ($\neg^*$ `(W)` — see Figure 18).

The initial values for these two new parameters are obvious. `L` is `nil` as the calculus has a priori not selected any nested `fails()` goal. The initial worst case `W` turns out to be the failure of each event that should install a wanted property: `W=`$\bot^*$. No matter how the loop situation looks like, it will now force the calculus to recognise a persistence inconsistency and therefore initiate a refinement on the current temporal order of events.

3.7 Remarks

Due to the different computational expense of the two versions of the Event Calculus, the EVE implementation includes both and allows the user to switch between them. Depending on the domain requirements, the user can freely choose the resources he is willing to spend. Together with the parametrisable proof procedure at the next layer, the EVE module becomes very flexible with respect to its resource demands. Further efficiency is gained by heuristics to prune the spawned search space and guide the 'execution' of the calculus. Since these are linked tightly to the theorem proving algorithm, they are described in more detail in the appropriate Sections 4 and 5.4.

The representation of time and action within EVE relies on the ideal assumption of events having no duration. As soon as duration must be modelled, a conflict with the notion of strong nonlinearity arises. This is caused by the non-existence of definite limits on the lifetime of pre- and postconditions. Furthermore, additional properties that arise as an event is expanded to sub events at a deeper abstraction level, forming the the so-called *during conditions*, have to be considered. Section 12.2 discusses some of the questions related to this issue.

Amongst the improvements that ease the customisation of the temporal reasoning process is the ability to model *context-dependent* action types with dependen-

[7] The execution layer checks the preconditions.

$\forall$P,Tp,L,W (holdstrue(P,Tp,L,W) $\equiv$ $\exists$EI,TyI (happens(EI) $\wedge$ act(EI,AI) $\wedge$ initiates(AI,P) $\wedge$ before(end(EI),Tp) $\wedge$ $\neg$fails(EI,L,$\neg^*$(W)) $\wedge$ $\neg$clippedtrue(P,end(EI),Tp,P,L,$\neg^*$(W)))
$\forall$P,Tp,L,W (holdsfalse(P,Tp,L,W) $\equiv$ $\exists$ET,AT (happens(ET) $\wedge$ act(ET,AT) $\wedge$ terminates(AT,P) $\wedge$ before(end(ET),Tp) $\wedge$ $\neg$fails(ET,L,$\neg^*$(W)) $\wedge$ $\neg$clippedfalse(P,end(ET),Tp,P,L,$\neg^*$(W)))
$\forall$E,L,W (fails(E,L,W) $\equiv$ (member(E,L) $\wedge$ W=$\perp^*$) $\vee$ $\exists$PD,A ($\neg$member(E,L) $\wedge$ act(E,A) $\wedge$ posprecondition(A,PD) $\wedge$ $\neg$holdstrue(PD,start(E),E|L,$\neg^*$(W))) $\vee$ $\exists$PD,A ($\neg$member(E,L) $\wedge$ act(E,A) $\wedge$ negprecondition(A,PD) $\wedge$ $\neg$holdsfalse(PD,start(E),E|L,$\neg^*$(W)))
$\forall$Tp1,Tp2,P,L,W (clippedtrue(P,Tp1,Tp2,L,W) $\equiv$ $\exists$EC,AC (happens(EC) $\wedge$ act(EC,AC) $\wedge$ terminates(AC,P) $\wedge$ $\neg$fails(EC,EC|L,$\neg^*$(W)) $\wedge$ $\neg$out(end(EC),Tp1,Tp2))
$\forall$Tp1,Tp2,P,L,W (clippedfalse(P,Tp1,Tp2,L,W) $\equiv$ $\exists$EC,AC (happens(EC) $\wedge$ act(EC,AC) $\wedge$ initiates(AC,P) $\wedge$ $\neg$fails(EC,EC|L,$\neg^*$(W)) $\wedge$ $\neg$out(end(EC),Tp1,Tp2))
$\forall$Tp1,Tp2,Tp3 (out(Tp1,Tp2,Tp3) $\equiv$ before(Tp1,Tp2) $\vee$ before(Tp3,Tp1) $\vee$ Tp3=Tp1)

Figure 18: The Extended EVENT CALCULUS of EVE

cies between conditions.[8] These can be transformed in a first approach into several non-context-dependent types. [9] Because additional actions of course increase the branching factor of the resulting planning search space, some dependencies would be better described by role constraints (see Sections 8.3 and A.2). Further research will try to incorporate some of the more sophisticated mechanisms to deal with context-dependencies in the EVENT CALCULUS (Section 12.2).

3.8 Summary

In this section, we have outlined the general techniques a theory of time and action has to provide to be suitable for nonlinear temporal reasoning. After introducing the well known SITUATION CALCULUS, its inability to cope with the frame problem in nonlinear settings was shown. The EVENT CALCULUS, however, solves this problem with the concept of persistence. Although designed for database updates, it is as well suited for plan representation purposes. The original formulation from [30] has revealed some disadvantages in the field of strong concurrency that have been improved on by [23]. The possibility of infinite reduction in this approach, the main focus of this report, is handled proposing two derivates of the EVENT CALCULUS. The first one, which is particularly simple to implement, is especially suited for cooperative domains and planning from scratch. It incorporates a worst case assumption about the action execution mechanism. The second alternative is not restricted to any special domain. It thus regards failed events as irrelevant and includes the critical failure checks for persistence destroyers. Termination, however, can still be guaranteed by mutual dependency detection and an appropriate worst

[8] If Pre_1 holds at the start of the action, then $Post_1$ will hold at the end ... If Pre_2 ... then $Post_2$ ...

[9] One action with $Pre_1 \rightarrow Post_1$, the other action with $Pre_2 \rightarrow Post_2$; ...

case assumption. In EVE, both methods can be switched which leaves the decision about the resource consumption trade-off to the user. In each case, completeness of the solution plans is slightly restricted but remains sufficient all practical planning applications.

4 Theorem Proving with Abduction: SLDNFA+

Given a calculus, e.g., the EVENT CALCULUS, in the clause syntax of Figure 1, its interpretation requires an underlying theorem proving mechanism. Whereas for clauses C (without negation), it suffices to use pure *resolution*, negation and the hypothetical reasoning behind *abduction* demand more sophisticated procedures. This section introduces the necessary conceptual background, presents several algorithms and discusses their soundness and completeness (with respect to solution substitutions/hypotheses) to finally enable the handling of clauses $\tilde{C}$ (with negation) with abductive inference.

4.1 Resolution: SLD

The concept of *resolution* goes back to [38]. It is a *refutation* procedure that derives a theorem T by showing that $\neg T$ is unsatisfiable. SLD [18] is a specialisation of this principle by dealing with first-order clauses instead of arbitrary formulae. Given a clause program $C_P = (C_1 \wedge \ldots \wedge C_n), \quad C_1, \ldots, C_n \in C$ and a goal $G = (\top \equiv D) \models\!\mid D$, it consists of the construction of a transformation $G = G_1 \vdash_{sld} G_2 \ldots \vdash_{sld} G_m$. A proof of G succeeds by reducing it to $G_m = \top$ and deriving an appropriate substitution μ for the existentially quantified variables of D such that $C_P \models \mu(G) \Rightarrow C_P \models G$.

The interesting part is the *resolution step* (see Figure 19) that is responsible for constructing each stage of the transformation ($G_x \vdash_{sld} G_y$). Based on the observation:

$$\textbf{(1)} \quad \neg\mu_y(G_x) \models_{C_P} \neg G_y$$
$$\Leftrightarrow \exists i \in 1, \ldots, n \quad \neg\mu_y(G_x) \models_{C_i} \neg G_y$$

the resolution step tries to *resolve* a selected subgoal $G_{x,j}$ of G_x with a clause *variant* $C_{i,v} = \forall \bar{W}_v (B(\bar{W}_v) \equiv D_v)$. In usual clause-based systems, this leads to unification (the **unify** function) of the clause head B and $G_{x,j}$ to produce the most general unifier μ_y and $G_y = \mu_y(\{G_{x,1}, \ldots, G_{x,j-1}, D_v, G_{x,j+1}, \ldots, G_{x,m}\})$. Since the normalised syntax that we chose realises unification with the equality predicate =, the unifier is in this case only a variable replacement in the clause variant. Variants (returned by the **variant** function) are unique variable renamings that introduce new names not clashing with already introduced symbols. They are therefore used to eliminate the quantifiers — existential in the goal and universal in clauses — in a sound manner.

If the notion of a *goal set* is used, it always means a conjunction of its members and builds the base structure for resolution. Disjunctions are handled by nondeterministic choice (the **choose** function) to obtain several, purely conjunctive proof stages that consist of a goal set and a substitution. Whenever a resolution step has been computed, the resolution procedure must decide which of all still competing stages (resulting from different disjunctive branches) to expand further with the next resolution step. Let us call this decision the *choice rule*.

Selection of the appropriate subgoal to resolve next can, however, be deterministic. The rule that decides about this selection is the *selection rule* (the **select** function). From its name follows the abbreviation SLD: *Linear resolution with a Selection rule for Definite Clauses*[10].

Note that the use of **choose** to find the right clause in the program with which to resolve next is just a simplification. Since there exists exactly one clause whose head is able to match the selected goal — remember the unique definition of normalised clauses — a deterministic comparison is also able to find it. In our completion syntax, it is disjunction that introduces several definitions for the same predicate.

[10] Better *Selection rule driven Linear resolution for Definite Clauses*

Equality and truth values have a special status. Equality could be axiomatised as a set of clauses, e.g., CET, Clark's equality theory, equivalent to the unification calculus. A better solution, however, is its immediate integration into the proof procedure that anyway has to provide some unification scheme. A selected $s{=}t$ subgoal therefore triggers the built-in unification procedure **unify** and is replaced by the implications of the returned by the unifier. Treatment of $\top$ and $\bot$ is straightforward due to the conjunctive semantics of the goal set — the **fail** function removes the current proof stage from the already collected stages and the choice rule has to find another candidate to continue the proof.

The soundness of the resolution principle with respect to tautologies and solution substitutions follows from the correctness of the resolution step **(1)** [11]:

$$\mu = \mu_m \circ \ldots \circ \mu_2$$

$$\neg\mu(G) = \neg\mu_m \circ \ldots \circ \mu_3(\mu_2(G_1)) \models_{C_P}$$

$$\neg\mu_m \circ \ldots \circ \mu_4(\mu_3(G_2)) \models_{C_P}$$

$$\ldots \models_{C_P} \neg\top \models \bot \quad [12]$$

$$\Leftrightarrow C_P \wedge \neg\mu(G) \models \bot \Leftrightarrow C_P \models \mu(G)$$

Completeness can only be guranteed if the choice rule is fair. This is also a requirement of all the following proof procedures. As already mentioned, the use of the most general unifier improves the proof procedure without affecting completeness.

4.2 Negation As Failure: SLDNF

Clauses C without negation form only a subset of first-order logic. To regain the full expressivity [22], it is necessary to introduce the negation operator: $\tilde{C}$. This operation turns out to be nonmonotonic — the denotational semantics is not as simple as in the SLD case. A way to incorporate it into clause resolution is through *negation as failure* (NF) [20] that relies on a closed world assumption and thus the completion of a program.

Whenever the selection rule chooses a negated subgoal, the truth of this subgoal can be demonstrated by the failure of a subordinate resolution proof on the positive version of the goal. When applied to SLD, the resulting algorithm is called SLDNF [18] (Figure 20).

However, goals with variables remaining in the scope of some quantifier give rise to several correctness and completeness questions. As they are not instantiated, they are treated as being universally quantified by the nested proof and can never obtain any value. Depending on the selection rule, one can expect incorrect solutions as shown in Figure 21. The double negation starts two nested proof attempts with variable `X`. The innermost succeeds because there are two possible substitutions `X=a` $\vee$ `X=b` for `q(X)` to derive the $\top$ goal. On the next higher-level, this result is inverted to a failure. Finally, the top level proof succeeds and is even able to unify `X=c` — this is of course an incorrect result. With another selection rule that selects `X=c` before $\neg$ `nq`, the same query leads correctly to a failure.

A common restriction to guarantee coincidence of the proof procedure with the semantics of negation is therefore to allow only expansion of *ground* (i.e. completely instantiated) negated goals. In our description, this test is done by $\mathcal{F}_{\bar{V}}(K) = \phi$. But now the completeness is restricted as seen in Figure 22. Selection of the negated

[11] $\circ$ is here the combination of substitutions.

[12] With our clause notation using $\equiv$, a constructive proof looks identical since the resolution step acts as a equivalence transformation. Usually, clauses are defined using back implication, $\subset$, and only admit the refutation approach in which the resolution step only computes an implication.

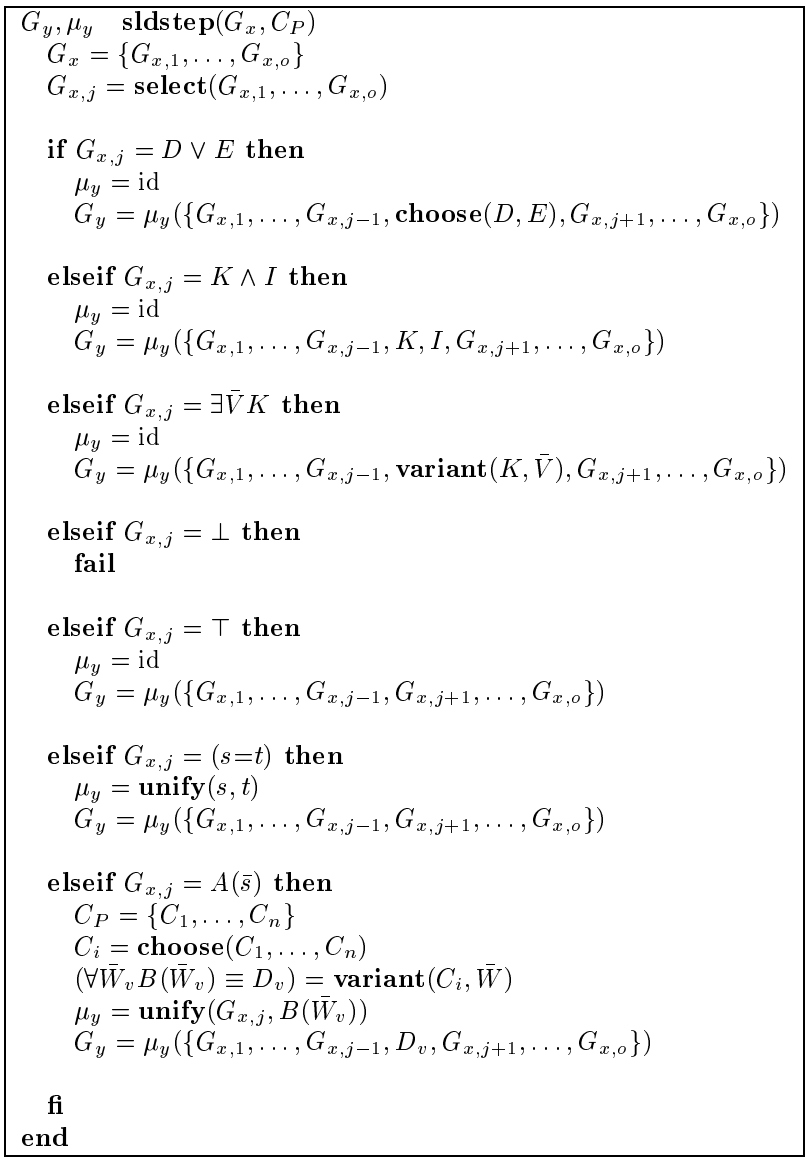


Figure 19: The SLD Step

G_y, μ_y **sldnfstep**(G_x, C_P)
  $G_x = \{G_{x,1}, \ldots, G_{x,o}\}$
  $G_{x,j} =$ **select**$(G_{x,1}, \ldots, G_{x,o})$

  if

  ... see the appropriate SLD *cases*

  elseif $G_{x,j} = \neg\exists\bar{V}K$ **then**
    if $\mathcal{F}_{\bar{V}}(K) = \phi$ **then** [a]
      if sldnf$(\exists\bar{V}K, C_P)$ **then**
        fail
      else
        $\mu_y = \mathrm{id}$
        $G_y = \mu_y(\{G_{x,1}, \ldots, G_{x,j-1}, G_{x,j+1}, \ldots, G_{x,o}\})$
      fi
    else
      fail
    fi

  fi
end

[a] $\mathcal{F}(K)$ are the free variables of K, $\mathcal{F}_{\bar{V}}(K) = \mathcal{F}(K) \setminus \bar{V}$

Figure 20: The SLDNF Step

$\forall$X q(X) $\equiv$ X=a $\vee$ X=b
$\forall$X nq(X) $\equiv \neg$q(X)
$\forall$X r(X) $\equiv \neg$nq(X)
$\top \equiv \exists$X(r(X) $\wedge$ X=c)

Figure 21: In-soundness of Non-ground NF

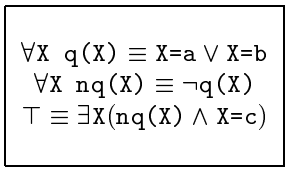


Figure 22: Incompleteness of Ground NF

goal could happen before the unification `X=c` and therefore fail the of groundness requirement. A further obstacle with respect to completeness is the inability of providing a fair choice rule, since the top-level proof has to wait until the subordinate negation as failure attempt delivers a result.

4.3 Constructive Negation

Although SLDNF is sufficient to interpret the EVENT CALCULUS as a clause program, its completeness has to be improved in order to build the base for the hypothetical reasoning that is introduced in Section 4.5. This can be obtained by the technique of constructive negation [7].

The basic idea is the extension of the *equality maintenance* system represented by substitution and unification to cope even with *inequalities* ($\mathtt{X} \neq s$) and disjunctive choice due to *disunification* (symmetrical procedure **disunify** to **unify** to derive $s \neq t$). Substitution application to a formula is no longer straightforward. Whereas pure equality is obtained by the appropriate variable replacements, inequality has to somehow annotate the formula with its implications ($\{F \mid\mid \mathtt{X}_j \neq s_j\}$). Finally, unification must be able to retain this information for an extended consistency check: as soon as a substitution renders the annotation inconsistent, i.e., the appropriate equations are satisfiable for all substitutions, the procedure fails.

4.3.1 SLDCNF

SLDCNF (Figure 23) uses these extended mechanisms to resolve even non-ground negations. Correctness is established by the collection of all relevant information from the nested proof (the conjunctive inversion of all solution substitutions $\mu_y = \mu_y \bigcup \neg\mu^-$) and by further maintenance in the equality system. The overall soundness with a consistent solution substitution is obtained by the following lemma:

Lemma 4.1 (Malc'ev's Lemma) *Given an infinite number of atom and function symbols, each formula* $\tilde{\exists}(\bigwedge \neg(\mathtt{X}_i{=}t))$ *is satisfiable,*

Proof. by induction over the sub formulae and by the infinite number of closed terms that can be assigned to each variable. □

Special cases such as the one in Figure 22 no longer depend on any selection rule, because the groundness restriction is removed. Construction of the complete solution substitution space is not possible, because $s \neq t$ has in general infinitely many solutions whose enumeration is not practical. Thus even a fair choice rule can no longer guarantee completeness because of the dependency of the top-level proof on every solution of the subordinate negation proof.

4.3.2 SLDNF$^+$

The lack fairness introduced by the nested proof schemes and the rather complex handling of the substitution formulae are reason to present another procedure:

G_y, μ_y **sldcnfstep**(G_x, C_P)
  $G_x = \{G_{x,1}, \ldots, G_{x,o}\}$
  $G_{x,j} =$ **select**$(G_{x,1}, \ldots, G_{x,o})$

  if

  ... see the appropriate SLD *cases*

  elseif $G_{x,j} = \neg\exists\bar{V}K$ **then**
    $\mu_y = \mathrm{id}$
    for $(\mu^- =$ **sldcnf**$(\exists\bar{V}K, C_P), \mu_y = \mu_y \bigcup \neg\mu^-)$
    $G_y = \mu_y(\{G_{x,1}, \ldots, G_{x,j-1}, G_{x,j+1}, \ldots, G_{x,o}\})$

  fi
end

Figure 23: The SLDCNF Step

$\forall V\, A(V) \equiv D$	$\models\!\!\!\models$	$\forall V\, (\neg A\ \ (V) \equiv \neg D\ \ )$
$\neg(D \vee E)$	$\models\!\!\!\models$	$\neg D \wedge \neg E$
$\neg(\exists V K)$	$\models\!\!\!\models$	$\forall V \neg K$
$\neg(K \wedge I)$	$\models\!\!\!\models$	$\neg K \vee \neg I$
$\neg\top$	$\models\!\!\!\models$	$\bot$
$\neg\bot$	$\models\!\!\!\models$	$\top$

Figure 24: The De Morgan's Laws

SLDNF$^+$ (Figure 25). This algorithm provides an interleaved proof of negated and positive goals. The basic idea for handling negated goals is via equivalence transformations based on De Morgan's laws (Figure 24). If the procedure is finally able to construct the *negation normal form* that lifts the negation signs to the deepest level of primitive predicates (truth values and equality), the result will be sound as well as complete. The application of the De Morgan's laws is trivial except for two cases.

First, there is the possibility that a substitution μ renders both sub formulae $\mu(K)$ and $\mu(I)$ of a negated conjunction $\neg(K \wedge I)$ equivalent to $\bot$ — the simple translation into $\neg K \vee \neg I$ has several equivalent solutions, which needlessly increases the possibilities for *backtracking* (choosing another branch at a choice point). The better approach is to choose $\neg K \vee (K \wedge \neg I)$ which is still equivalent to $\neg(K \wedge I)$ but produces real disjoint sub formulae.

Second, the result from De Morgan's laws for the negated, existentially quantified conjunctions, $\neg\exists\bar{V}K$ with $K = K(\bar{V})$, produces universal quantification . The procedure is not able to cope with such formulae in infinite domains. A *finite domain* assumption F about the local variables of the negated goal delivers the necessary background theory:

$$F = \bigwedge_{\mathtt{X} \in \bar{V}} \mathtt{fido(X)} \models\!\!\!\models (\bar{V} = \bar{s}_1 \vee \ldots \vee \bar{V} = \bar{s}_n)$$

$$\neg\exists\bar{V}K(\bar{V}) \models\!\!\!\models \forall\bar{V}(\neg K(\bar{V})) \models\!\!\!\models_F \forall\bar{V}(\bigwedge_{\mathtt{X} \in \bar{V}} \mathtt{fido(X)} \wedge \neg K(\bar{V}))$$
$$\models\!\!\!\models_F \neg K(\bar{s}_1) \wedge \ldots \wedge \neg K(\bar{s}_n)$$

The example in Figure 26 shows the functionality of SLDNF$^+$, but, for simplicity, the derivation always takes the right choice and we present the results of several steps in one line. Summarising the foregoing explanations, it can be stated that a fair SLDNF$^+$ is sound and complete for programs that provide a finite domain assumption for the variables involved in negated goals. A generalisation of the finite domain assumption to all variables, however, is not sound, because Malc'ev's lemma does no carry over to this case.

4.3.3 Efficient Negation in SLDNF$^+$

In general, SLDNF$^+(\exists\bar{V}K)$ branches exponentially with the size of the domain and number of variables. But much of the search space is immediately pruned in most calculi by the expression $K = K(\bar{V})$[13] itself. A better starting point for the translation is therefore to distinguish parts $K_g(\bar{V})$ in which global variables are involved and $K_l(\bar{V})$ in which only elements of $\bar{V}$ appear. Restricting the instantiations $\bar{s}_1, \ldots, \bar{s}_n$ for $\bar{V}$ to the solutions of $\exists\bar{V}(\bigwedge_{\mathtt{X}\in\bar{V}} \mathtt{fido(X)} \wedge K_l(\bar{V}))$ delivers $\neg K_g(\bar{s}_1) \wedge \ldots \neg \wedge K_g(\bar{s}_n)$ as the result of the transformation. $K_l(\bar{V})$ could even describe some finite domain and therefore suffice to ground $\bar{V}$, i.e., the finite domain assumption can be totally removed.

It is not necessary to restrict this improvement to the conjunctive level of K alone. Nested, limited expansion of K can further enlarge K_l and thus the solution space of the local variables. But there has to be a sound manner to deal with disjunctions and negations to obtain local and global sub formulae K_l and K_g. [33] develops this technique to its limit by describing a general framework for sound rewriting rules on arbitrary expansion depths (delivering a *contour* of K).

4.4 Abduction: SLDA

Abduction has been proposed as being a quite different inference principle to *deduction*. It extends the deductive facilities by providing hypothetical reasoning that is especially used in diagnostic systems (medical diagnosis, fault diagnosis, etc.). Compared to resolution, it aims to find a set of axioms $Ab = \{\forall\bar{V}(\ A(\bar{V}) \equiv \bigwedge_{\mathtt{X}\in\bar{V}} \mathtt{X{=}}s\ )\}$, also called *facts*, that enable the program C_P to imply the goal: $(C_P \wedge Ab) \models G$. The easiest way to extend the resolution step with clauses C to generate this set is to allow the selected goal to serve as an *abducible*(*assumption*, *hypothesis*). This is of course not desirable for any literal. With no additional structural information, the space of possible hypotheses for a certain observation is too large to be practically enumerable. One therefore takes assumptions from a predefined set, the *abductive* predicates (selected by the **abductive** function in our algorithms). Once assumptions are collected, further goals are also provable by unification with those already abduced facts. The additional mechanism to the resolution step is called the *abduction step* and the actual set of collected abducibles is called *abduction set*, *abduction state* or *residue*.

An important question is the treatment of uninstantiated variables in the goal to abduce. If one treats the result of such an abduction step as a clause by adding it to the program C_P, the variables would be universally quantified, which is too strong. In order to preserve correctness, either *skolem terms* have to be inserted which requires an extended unification algorithm or the assumptions have to be separated from C_P and variables are only allowed to appear quantified as in the procedure presented in Figure 27.

What about the semantics of the abductive predicates? Each domain tends to put certain restrictions on the denoted relations, e.g., *reflexivity*, *symmetry*, *transitivity*, or special structured arguments. This is ensured in the proof procedure

[13] K() denotes a function that produces an expression from its arguments.

G_y, μ_y **sldnf**$^+$**step**(G_x, C_P)
  $G_x = \{G_{x,1}, \ldots, G_{x,o}\}$
  $G_{x,j} =$ **select**$(G_{x,1}, \ldots, G_{x,o})$

  if

  ... see the appropriate SLD *cases*

  elseif $G_{x,j} = \neg(D \vee E)$ **then**
    $\mu_y = \mathrm{id}$
    $G_y = \mu_y(\{G_{x,1}, \ldots, G_{x,j-1}, \neg D, \neg E, G_{x,j+1}, \ldots, G_{x,o}\})$

  elseif $G_{x,j} = \neg(K \wedge I)$ **then**
    $\mu_y = \mathrm{id}$
    $G_y = \mu_y(\{G_{x,1}, \ldots, G_{x,j-1}, \mathbf{choose}(\neg K, K \wedge \neg I), G_{x,j+1}, \ldots, G_{x,o}\})$

  elseif $G_{x,j} = \neg\exists\bar{V}K$ **then**
    $\mu_y = \mathrm{id}$
    $G_y = \mu_y(\{G_{x,1}, \ldots, G_{x,j-1}, G_{x,j+1}, \ldots, G_{x,o}\})$
    for$((\exists\bar{W}_v K_v) =$ **variant**$(\exists\bar{V}K)$,
      $\mu = \mathbf{sldnf}^+(\{\bigwedge_{\mathtt{X}\in\bar{W}_v} \mathtt{fido(X)}\}, C_P)$,
      $\mu_y = \mu_y \bigcup \mu$,
      $G_y = \mu_y(G_y \bigcup \neg K_v))$

  elseif $G_{x,j} = \neg\bot$ **then**
    $\mu_y = \mathrm{id}$
    $G_y = \mu_y(\{G_{x,1}, \ldots, G_{x,j-1}, G_{x,j+1}, \ldots, G_{x,o}\})$

  elseif $G_{x,j} = \neg\top$ **then**
    fail

  elseif $G_{x,j} = (s{=}t)$ **then**
    $\mu_y =$ **disunify**(s, t)
    $G_y = \mu_y(\{G_{x,1}, \ldots, G_{x,j-1}, G_{x,j+1}, \ldots, G_{x,o}\})$

  elseif $G_{x,j} = \neg A(\bar{s})$ **then**
    $C_P = \{C_1, \ldots, C_n\}$
    $C_i =$ **choose**$(C_1, \ldots, C_n)$
    $(\forall\bar{W}_v B(\bar{W}_v) \equiv D_v) =$ **variant**$(C_i, \bar{W})$
    $\mu_y =$ **unify**$(G_{x,j}, B(\bar{W}_v))$
    $G_y = \mu_y(\{G_{x,1}, \ldots, G_{x,j-1}, \neg D_v, G_{x,j+1}, \ldots, G_{x,o}\})$

  fi
end

Figure 25: The SLDNF$^+$ Step

$G_1 = \{\underline{\neg \texttt{q(X)}}, \texttt{X=c}\}$ $C_P = \{\forall \texttt{X (q(X)} \equiv \exists \texttt{Y r(X,Y))},$ $\quad \forall \texttt{X,Y (r(X,Y)} \equiv \texttt{X=a} \wedge \texttt{Y=a)}\}$ $FiDo = \forall \texttt{X(} \equiv \texttt{X=a} \vee \texttt{X=c)}$	query[a] program finite domain for negations
$G_2 = \{\underline{\neg \exists \texttt{Y r(X,Y)}}, \texttt{X=c}\}$	negative resolution
$G_3 = \{\underline{\neg \texttt{r(X,a)}, \neg \texttt{r(X,c)}}, \texttt{X=c}\}$	$FiDo$ negation
$G_4 = \{\underline{(\neg \texttt{X=a} \vee \neg \texttt{a=a)}}, (\neg \texttt{X=c} \vee \neg \texttt{c=a}), \texttt{X=c}\}$	negative resolution & negative conjunction
$G_5 = \{\underline{(\neg \texttt{X=c} \vee \neg \texttt{c=a)}}, \texttt{X=c} \parallel \texttt{X} \neq \texttt{a}\}$	choice & disunification
$G_6 = \{\underline{\texttt{X=c}} \parallel \texttt{X} \neq \texttt{a}\}$	choice & disunification
$G_7 = \{\top \parallel \texttt{c} \neq \texttt{a}\}$	unification & success

[a] The underlined goals are chosen by the selection rule for the next expansion step.

Figure 26: Example Proof with SLDNF$^+$

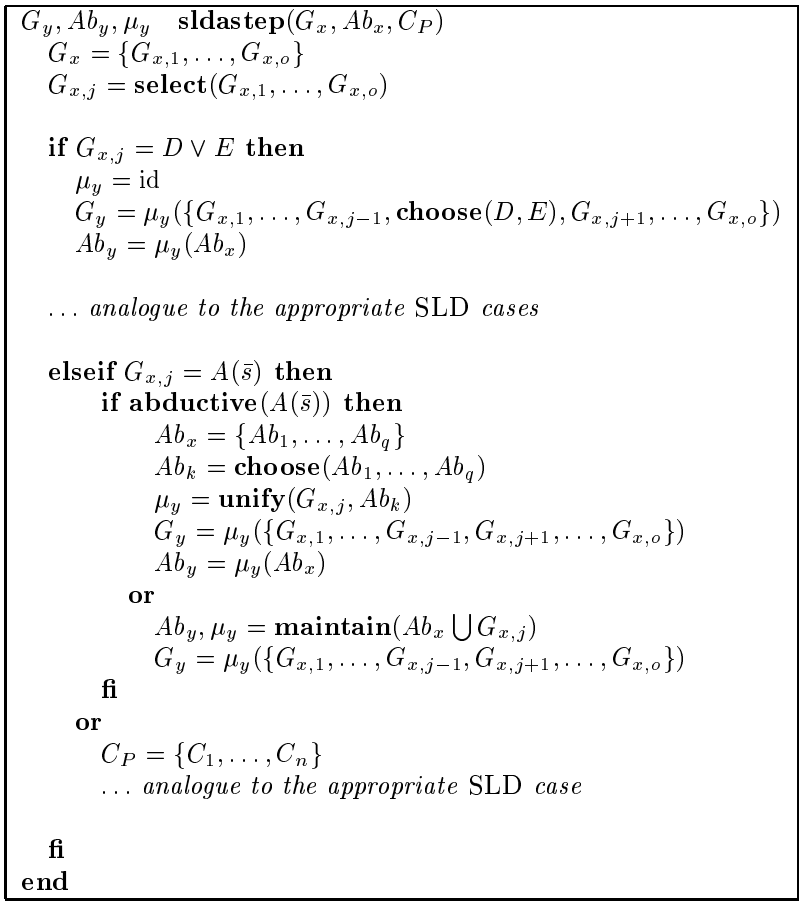


Figure 27: The SLDA Step

through *maintenance* (**maintain**) of the abducibles. It is responsible for unification, consistency checking and the computation of an appropriate completion of the abduction set. Maintenance can also be a further source of choice points.

In a similar way to resolution, the soundness of the solution assumptions is proved by the following invariant:

$$\neg\mu_y(G_x) \models_{C_P \wedge Ab_y} \neg G_y$$

For the whole abduction process can the be then stated:

$$\mu = \mu_m \circ \ldots \circ \mu_1$$

$$(C_P \wedge Ab_m) \models \mu(G)$$

Completeness, however, is not as easy. As we have already mentioned, it is inadvisable to take the whole hypotheses spaces as the base for completeness. Thus we state a *least commitment* completeness (given a fair choice rule) that eliminates the infinitely many possibilities that do not contribute to the goal in any way.

$$\forall Ab \quad (C_P \wedge Ab) \models G \Leftrightarrow \exists Ab' \quad Ab \models Ab', C_P \vdash_{slda,Ab'} G$$

Looking at the operational semantics of abduction, a closed world assumption is no longer possible. Incorporating both abduction and negation is therefore not trivial as the two interfere highly[14].

4.5 Negation and Abduction

4.5.1 SLDNFA

At first sight, it may not seem too difficult to bring the results of the former proof principles together. One has just to add an abduction step to SLDNF to obtain SLDNFA (Figure 28). However the introduction of new assumptions has to be disabled when the proof is in the negative mode ($M = -$) as we are only collecting positive facts.

An important observation follows from investigating the search spaces in the negative mode. Additional abducibles could increase these proof trees and render previous NF proofs incorrect ($\underline{\neg \mathtt{q(X)}} \wedge \mathtt{q(X)}$). To circumvent this and to obtain a sound procedure, it is necessary to collect the already proved negations (N_x). As soon as new assumptions are added in positive mode, the members of N_x have to be proved again (**reprove**) under the current abduction state. More reasonable behaviour would be obtained with a selection rule, that delays negations until the abduction steps are done. Again, as with SLDNF, SLDNFA has limited completeness because of the groundness restriction and problems with fairness. Furthermore, nested negations do not re-enable the collection of new hypotheses, thus not all relevant assumptions are generated.

4.5.2 SLDCNFA

The same approach works for extending SLDCNF with abductive features to SLDCNFA (Figure 29). Nested negative proofs happen on a static abduction state and a consistency check after additional assumptions has to be carried out to derive further equality/inequality information. Completeness is restricted, because the nested proof scheme cannot guarantee fairness and the abduction state is frozen within subordinate proofs.

[14] Unification, indeed, is the **base** of the abduction step. The problems of non-ground negation thus are a special case of the general interference of negation and abduction.

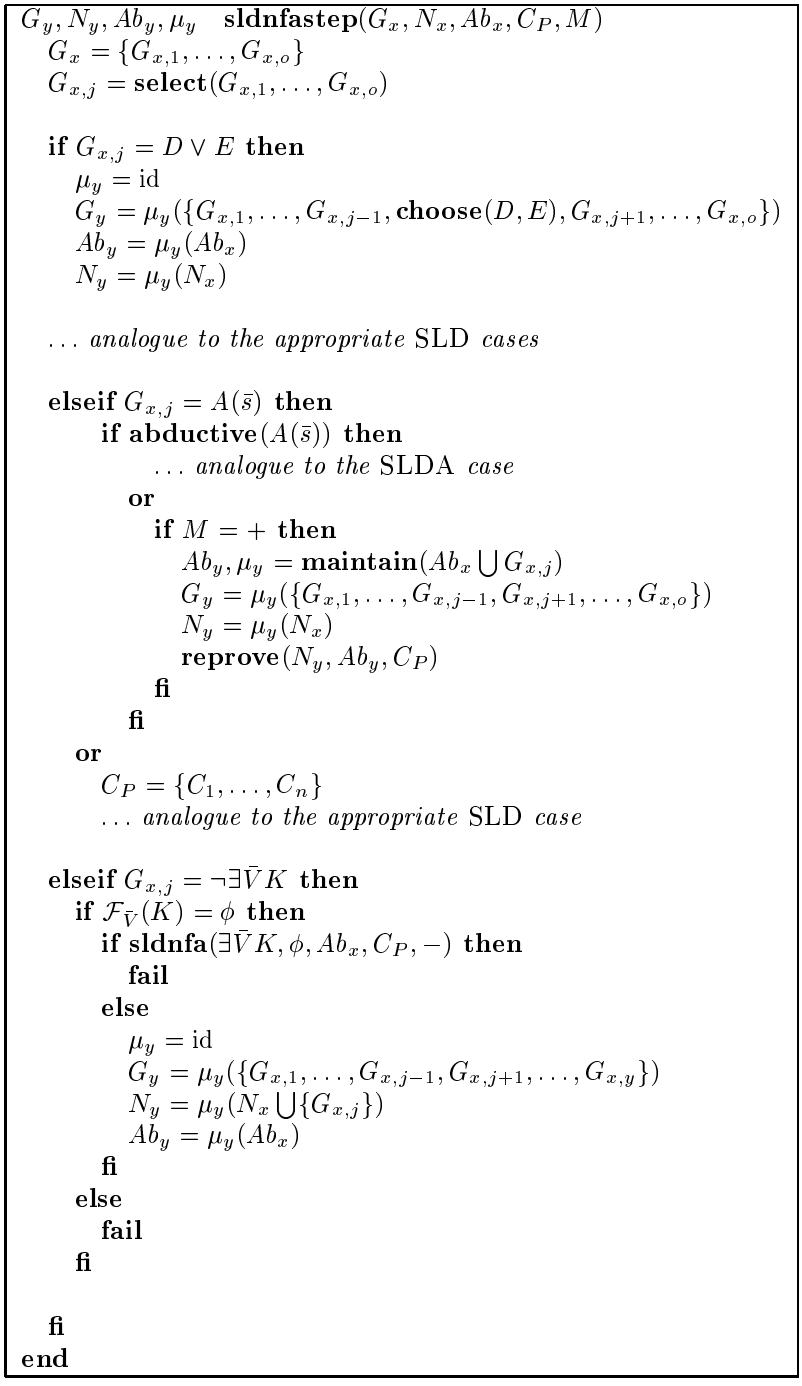


Figure 28: The SLDNFA Step

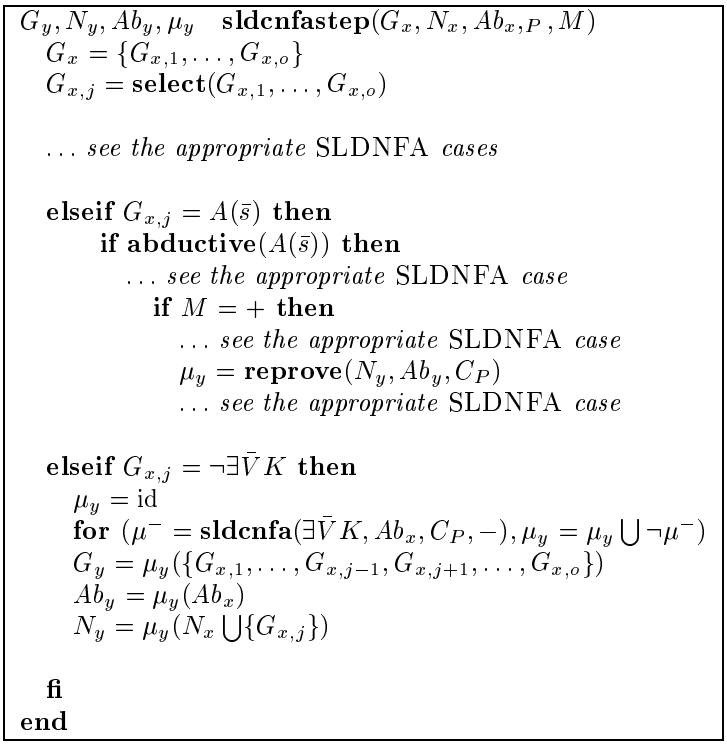


Figure 29: The SLDCNFA Step

4.5.3 SLDNFA$^+$

Neither SLDNFA nor SLDCNFA can guarantee completeness of the abducibles due to the frozen abduction state in nested proofs and the lack of fairness. To improve this situation we need to introduce further assumptions in nested negation proofs. The interleaved proof method of SLDNF$^+$ provides a better base to include this into SLDNFA$^+$ (Figure 30). The treatment of positive goals handles the addition of new assumptions. On the negative side, requests about the abduction state, that is about *negative abductive* goals, require constructive negation. Since the procedure is even able to memorise the structure of the proof trees, a correctness check following several abduction steps can be carried out more easily. Only branches that involve accessing the abducibles need to be revisited, instead of reconstructing the whole tree from scratch. Selection rules that prefer positive goals to negated ones further reduce the number of restructuring tasks.

SLDNFA$^+$ is derived from a description [25] that is proved to be sound and complete with respect to the least commitment solutions typical in the abduction literature. The soundness result also holds for SLDNFA$^+$, whereas overall completeness requires the finite domain assumption from Section 4.3 and fairness.

A more efficient treatment of negation as in SLDNF$^+$ is also possible in the abductive case, but with some special considerations. Those parts of the goal that migrate into the finite domain enumeration are restricted not to change the abduction state. Abductive parts should reside outside and thus contribute to the constructive result of the negated goal proof.

G_y, N_y, Ab_y, μ_y **sldnfa$^+$step**(G_x, N_x, Ab_x, C_P)

... analogue to the appropriate SLDA *cases*

elseif $G_{x,j} = A(\bar{s})$ **then**
  if abductive$(A(\bar{s}))$ **then**
    ... analogue to the appropriate SLDA *case*
      $Ab_y = \mu_y(Ab_x)$
      $N_y = \mu_y(N_x)$
    or
      $Ab_y, \mu_y =$ **maintain**$(Ab_x \bigcup G_{x,j})$
      $G_y = \mu_y(\{G_{x,1}, \ldots, G_{x,j-1}, G_{x,j+1}, \ldots, G_{x,o}\})$
      $N_y = \mu_y(N_x)$
      $Ab_y, \mu_y =$ **reprove**(N_y, Ab_y, C_P)
  fi
or
  $C_P = \{C_1, \ldots, C_n\}$
... analogue to the appropriate SLDNF$^+$ *cases*

elseif $G_{x,j} = \neg\exists\bar{V}K$ **then**
  $\mu_y = \mathrm{id}$
  $G_y = \mu_y(\{G_{x,1}, \ldots, G_{x,j-1}, G_{x,j+1}, \ldots, G_{x,o}\})$
  for$((\exists W_v K_v) =$ **variant**$(\exists\bar{V}K)$,
    $\{\phi, \mu\} =$ **sldnfa$^+$**$(\bigwedge_{\mathtt{X}\in\bar{V}} \mathtt{fido(X)}, \phi, Ab_x, C_P)$,
    $\mu_y = \mu_y \bigcup \mu$,
    $G_y = \mu_y(G_y \bigcup \neg K_v))$
  $N_y = \mu_y(N_x \bigcup G_{x,j})$

... analogue to the appropriate SLDNF$^+$ *cases*

elseif $G_{x,j} = \neg A(\bar{s})$ **then**
  if abductive$(A(\bar{s}))$ **then**
    $\mu_y = \mathrm{id}$
    for$(Ab_k \in Ab_x; \mu =$ **disunify**$(G_{x,j}, Ab_k); \mu_y = \mu_y \bigcup \mu)$
      $G_y = \mu_y(\{G_{x,1}, \ldots, G_{x,j-1}, G_{x,j+1}, \ldots, G_{x,o}\})$
      $Ab_y = \mu_y(Ab_x)$
      $N_y = \mu_y(N_x)$
  fi
or
  $C_P = \{C_1, \ldots, C_n\}$
... analogue to the appropriate SLDNF$^+$ *cases*

fi
end

Figure 30: The SLDNFA$^+$ Step

4.6 Remarks

That the complexity of theorem proving problem is NP-hard tells us that there will be no general, non-scalar improvement in the proof procedures applicable to all queries. A major reduction of the amount of search can therefore only be obtained with input-dependent guide rules: *heuristics*.

Whereas in the resolution case, heuristics are equivalent to dynamic control of choice points, abduction needs a similar, but extended notion. A general strategy is to first check the goal for unification with already assumed hypotheses and only later consider adding to the abduction set. Obeying this principle, a proof will derive minimal abduction sets, which in turn usually reduces the proof complexity. Furthermore the order of goal comparison within the current abduction state should be dependent on the program and is often the major key to efficiency, as in the case of the EVENT CALCULUS (see section 5.4). There are also environments that require further choice in the maintenance of the abduction set, which should be also be amenable to heuristic guidance.

A close inspection of the control flow through the abduction steps can also be a rewarding activity. Depending on the kind of calculus and the relations denoted by the abductive predicates, the resource hungry consistency check of negated goals is not always necessary. EVE's EVENT CALCULI are examples of this and some of those checks may be left out (see Section 5).

4.7 Summary

From resolution, negation as failure, constructive negation and abduction, we have derived in this section a sound proof procedure SLDNFA$^+$, which is complete at least for the case of finite universe negation, fair choice rules and least commitment of hypotheses. After the presenting of the expansion steps of all the algorithms, we discussed in detail the important practical problems that all these procedures are faced with besides the theoretical questions of soundness and completeness. Although SLDNFA$^+$ is generally restricted to finite universe negation, we have demonstrated a more efficient handling of negation for most calculi.

5 The Event Calculus under SLDNFA$^+$

The SLDNFA$^+$ procedure presented above that supports resolution, negation, and abduction can be used for the interpretation (proof) with the Event Calculus in order to solve the planning problems described in Section 2.2. The following discussion focusses on the basic questions arise bringing the clause program and the theorem proof procedure together.

5.1 Planning by Theorem Proving

5.1.1 Plan Analysis and Evaluation

From the above results regarding theorem proving together with our formulations of the Event Calculus *EC*, the realisation of plan analysis and evaluation using resolution and negation in SLDNF$^+$ would seem to be easy. Given an axiomatisation of the planning domain *DOM* describing preconditions and effects of the existing action types (via `posprecondition()`, `negprecondition()`, `initiates()`, and `terminates()`), one is now able to compute the overall properties that plan *Plan*, consisting of `happens()`, `act()`, and `before()`, establishes at a certain point in time:

$$(Plan \wedge Dom \wedge EC) \vdash_{\text{SLDNF}^+} Goal$$
$$Goal = \texttt{holds(P,T)}$$

A query with an uninstantiated property `P` requests the answer substitutions that give a complete overview of the implicit situation at time point `T`. Partially instantiated properties `P` are evaluation requests, i.e. one would like to know whether there is an instantiation of property term that holds at time point T. Section 5.2 describes how to introduce a so-called 'dummy' event that represents the overall end of each plan and usually is used to bind `T` to get the overall effects.

Due to the three-level architecture using a theory of time and action and a theorem proof procedure, the soundness and completeness results depend on a combination of the individual requirements of each of those layers. Whereas the concept of a solution plan is determined by the plan execution model on which the appropriate version of the Event Calculus relies, the finite domain assumption for negations on the SLDNF$^+$ level imposes limits at a more theoretical level (see Section 5.3). Regarding the axiomatisation of the calculus, finite domain negation requires a finite number of action types, their conditions, and the plan steps, and thus finite plans. Finally, fair handling of choice points (Section 5.4) is also required to establish plan (and property) soundness/completeness at the top layer.

5.1.2 Plan Synthesis and Modification

The harder problem of plan synthesis and modification can be achieved with an abductive proof procedure:

$$Dom \wedge EC \wedge Plan_i \vdash_{\text{SLDNFA}^+, Plan_a} Goal$$
$$Goal = \texttt{holds(P}_1\texttt{,T}_1\texttt{)} \wedge \texttt{holds(P}_2\texttt{,T}_2\texttt{)} \wedge \ldots \wedge \texttt{holds(P}_n\texttt{,T}_n\texttt{)}$$

To construct a complete plan $Plan = Plan_i \bigcup Plan_a$ from an initial abduction set $Plan_i$, one has to introduce abductive predicates that are able to extend the plan $Plan_a$ (Section 5.2). The choice rule has to be extended supporting efficient abduction and maintenance of hypotheses (Section 5.4). Finally, a closer inspection

of the nonmonotonic interference within the proof reveals several opportunities for customising SLDNFA$^+$ to save unnecessary computation (Sections 5.5 and 5.6).

Depending on the number of predefined assumptions, the planning task resides somewhere between pure plan synthesis ($Plan_i = \phi$) and plan modification ($Plan_i \neq \phi$). This is only approximately exact, as the start event that describes the initial situation should be introduced in either case by the planning domain definition and the initial abduction set (Section 5.2).

Bearing in mind the statements about SLDNFA$^+$ from Section 4, the requirements of Section 5.1.1 for fairness, finite plans, finitely many action types, and finitely many conditions also hold in the case of plan synthesis. The following equation can be established for the overall planning procedure:

$$(Plan \wedge Dom \wedge EC) \vdash_{\text{SLDNF}^+} Goal$$
$$\Leftrightarrow \exists Plan' \quad Plan \models Plan', Dom \wedge EC \vdash_{\text{SLDNFA}^+, Plan'} Goal$$

On the one hand, this guarantees soundness with respect to the solutions of the appropriate EVENT CALCULUS version ($\Leftarrow$, $Plan' = Plan$). On the other hand, the least commitment of the abduction principle leads to a special form of plan completeness. Each reasonable extension to an abduced plan that does not destroy its special structure with respect to the goal can also form a solution. This is no restriction, since in general these extensions that are not relevant for obtaining the goal anyway. As in the general setting, dropping least commitment requirement only leads to unwanted solutions.

5.2 Abductive Predicates and their Maintenance

Enabling abductive facilities within the logical 'execution' of the EVENT CALCULUS requires some considerations with regard to abductive predicates and their semantics. As we mentioned above, a plan in the EVENT CALCULUS consists of the predicates `happens()`, `act()`, and `before()`. Thus, they also serve as abducibles for plan synthesis purposes:

- `happens()`, a unary predicate, requires no special maintenance. It does not need to be restricted in any way. Due to the definition of the calculus, it contains either unique event constants or terms involving only existentially quantified variables.

- `act()` is restricted to have a event as its primary argument (introduced with `happens()`) and a term that describes an action type with its roles as its second argument. The denoted relation should is functional, i.e., each event is only allowed to have one action type associated with it. Note that `act()` should not be responsible for introducing a concrete type for an event. This is achieved by later unifications (e.g., in a `postcondition()` axiom) with the existentially quantified type as an argument (see Section 5.4).

- `before()` has time points related to existing events as both of its arguments. The denoted relation is *anti*-symmetric and transitive. The best way to maintain this predicate is by the calculation of the transitive closure and failure on the detection of inconsistent assumptions. A default fact that has to be included for each event `E` introduced by `happens(E)` is `before(start(E),end(E))`.

Furthermore, the initial situation is incorporated into the planning problem by representing it as an event. The domain *Dom* is responsible for defining a special action type `initial`* that always succeeds (because of

empty preconditions) and initiates the appropriate facts. The abduction set is initialised with some default event and the facts `happens(initial)` ∧ `act(initial,initial*)` ∧ `before(start(initial),end(initial))`. As soon as `happens(E)` is abduced abduction maintenance is extended by a further default fact: `before(end(initial),start(E))`.

A similar method can be used to introduce an explicit time point for the implicit end of the plan that is commonly needed to request information about the overall effects. The appropriate dummy event and its action type are called `end`,`end*`. The type needs neither preconditions nor effects as it only serves as an argument for `start()` in `holds()`. Again, the maintenance of `before()` requires a default assumption for each introduced event `E`: `before(end(E),start(end))`.

5.3 Treatment of Negations

Sections 4.3.3 and 6.5 demonstrated how to improve the treatment of negation in order to save search space while enumerating a finite domain over local variables. The Event Calculus is a perfect example which provides the ability to put some subgoals into the enumeration part of SLDNF$^+$ and SLDNFA$^+$. This even works for all possibly negated goals with local variables (for simplicity, we have chosen to present them before the next resolution step occurs and exposes the quantified expression):

- ¬`fails(E)`: The local variable `Ty` is uniquely determined by `act(E,Ty)`. `P` is member of the preconditions of `Ty`. `act()` as well as `precondition()` can be moved into the enumeration part, if finitely many action types and finitely many conditions are preserved.
- ¬`clipped(P,Tp1,Tp2)`: `EC` as the destroyer of persistence can only be an already abduced event. `happens(E)` spawns the appropriate search space and migrates into the finite domain enumeration, assuming that there are finitely many plan steps. The other goals, `terminates()`, ¬`fails()`, and `out()`, should remain in the global part of the goal, because they either involve global variables or are able to introduce new hypotheses.
- ¬`holds(P,Tp)`: `E` as the destroyer of `P` is member of the abduced events. The transformation and the corresponding requirements are similar to the preceding case.

5.4 SLD-Rule, Choice Points and Heuristics

Though fairness of the choice rule is sufficient to obtain sound and relatively, complete planning procedures from the combination of the Event Calculus and SLDNFA$^+$, efficiency with respect to performance and optimal solutions is only established if the choice and selection rules are further restricted.

As it can be seen from the formulation of the Event Calculus and the definition of its abductive predicates, expanding the calculus in a depth-first manner (in both the selection and choice rules) is easy to implement and well suited to the problem. Instantiations occur in a reasonable fashion according to the selection rule, especially for negated subgoals. The complex treatment of negation and the maintenance of the abduction set are simple. There are however some opportunities to vary the pure depth-first approach to guarantee fairness and incorporate heuristics.

One important place to explore fair behaviour is the treatment of recursion due to `fails()`. Both the simple as well as the extended Event Calculus of EVE are designed to ensure termination on a static abduction set. But if abduction is

switched on, a depth-first method for selection as well choice yields infinite computations, depending on the particular planning domain axiomatisation and goal, e.g., in the case of several actions that serve as mutual precondition initiators. This is of course not fair and certainly does not produce a complete solution plan set.

The *iterative deepening* approach which postulates a proof *depth* limit that is increased after covering the complete search space at a certain depth is most appropriate here. Compared to the usual bounding measure of absolute proof depth, abduction in the case of the EVENT CALCULUS allows an even smarter bound: the number of yet abduced events by `happens()`. Besides fairness and thus completeness relative to the definition of EVE's calculi, this measure is also responsible for yielding optimal plans with the fewest steps [15]. Even with minimal and maximal limits, EVE is able to implement correct solution management.

Heuristic knowledge, however, must also be applied at important choice points and the abductive steps. Whereas plan analysis always tries to explore all the existing proof paths to get an overview about an implicit situation and thus demands no extensive use of heuristics, synthesis is faced with the far more complex task of generating the right action at the right place and requires the best possible guidance in order to reach its solution fast and efficiently.

The EVENT CALCULUS incorporates three different places to influence depth-first choice towards a more efficient behaviour:

1. **Choice between a new abducible and a match with the already collected assumptions:** a general rule is to first check existing hypotheses. Since complexity increases with additional hypotheses, the abduction step has to respect this order (*extended* least commitment).

2. **Choice of abducibles to match the goal with:** Finding a goal-initiating event is best done by accessing the start situation and the already present events first. The solution plans stay minimal, because it will be not necessary to have a large number of events beside the knowledge base, but only if the environment is suitable.

3. **Choice of action type associations:** Choosing a type for a certain event is best not done, until there are certain property requirements, e.g., `postcondition()`[16]. In this case, the disjunctions introduced by the domain axiomatisation are a factor that can be influenced by domain-dependent heuristics. Considerations here involve exploration of more abstract dependencies in the conditions of actions than the calculus alone is able find by complex execution. The heuristics are preferences or filters on action types with respect to the actual goal and the abduction state. As EVE does not provide any explicit cost specification in the actions itself, these guidelines have to take account of weight of the actions implicitly.

5.5 Execution of the Simple EVENT CALCULUS of EVE

Omitting the success check in the persistence axiom (Figure 17) has, besides guaranteeing termination, a further side effect on the behaviour of the proof procedure. The basic properties of the axioms that are involved in this observation can be stated as follows:

1. `clipped()` no longer depends on a `fails()` subgoal.

[15] the current version of EVE does not provide explicit specification of the cost of actions, see Section 12.2 for a discussion

[16] In fact, this renders the planning procedure goal-driven, and is thus an instance of backward-planning.

2. `fails()` therefore only occurs in the negative mode as a subgoal of `holds()`.

3. `holds()`. Nested, negated `holds()` subgoals within `fails()` turn positive again and preserve the invariant 2.

4. `clipped()` also only appears negated by the same invariant 2.

5. `holds()` and `fails()` as already proved goals will stay correct as long as their `clipped()` subgoals remain so.

6. `clipped()` as a negated goal can only introduce `out()` as a positive subgoal.

This leads to a lemma that reveals the positive effect of EVE's simple theory of time and action on the proof procedure:

Lemma 5.1 *Any* SLD*-rule in* SLDNFA$^+$ *with a fair treatment of choice points that prefers* `fails()` *selection to* `clipped()` *will, by executing the simple* EVENT CALCULUS *of* EVE *and without any consistency check of already elaborated negated goals, result in a planning procedure that is sound with respect to the worst case assumption about failed events and strong nonlinearity. Completeness is restricted to least commitment, finite plans, finitely many action types and finitely many action type conditions.*

Proof. As completeness is not affected by omitting the repeated consistency proofs of negated goals, we are focussing on soundness. Assume that a proof of the procedure succeeded and regard the collected negated goals whose expansion involves the abduction state (other negated goals are not affected by later abduction steps and remain correct). These are only of type `fails()` and `clipped()`. Observation 5 shows that general soundness only depends on the correctness of the `clipped()` parts. As soon as some negated `clipped()` goal is selected, there is no possibility of selecting any further `fails()` goals later. This is due to observation 1 and the restriction on selection rule. From this stage on, additional abductions inserted into the abduction state can therefore only be of type `before()` (due to 3 and 6) and thus only extend the knowledge about the transitive, anti-symmetric ordering of events. No further events are introduced into the solution plan. Suppose now encounter a negated `clipped()` goal to be proved, any further extension of the temporal relation between the known events will not destroy this persistence anymore, since `out()` has already considered all possible destroyers of the wanted property by reasoning over all possible linearisations. □

This planning procedure delivers an algorithm also found in traditional approaches to nonlinear planning: one first expands the plan to match goals and preconditions; later one finds conflicts and resolves them. The advantage of the logic-based version, however, is the additional capability to realise plan analysis, evaluation and modification gained by the flexibility of theorem proving. Axiomatisations are furthermore much easier to extend than fixed procedural approaches — the intended research with respect to event duration and conditions (Section 12.2) is an example for this.

5.6 Execution of the Extended EVENT CALCULUS of EVE

The basic property that allows a simplified proof procedure in the case of EVE's simple calculus, the independence of `clipped()` from `fails()`, does not hold in the extended version. Possible positive occurrences of `fails()` and `clipped()` goals trigger abduction steps that intentionally destroy some persistence. Correctness of

already proved negated goals is therefore not self-evident — the consistency check is required occur.

Close investigation of the general behaviour now reveals inefficiency. Events E that are recognised as initiators for a property P to hold could destroy another persistence PE. The attempt to yield PE via failure of E spawns a search space that is condemned to fail a priori as the persistence from E to P depends on the success of E. Unrestricted abduction therefore leads to unnecessary computation. If the proof procedure could predict the events E that will form the basis of persistence checks, it would at least be possible to disable abduction in positive `fails(E)` goals. Only events that are not 'in use' (perhaps within a modification task) are tried for failure in this case.

If the recognition of failure is alone sufficient, abduction could be generally disabled within the expansion of positive `fails()` goals. A similar situation as in the simple calculus version arises:

1. `holds()` and negated `fails()` goals as already proved stay correct as long as the negated `clipped()` subgoals remain so.

2. `clipped()` can only introduce abductions by its `out()` subgoal.

Lemma 5.2 *Any* SLD*-rule in* SLDNFA$^+$ *with fair treatment of choice points that prefers negated* `fails()` *selection to negated* `clipped()` *will, by executing the extended* Event Calculus *of* EVE *without any consistency check of elaborated negated goals and restricting additional abductions not to happen within the subgoals of positive* `fails()`*, result in a sound plan procedure with respect to strong nonlinearity. Completeness is restricted to least commitment, finite plans, finitely many action types, finitely many action type conditions and only the recognition of failed events.*

Proof. The argument is similar to that in the simple calculus case because of 1. Positive `fails()` and thus `clipped()` goals can only occur during elaboration of negated `clipped()` after all the negated `fails()` goals are expanded. Due to the restricted abduction (2) then, there is no possibility of introducing new events. Only the `before()` relation can be further specified and will not destroy the proved `clipped()` negations and their subgoals because of the treatment of incomplete temporal knowledge by `out()` in the persistence axiom. □

Completeness is of course restricted to pure event failure recognition. More solution plans can only be reached with the general abduction procedure. Another approach dealing with the improvement of the problems of persistence is *maintenance* [23]. See Section 12.2 for its definition and perspectives.

5.7 Summary

Execution of the Event Calculus with SLDNFA$^+$ demands certain customisation of the abduction step and the SLD-rules, e.g., the definition of abductive predicates, fairness by depth-first expansion and iterative deepening, extended least commitment and reasonable defaults. Besides incorporating domain-dependent heuristics, EVE's special versions of the calculus furthermore allow simplification of the proof procedure with respect to interference between abduction and negation. Based on the treatment of event failure, the results are planning procedures with different classes of solution plan.

6 SLDNFA$^+$ by Constraint Logic Programming

EVE's integration of layer 1 (the EVENT CALCULUS) and 2 (the SLDNFA$^+$ procedure) into its implementation platform, OZ, is more sophisticated than just building the theorem proof procedure and relevant data structures scratch. We have instead chosen to tie this logical framework closely to the higher-order, constraint-based setting of the OZ calculus. This is realised through transformation functions that convert clause-based logic programs such as the EVENT CALCULUS into constraint expressions such that, when interpreted with the machinery of the OZ calculus, they produce a sound SLDNFA$^+$ proof procedure that is complete for finite universe negation and fair treatment of choice points. This section introduces the basic language constructs with their informal semantics and then incrementally describes how the proof principles underlying SLDNFA$^+$ are preserved in the translation.

6.1 The OZ calculus

A new generation of programming languages have been developed in the last decade. These constraint logic programming (CLP) languages mainly focus on bringing logical approaches to the rest of the programming world by combining state-of-the-art paradigms into its framework without loosing the great expressivity of logic.

An example of such a constraint language is OZ, developed at the Programming System Lab at the Universität des Saarlandes, headed by Gert Smolka [11; 12]. The corresponding constraint machine, DFKI OZ:

- explores the whole CLP suite:
  - efficient *rational* unification: s=t,
  - *records*, *lists*, *numbers*, etc.: are supported,
  - conjunction: $E_1 \cdot E_2$,
  - *conditional*: **if** G **then** E_1 **else** E_2 **fi**,
  - *local* variables: **local** V **in** E **end**
  - *computation spaces*: a hierarchy of constraint stores allows information inheritance from root to leaves,
  - *ask*: no inconsistent state of the top-level computation space,
  - *abstraction*, predicates: **proc{A** $\bar{V}$**}** E_1 **end**,
- and extends it with features still uncommon in logic languages:
  - *concurrency,threads*: suspension/resume of computations
  - declarative as well as functional programming style: **fun**$\models\models$**proc**; *cells* model state,
  - object orientation: state and methods `{Object method(`$\bar{V}$`)}`.
- Finally, it even introduces handling of
  - disjunctions: **or** E_1 **[]** E_2 **ro**; on the top level, restricted to real disjoint, decidable situations, as well as in
  - *encapsulated search*: provides a variety of search strategies [4]. Disjunctive constraints are applied by cloning the appropriate computation space and distributing the sub constraints to the siblings. First-class computation spaces make the results manageable from the upper level.

Besides the CLP systems having a clean, logical semantics making them suitable for *program verification*, the following deliberations represent reasons to use Oz as the implementation base for EVE:

1. Strong nonlinear planning is only of use, if the linearisation happens as late as possible in the execution scheme, if necessary at all. Concurrency supports simple and natural execution mechanisms in this respect.
2. The constraint-based platform does not require expensive implementation of logical data structures, as the translation functions below show.
3. Concurrent, competing planning tasks give the system more flexible reactive behaviour.
4. Object orientation is the basic key for structured, modular programming. In EVE, it serves to model a transparent representation of actions, plans and planning services.
5. The agent model InteRRaP [19] which provides EVE's main application (Section 10) has been ported to Oz [28]. Such agent-oriented and reactive approaches require:
   - the modularity of object orientation,
   - the reactiveness of concurrency, and
   - fair and controllable resource scheduling.

6.2 Resolution in Oz

As just presented, the Oz calculus already provides treatment of conjunctions at the top level, disjunctions by encapsulated search and even $CET_0 \subset RAT$ equality. Thus, the transformation from C clauses into Oz-constraints is straightforward. See the translation scheme T_1 in Figure 31 [17].

We have already mentioned that we would like to take as much advantage of the host language as possible. Clauses are therefore directly translated into appropriate Oz-abstractions by replacing conjunctions, disjunctions, and equality assertions by their immediate constraint-based equivalent. Universal quantification of clauses is implicitly represented by the concept of abstraction: each *application* of an abstraction, the constraint-based replacement of the resolution step, produces fresh parameters. **`local`** produces a variant of its scope. Because encapsulated search requires the use of unary queries in order to propagate solutions back to the top level, we allow free variables $\mathcal{F}(D)$ in the body of a query. These are implicitly existentially quantified and mapped to the root variable `Sol`. A depth-first, all-solutions proof of the resulting Oz program can then be computed by applying the predefined abstraction `Solve.all.eager`:

```
{ForAll {Solve.all.eager Query}
 proc{$ solved(Sol)} {Browse {Sol}} end}
```

Of course, a translated version is more restricted than the general SLD case: the selection rule and choice point preference depend on the order in which Oz treats new conjunctions and disjunctions. Hence, the execution of conjunctions can be regarded as being concurrent[18] with preference given to a depth-first, user-predefined

[17] For each translation function $T_{i,s}$ on type s, there exists a homomorphous extension $\bar{T}_{i,s}$ with $\bar{T}_{i,s}(\bar{s}) = \bigcirc_{s \in \bar{s}} T_{i,s}(s)$ and $\circ$ the sequencing/conjunctive operation on s

[18] This holds true for the reduction strategy of Oz 1.x. As the current development version of the abstract machine adopts a sequential strategy with threading on demand, the behaviour of the translated calculi changes accordingly.

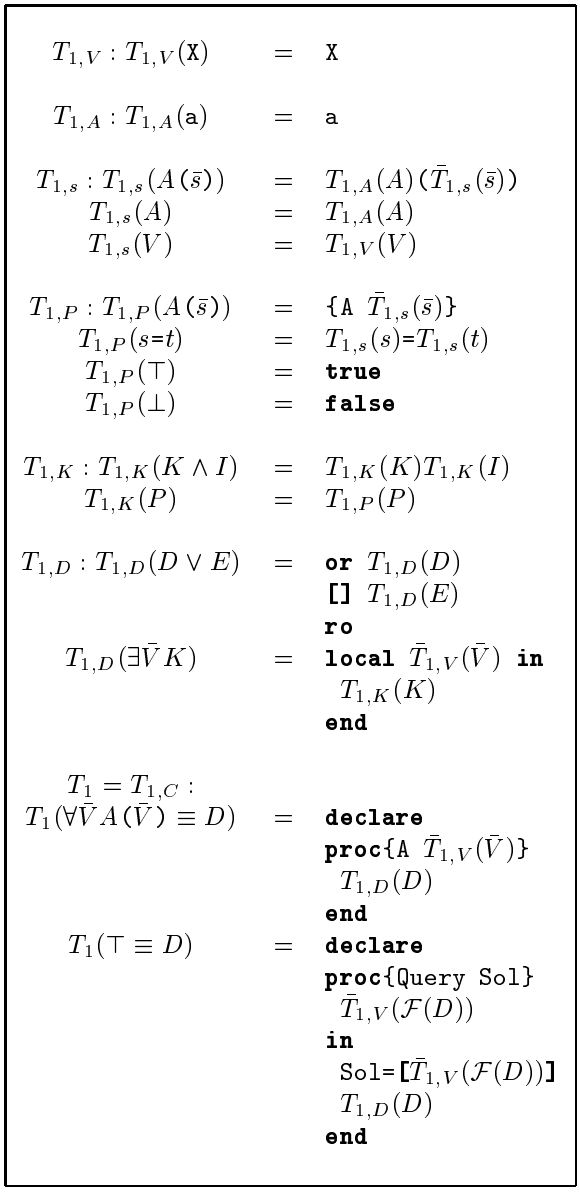


Figure 31: Translation Scheme T_1 – Resolution

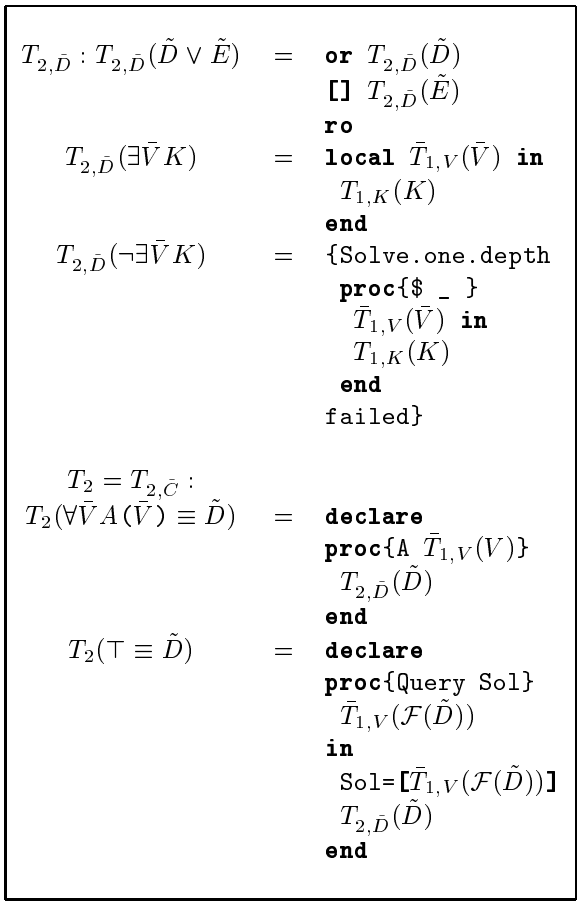


Figure 32: T_2 – Negation As Failure

order. Disjunctions create choice points, which are expanded as the driver procedure for the search process (`Solve.all.eager` in the example above) constructs the local computation spaces.

In spite of the loss of global control, we have gained a lot: efficient rational unification, concurrency, and the possibility of including the full computational power of Oz into the problem specification. We are now able to use logical as well as functional approaches; even conditional expressions and therefore *suspensions* are allowed — a method to reestablish influence over the data flow in the calculus.

6.3 Negation As Failure in Oz

Following the list proof procedures in Section 4, the next extension is to deal with first-order formulae in general by the introduction of negation. Let us first focus on the notion of negation as failure and SLDNF. As in most implementations, a negated goal proof recursively calls the proof procedure and inverts the success / failure result. In scheme T_2 (Figure 32), we have accomplished this behaviour by starting another encapsulated search within the current one. The application of `{Solve.one.depth Q failed}` only succeeds if the nested, depth-first, and one-solution proof of the abstraction `Q` fails. Note that the symbol `$` in Figure 32 represents the return value of the actual **proc** expression bound to the abstraction to negate. `_` produces an anonymous, fresh variable.

The groundness restriction is implicitly realized by the Oz code as the `Solve-Combinator`, the basic predicate that underlies each search procedure, returns, in the case of a nonground negated call, a stable computation space S (`stable(`S`)`), signfying that the information within the space hierarchy is not sufficient to guarantee correct distribution. Thus the unification with `failed` will fail.

6.4 Constructive Negation in Oz

First attempts to circumvent the groundness condition on negations by just delaying them until the end of the positive proof did not succeed because of the concurrent environment[19]. Whether a variable will ever be instantiated cannot be predicted in the Oz calculus. Furthermore, when going over to abduction, a switch between positive and negative goals has to be done without central control. This proof-'mode' information and the corresponding implications have to handled locally by the translation process.

To avoid the completeness problems of pure negation as failure, we propose the use of constructive negation as in SLDNF$^+$. The best way to handle this in constraint-based technique is by compiling two, logically inverted versions of each clause. Allowing the negation normal form requires the translation to introduce inequality constraints.

Within Oz, which regards constraints as actors on a blackboard, inequality maintenance follows from unification: **`if X=Y then false else true fi`**. As soon as the guard `X=Y` is entailed, the conditional could fire thus rendering the current computation space inconsistent (`false`). Disentailment just removes the actor from the blackboard (`true`) which is correct, since the Oz calculus only allows further monotonic extensions to the computation space.

In translating the negation normal form we apply De Morgan's as described in Section 4.3. Thus for the special cases $\neg(K \wedge I)$ becomes $\neg K \vee (K \wedge \neg I)$ to produce disjoint solutions. The translation also introduces the finite domain assumption F to handle universal quantification (see Section 4.3).

The only task that nested, encapsulated search still has to perform is enumeration of the finite domain over the local variables: $\bar{s}_1, \ldots, \bar{s}_n$. These substitutions are applied to $\neg K$ and build the new conjunctive goal to be further expanded. Scheme T_3 (Figures 33, 34, 35) uses this technique, given an appropriate unary Oz abstraction `FiDo`[20] for the domain enumeration.

The more efficient treatment of negation that we propose for SLDNF$^+$ (Section 4.3.3) can be also applied to the translation scheme T_3. The parts of the negated goal $\neg\exists\bar{V}K$ that only depend on the local variables $\bar{V}$, K_l, could be moved into the nested encapsulated search that enumerates the finite domain. The general look-ahead technique mentioned in Section 4.3.3, however, is not possible, because this requires full access to the content of abstractions. Without this *meta programming* support, our effort is only able to perform a much simpler precomputation during translation.

6.5 Abduction in Oz

We now describe the fully-fledged SLDNFA$^+$ translation. The first extension to the framework presented so far is the basic functionality of the abduction step included in the `MetaAbduction` predicate (Figure 36) to which the general requests will be mapped to using term expressions. Remember that the separation of clauses from the abduction set has the advantage of avoiding skolemisation and allows us to

[19] This would be equivalent to a selection rule that prefers positive to negative goals.

[20] We chose this rather canine name, because `FD` was already taken by predefined procedures in Oz.

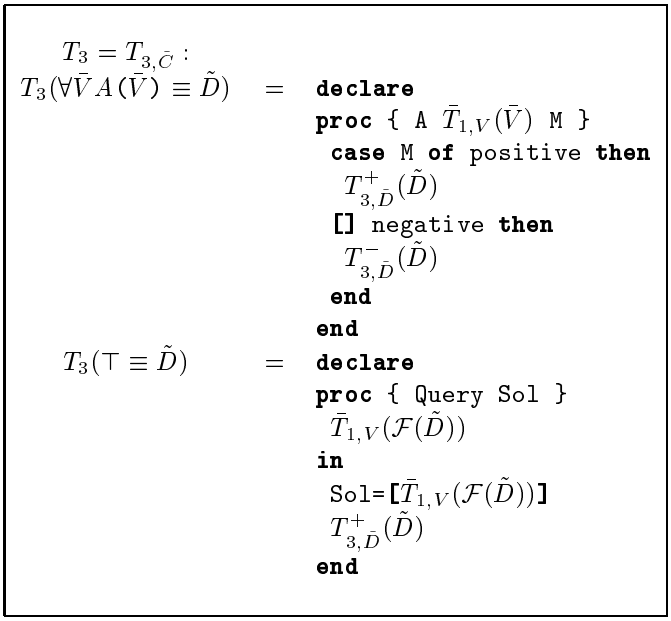


Figure 33: T_3 – Constructive Negation

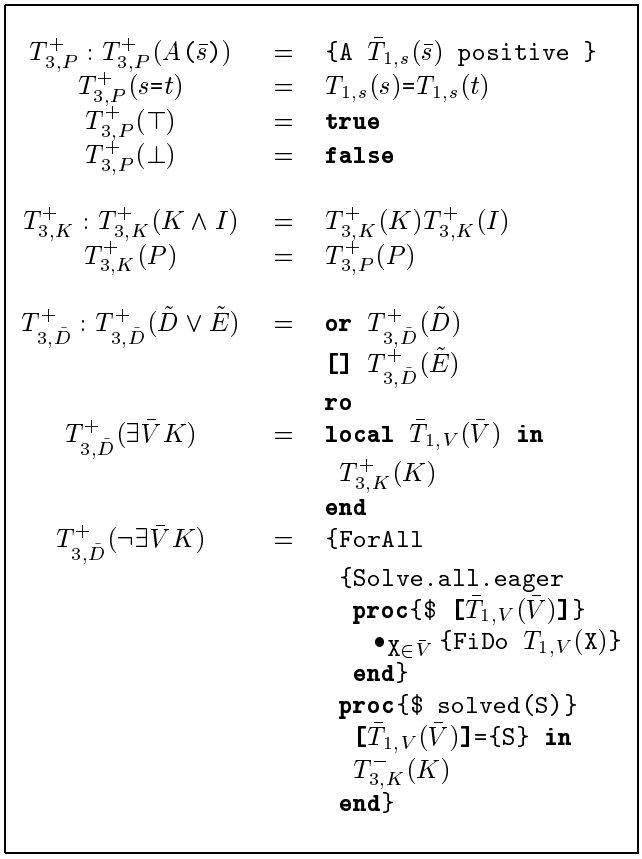


Figure 34: T_3^+ – Constructive Negation

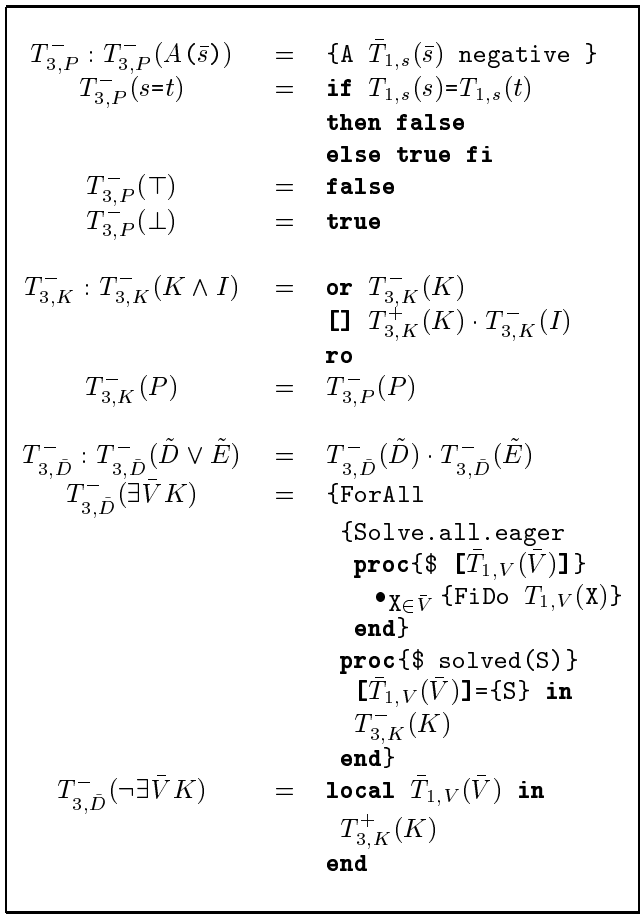


Figure 35: T_3^- – Constructive Negation

```
proc{MemberDisj Term List}
   case List of Head|Tail
   then or Term=Head
        [] {MemberDisj Term Tail} ro
   else false
   end
end

proc{MetaAbduction Term Mode AI AO}
   case Mode
   of positive then
      or {MemberDisj Term AI}
         AO=AI
      [] {Maintain Term|AI AO} ro
   [] negative then
      {ForAll AI proc{$ A}
                    if Term=A then false
                    else true fi
                 end}
      AI=AO
   end
end
```

Figure 36: The `MetaAbduction` Predicate

rely totally on existential quantification. The `Maintain` predicate is responsible for the abduction maintenance as well as the check for abducibility of goals (this is a simplification of the abduction procedures in Sections 4.4 and 4.5 a bit). The Addition of new assumptions has to be restricted to the `positive` mode.

The full constructive translation scheme extending T_3 with abduction capabilities and implementing the full SLDNFA$^+$ including the mapping of the abduction step to the `MetaAbduction` predicate is given by T_4 (see Figures 37,38,39). A special technique, however, remains to be described: the consistency check of negated goals with respect to abduction/negation interference. As the structure of the proof is not collected and thus no efficient check is possible, negated goals are first collected as abstractions and delayed as long as possible to decrease the amount of checking. Whereas a delay with respect to variable assignments is not possible, for reasons given above, the propagation of the abducibles creates some linearisation in the calculus and allows a delay with respect to accesses to the abduction set. The result is several proof stages that are managed by the driver function in Figure 40. Stableness of the abducibles and the collected negated goals signifies success.

The techniques for efficient negation of T_3 are also applicable to T_4, but follow the considerations of Section 4.5. Whenever there are subgoals that involve only local variables, but contribute to the abduction result, they have to reside in the global part of the translation.

6.6 Remarks

As can be seen from the translation schemes, the selection rule and the choice point preferences depend on the constraint solving mechanisms. By introducing abductive control flow, the user-predefined order of the applications is for the most part responsible for the conjunctive behaviour of abductive predicates. That this is not entirely the case, is due to the concurrent setting: a suspension only occurs, if a decision about (dis)entailment has to be delayed. Such suspensions can, however,

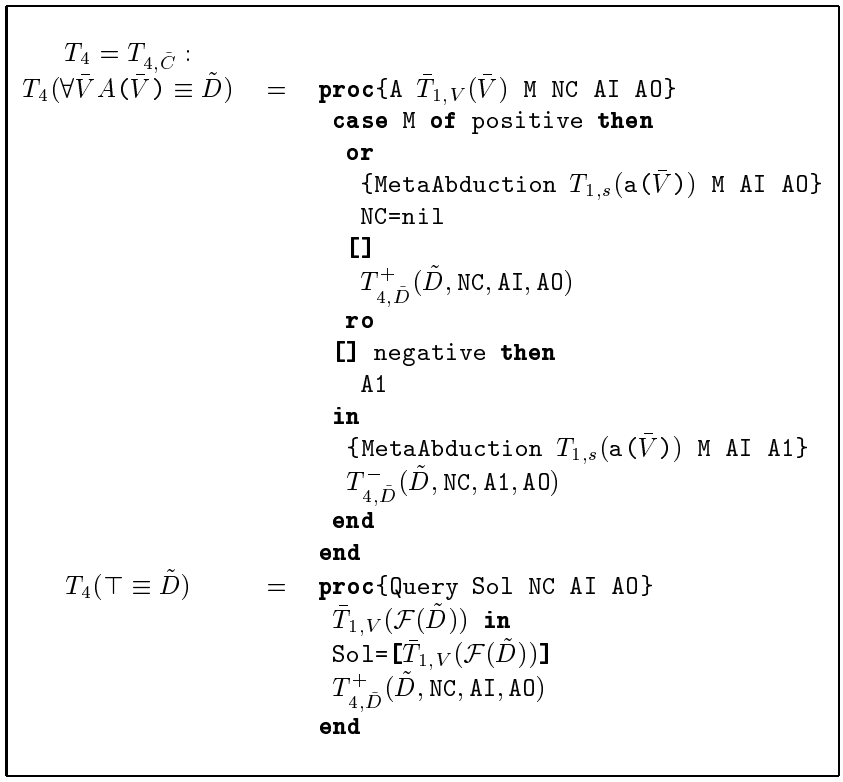


Figure 37: T_4 - Abduction

be made explicit using conditionals checking for determinance: **if** {Det Var} **then** *Exp* **fi**. indexDet The suspension occurs until the appropriate variable Var is bound and the conditional will then free the delayed constraint *Exp*.

Closely related to synchronisation is the question of heuristics. The fixed constraint solver rule is not suited for dynamic choice point preference[21]. The only place where the abduction scheme T_4 is in general able to directly influence any choice order is in the MetaAbduction predicate.

As minimality of the abducibles is one condition required of abduction, the addition of new abducibles has been placed in the second disjunctive choice after unification with already assumed literals. This is of course can only be guaranteed, if exactly this disjunction is solved in a depth-first manner which depends on the encapsulated search driver used.

Some control from within the local computation spaces influences the order of unification with current hypotheses. It works by reordering the appropriate abduction state, represented in our schemes by a list, before applying MemberDisj. Real pruning of the search space is obtained by omitting some of these assumptions. This is, of course, only sound in positive mode. The method could be extended to arbitrary, important choice points that can be guided by accessible data structures to install a broad opportunity for heuristics.

Recent research, however, has revealed some quite close relations between constraint logic programming and the concept of abduction. Section 12.4 tries to explain the main ideas that could lead to improved translation functions.

[21] The encapsulated search is of course influenced by the driver function, but there is no introspection into the computation spaces from the upper level (or at least what introspection there is, is very expensive).

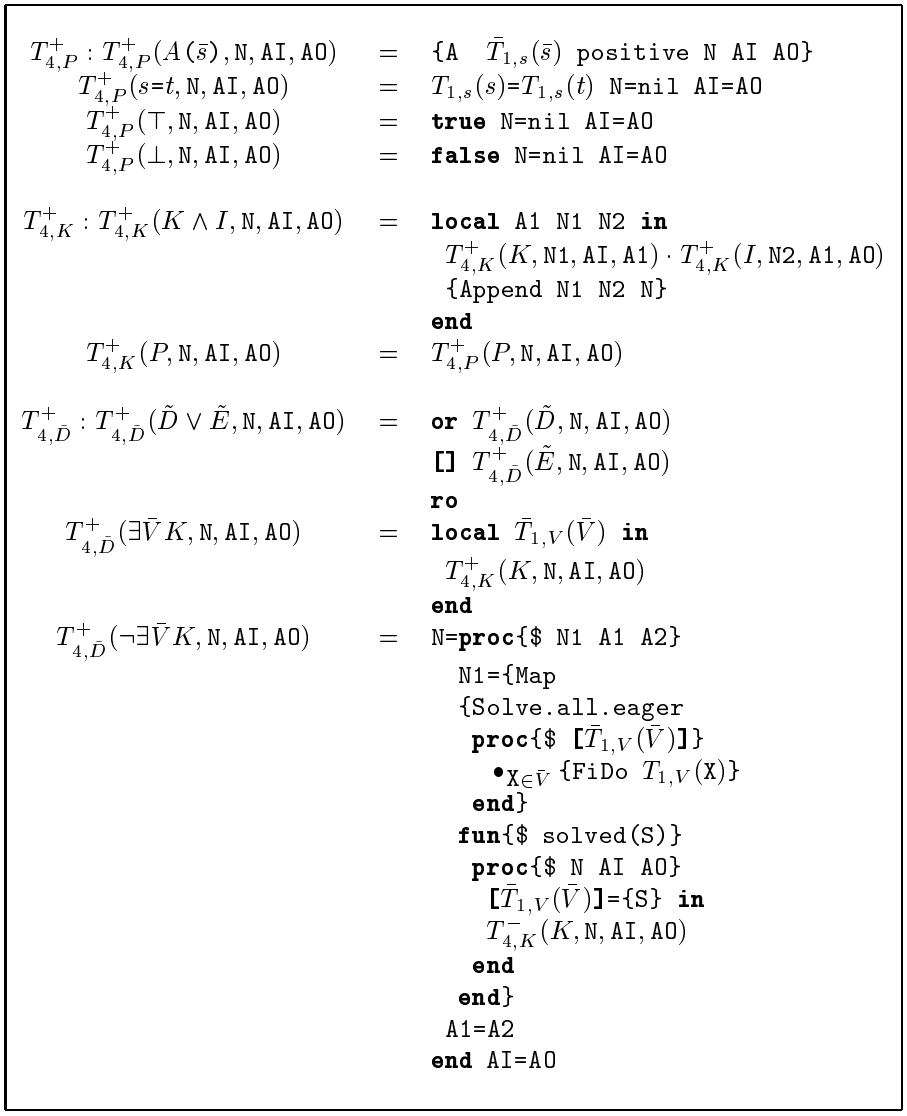


Figure 38: T_4^+ – Abduction

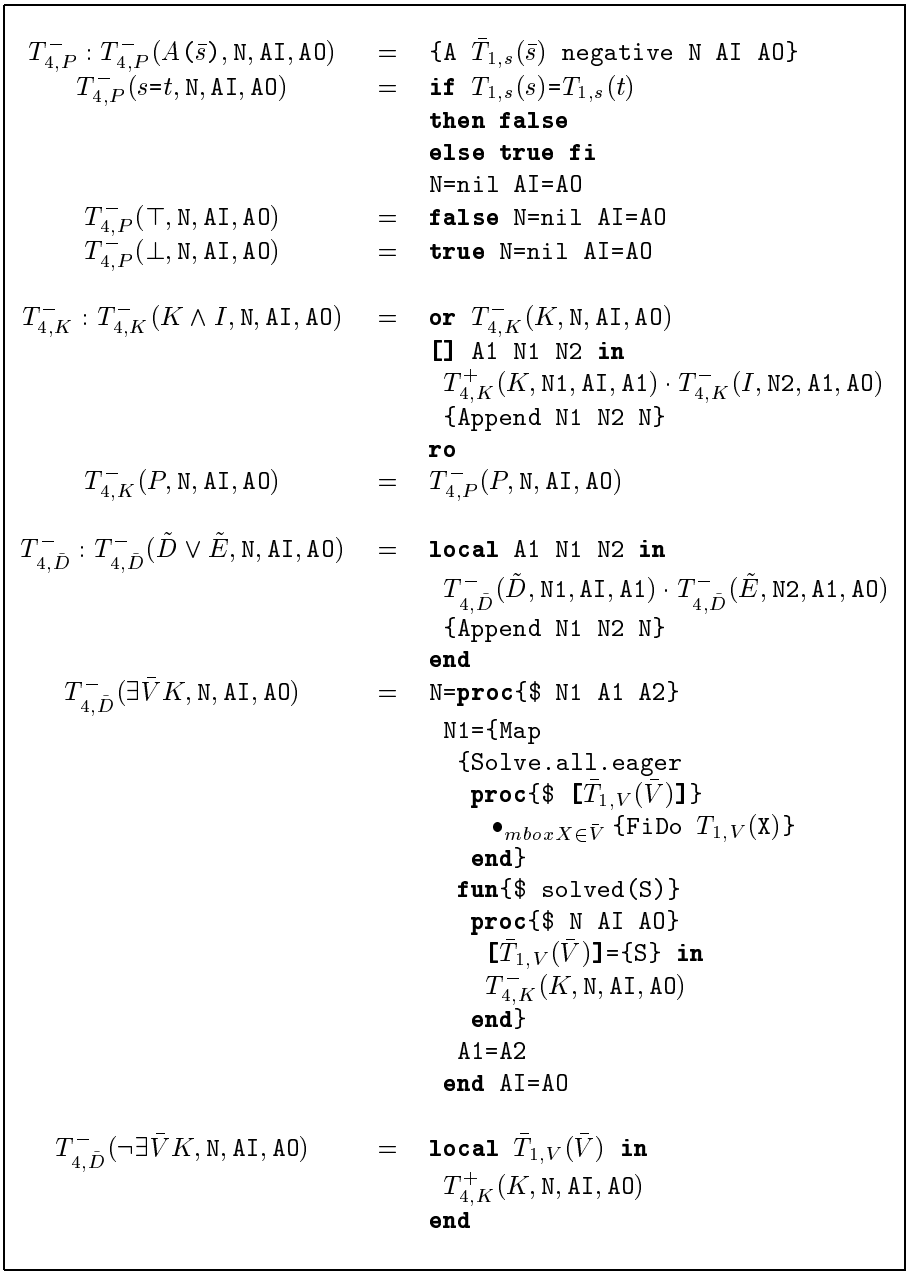


Figure 39: T^-_4 – Abduction

```
proc{Driver X Q AI AO}
   A1 N
in
   {Q X N AI A1}
   {Expand N A1 AO}
end

proc{Expand N AI AO}
   NO A1
in
   {FoldL N
    proc{$ N1#A1 N2 N3#A3}
       {N2 negative N1 A1 A3}
       N3={Append N2 N1}
    end
    N#AI NO#A1}
   if NO=N A1=AO then
      AO=AI
   else
      {Expand NO A1 AO}
   fi
end
```

Figure 40: Driver for Scheme T_4

6.7 Summary

With the implementation platform Oz that is highly suited to nonlinear, declarative planning within multi-agent domains, we have shown that it is possible to realize theorem proving facilities in a constraint-based setting providing abstraction and encapsulated search. The translation schemes presented cover not only basic resolution and negation principles, but also abduction. They solve soundness and completeness problems in SLD-resolution, SLDNF$^+$, and SLDNFA$^+$ using a local rewriting approach that allows execution by the Oz abstract machine without any further control. General heuristics that serve as dynamic guides to the choice rule are not immediately affected by these transformation schemes. The abduction step, however, can be controlled in several ways and serves as an example of how to influence the search process from within the computation spaces.

7 The Constraint-Based Event Calculus

The search procedure underlying EVE is not just the result of a plain transformation like those presented in Section 6. Besides additional techniques to influence the behaviour of the calculus, it is necessary to handle the object representation provided by EVE (see Section 8).

7.1 Meta-Programming

In Section 6.5, we pointed out that meta programming facilities in CLP are necessary to achieve efficient negation proofs,with or without the finite domain restriction. But there is another reason to use an extended treatment of constraint: it is possible to control choices made in constraint solving.

In EVE, a first attempt towards structuring of conjunctions was implemented. Again, the basic technique is abstraction and the idea is based on collecting and delaying negated subgoals in T_4:

```
proc{Clause Vars PosSubGoals NegSubGoals CriticalNegSubGoals Ā}
   PosSubGoals={MetaSubGoal1 Vars}|...|{MetaSubGoali Vars}|nil
   NegSubGoals=...|{MetaSubGoalj Vars}|nil
   CriticalNegSubGoals=...|{MetaSubGoalk Vars}|nil
...
end

fun{MetaClause Vars}
   proc{$ Ā}
      {Clause Vars AI AO}
   end
end
```

Instead of directly exposing subgoals to the underlying constraint calculus, this new scheme abstracts them using meta-level procedures (`MetaClause` — $\bar{A}$ are additional parameters including the abduction state propagation). Besides positive and negative subgoals, a more abduction-aware technique also marks the negations that may be affected by further abduction steps (`CriticalNegSubGoals` — those involving negative abductive goals) and separates them from other goals that either are not affected by abduction or that are subgoals of such marked abstractions and are therefore considered while checking the parents for correctness. An extended driver procedure with the ability to select the next goal to solve is now necessary for the execution. Research towards special purpose, complete meta programming facilities is not the main aim of EVE, since, in any event, the underlying Oz system can be relied upon to incorparate the latest techniques in constraint logic programming.

7.2 Heuristics

The control over the conjunction elaboration by using abstractions, as just presented, enables the incorporation of selection rules. As stated in Section 5.4, however, an approximate depth-first approach for selection and choice in a bounded search is fully sufficient for the purposes of the Event Calculus. Exceptions are the choice points that are introduced by abduction and the domain description that could be used to incorporate heuristics.

Whenever the abduction state or the content of the domain specification is accessed, Eve allows these guiding procedures to help. The approach is implemented within the search process, since introspection into computation spaces from the upper level is expensive. Thus heuristics turn out to be list-restructuring procedures

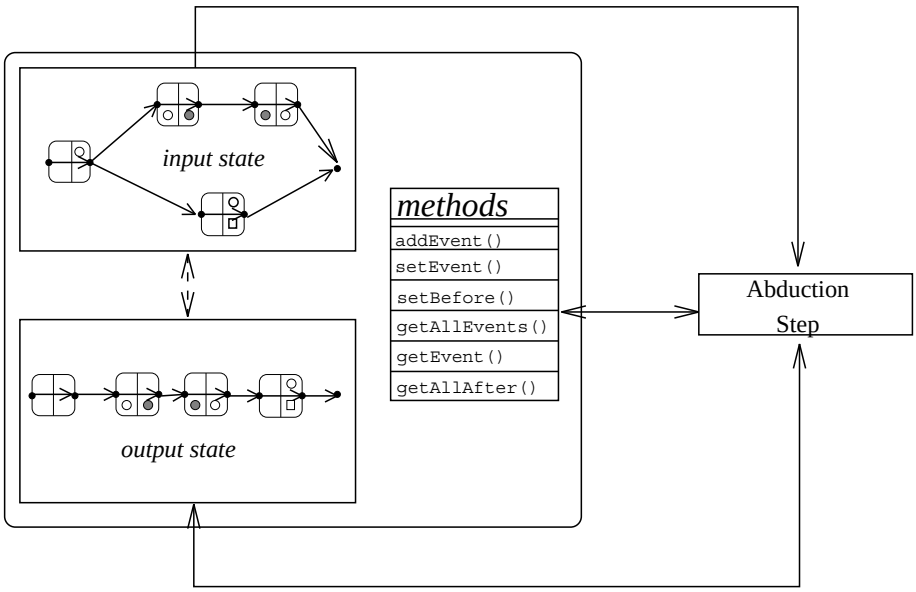


Figure 41: Object-Oriented Abduction

that are propagated into the search for planning. The lists constructed in this way later serve to spawn some portion of the search tree — the default heuristic is thus the identity procedure. Deleting elements from lists results in pruning of the search space. To enable reasonably informed planning, contextual information, such as the location of the choice point, the abduction state, the current goal, and the state of the knowledge base are passed to the heuristic procedures.

7.3 Object-Oriented Abduction

The capability of EVE's constraint platform, Oz, of bringing logic programming and object orientation together also influences the implementation of the abduction step in EVE. A list-based representation of abducibles is not only hard to maintain and inflexible as an interface to the module, given a reasonable object system as EVE provides (Section 8), this would also require translating between list and object based representations as data is passed between the top-level interface (the planning service) and the temporal reasoning calculus. Why not have an object replacing the set of abducibles?

That is, indeed, the EVE approach. Propagation of the abduction state is replaced by propagation of object state. Instead of accessing the maintenance procedure via terms, the abduction step is mapped to appropriate method calls. The methods automatically handle abduction maintenance including the default axioms (Section 5.2) in a far more efficient way. e.g., `{Maintain before(start(E1) end(E2))|AI AO}` is replaced by `{AI setBefore(start(E1) end(E2))}`. We call this technique *object-oriented abduction* (Figure 7.3).

Since objects are conceptually global entities, their use in encapsulated search and thus local computation spaces is problematical. Methods, in general, change this global state. Problems naturally arise when competing branches of the search tree try to modify this state. To handle this, rather than propagating an object's name, the pointer to the global structure, it is the object's state that is propagated as intermediate abduction results. As soon as access to the object is necessary, the encapsulated calculus has to instantiate a local copy with which it can interact and

whose internals can be read out again to serve as the next abduction state. The necessary state accessing methods that the internal data structures have to provide are `getState()` and `setState()` (see Section 8.3).

7.4 The default Abducibles and the Knowledge Base Interface

Due to the special behaviour of the `initial` and `end` events with respect to the action data structures, to be described later, their conditions have to be preserved within the EVENT CALCULUS. Whereas `end` only serves for the description of time points and thus the conditions of its type are empty, the `initial`* situation's post-conditions are mapped to the generic knowledge base interface.

In addition EVE assumes that the knowledge base is an object that provides a method to disjunctively (it is an equivalent to `initiates(initial*,P)`) match its contents with a property term: `getfactdisj(P)`. Again, this encounters the problem of object communication within search. Since there is no possibility of distinguishing state-changing from state-preserving methods (the initial state should be frozen and thus not changed during planning), the state propagation technique of object-oriented abduction has to be applied again. The state accessing methods therefore also have to be supported by the knowledge base object. Furthermore, a dynamic knowledge base can be made much more abduction-aware. Hypothetical reasoning about the start situation, as requested by approaches discussed in Section 12.2, becomes possible.

EVE replaces `terminates(initial*)` by a constructive negation access to the positive facts in the KB. This has the advantage of avoiding explicit termination actions to install a `holdsfalse()`. If there is much information from the start situation and the access to the knowledge base has not been improved for partial instantiations on the properties, it is necessary to have a switch to dis-/enable this feature.

7.5 Summary

EVE extends the clause-based translation approach with a novel meta-programming approach to achieve additional control over the computation flow. Furthermore, the incorporation of heuristic procedures that have influence on the important choice points in the EVENT CALCULUS was presented. The extended theorem proving facilities are supplemented by the notion of object-oriented abduction that resolves conceptual discrepancies between object orientation and encapsulated search and provides a generic knowledge base interface.

8 Object-Oriented Representation

In order for the computational layers to be usable, the rather crude representation must be given a user-friendly, customisable, interactive interface, that provides access to the top-level as well as in the system internals. The object-oriented principles realized in Oz, one of the first logic languages to support OOP (object-oriented programming), play an important role in this respect and form the base for EVE's service class and the hierarchy of plans and actions.

8.1 Representation of Properties

The Oz universe does is not restricted to atoms and finite terms as in pure clause-based logic. The full range of rational trees, records, numbers, tuples, lists are predefined and handled efficiently. It would thus be unwise to restrict ourselves to the pure first-order term syntax.

A property denoting that entity `robo1` is holding entity `blocka` can thus be more clearly defined as `holding(agent:robo1 object:blocka)`. In fact EVE allows any Oz term expression to be used as a property description, and relies on the built-in unification semantics, e.g., lists do not behave like sets.

8.2 The Planning Service Class: `EVE`

Using a planner in a concurrent environment should not be restricted to interfacing a single precompiled module that is locked during processing. In the case of EVE, there exists a special *class* `Eve` (Figure 8.2) predefining the state and methods needed to build a concrete planner instance by inheritance. Such an object plays the role of a planning module by accepting service calls, via method application, that request plan analysis (`analysis()`), synthesis (`synthesis()`), evaluation (`evaluation()`) or modification (`modification()`). Some information in those calls has to be preserved in order to perform the requested task (e.g., `!Term`) — solution generation is done asynchronously by instantiation of the appropriate arguments (e.g., `?Plan`). The method `next()` is responsible for generating a further solution plan for the previously specified synthesis/modification problem.

If the accessed object is busy (`?Busy=True`), the environment always has the choice of halting (`halt()`), continuing (`continue()`) or abandoning (`stop()`) the computation concurrently in progress. It is also possible to construct another instance of `Eve` and request a search concurrent to the former one.

To interface an external knowledge base object, the user has to register it: `setKnowledgeBase()`. The compatibility with the Event Calculus, however, places special requirements on the KB object that are explained in detail in Sections 8.5 and 7.4.

Finally, the service class includes many ways to influence the search process in order to be flexible:

- The types of the domain events that are allowed to be abduced in synthesis and modification are registered with `setAllowedActionTypes()`.
- The overall number of events that are allowed to be abduced during the planning process is restricted by an upper and a lower bound to get a fair and thus relatively complete planning procedure (see Section 5.4). These limits are iteratively increased as the search goes on for further solutions; their initial values, however, can be set with `setActionBounds()`.
- Another parametrisation to the search, is a specification of the version Event Calculus to be used. The trade-offs in terms of computational expense and

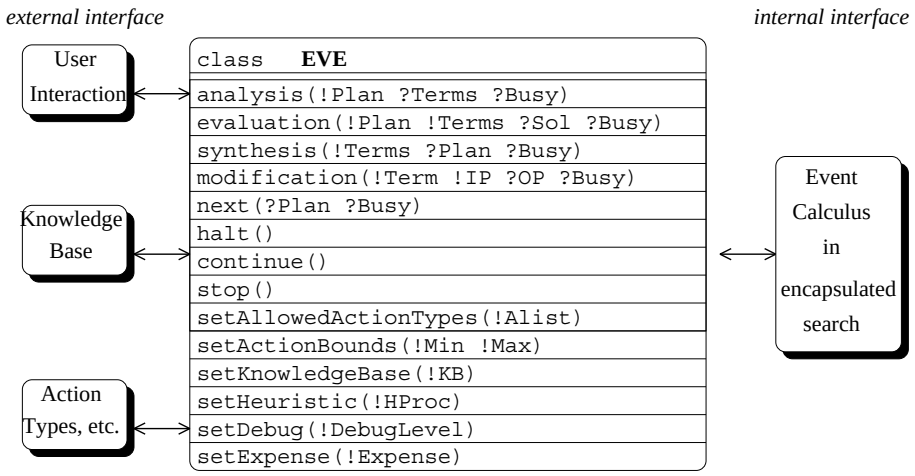


Figure 42: the Plan Service Class `Eve`

expressivity are described in Section 3.7. `setExpense()` switches between the different axiomatisations and proof procedures:

  - the simple EVENT CALCULUS with the simplified proof procedure (see Section 3.5 and 5.5).
  - the extended EVENT CALCULUS with the simplified proof procedure (see Section 3.6 and 5.6).
  - the extended EVENT CALCULUS with the full proof procedure (see Section 3.6).

- Heuristics are incorporated as procedures (see Section 7.2) and introduced to `Eve` with `setHeuristic()`.

- Since there are many traps and pitfalls in designing planning domains, we intend to integrate a user-friendly debugging tool into the `Eve` class that interacts with the OZ `Solver`, a graphical search debugger implemented in the latest OZ version. For the moment, the debugging consists only of control and abduction flow information whose level of detail can be selected with `setDebugLevel()`.

8.3 The Event Class: `UrEvent`

The planning facilities themselves, as well as the event (action) and plan data structures and their types are realized as objects in EVE. Their functionality is concentrated in a generic class `UrEvent` (Figure 8.3) whose services can be distinguished by the following functionality:

- First there are **definition** methods that allow the modelling of new actions and types. The basic step is the introduction of roles that serve as parameters to the event object. Because of their unique name, e.g., `{Event addRoles(roles(agent:_))}`, one can refer to those roles in the condition definition section:

```
{Event addConditions(negprecondition
```

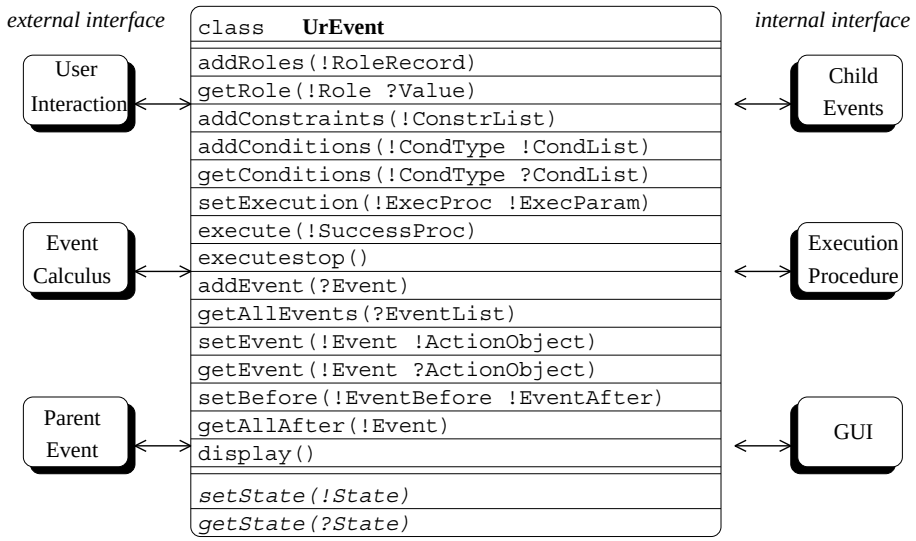


Figure 43: The Event Class `UrEvent`

```
[busy(subject:evevar(agent:_))])}
```

Subexpressions matching the pattern `evevar(`$< Role >$`:_)` are replaced by the appropriate value of role $< Role >$. Besides their appearance in conditions, these expressions can also be used to describe the parameters to the execution procedure (`setExecution()`) that is registered and triggered in the case of primitive action execution. The case of plans is more complex as each plan contains several sub steps introduced with `addEvent()` and associated with a certain action type via `setEvent()`. Furthermore the temporal relation between these children is refined by `setBefore()`. To map the roles of a plan to those of its child events, there exists a special method `setRoleMap([`...`[`$< Prole >< Event >< ERole >$`]`...`])` that is responsible for unifying the value of $< Prole >$ to the sub event $< Event >$'s value of role $< ERole >$. Finally, the expressivity of the object system is increased by placing constraining procedures via `addConstraints()` on the different roles. Constraining procedures receive the complete role descriptions including their values and are thus able to construct arbitrary Oz constraints around them. These can be used to demand domain/type restrictions (like finite domains, etc.) or even do arithmetic (see Section A.2).

- The second type of method calls allows **interaction** with the already defined actions and plans. Whatever information the user has put into a definition is, of course, accessible again (`getRole()`, `getConditions()`, `getAllEvents()`, `getEvent()`, `getAllAfter()`, `getEvent()`). Undetermined values that are published in this way can then be instantiated. A graphical overview of the internal object structure is obtained with the `display()` method (see Figure 8.3). Perhaps the most important interaction, however, consists of execution of actions or nonlinear plans. In the case of primitive actions, `execute()` will just trigger a registered procedure that reports its success. The nonlinear plan execution scheme of EVE, however, takes advantage from the concurrent environment. All child events ready to execute are triggered via `execute()` and advised to give their success results back. As soon as such a substructure sends a positive reaction, the next possible execution stage is computed and

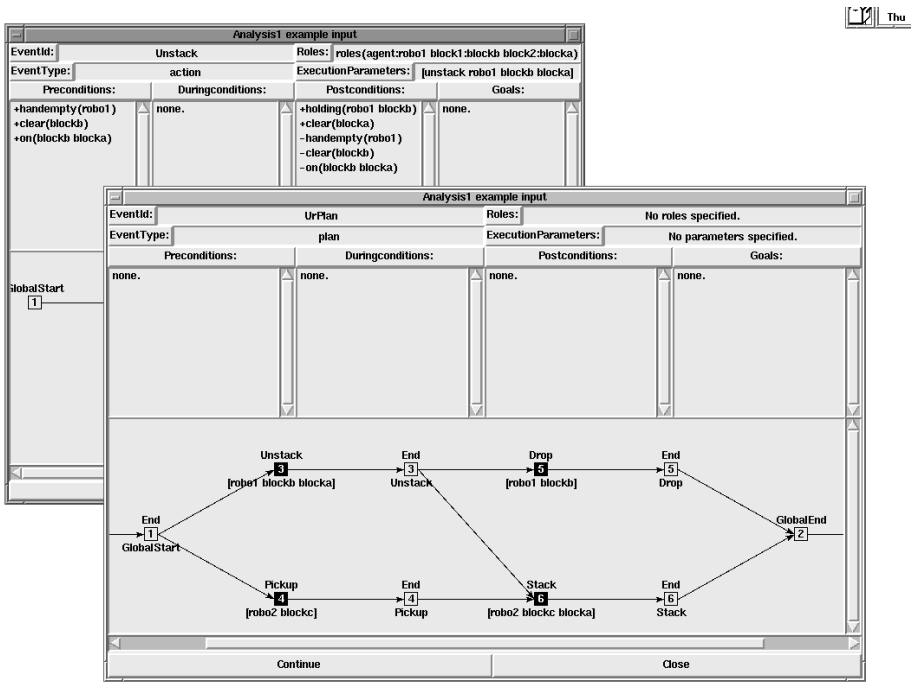


Figure 44: A `display()` Result

triggered until all children have been successfully executed. In the case of a failure report, the early achievement of requested goals, or an interruption of execution by the user (`executestop()`), a symmetrical, recursive application of `executestop()` is started to notify the still active sub events. Linearisation thus takes place, if necessary at all, on the deepest level within the execution scheme.

- Additionally, there is the need for **reading and writing the overall state** of an object, if it is interfaced or even changed during search, as explained in section 7.3. The respective methods are called `getState()` and `setState()` and are only of internal use. Of course, the state propagation also covers the sub event representation in nonlinear plans. Instead of pointing to a child object, the plan rather contains its state. The methods that handle the association of sub events (`setEvent()`, `getEvent()`) respect this requirement as they are also based on state access. Furthermore, the execution scheme requires additional instantiations of child events.

From the uniform description of plans and actions, it can be seen that EVE follows a hierarchical approach with the only distinction between the two being at the execution level. However, to inherit this specific execution feature, there are two subclasses of `UrEvent` available: `UrAction` and `UrPlan`.

The interactive definition of events and the treatment of roles needs an extended inheritance mechanism. The straightforward mapping of types to classes and instances to objects does not work because of EVE's need for interactive definition. Therefore, the inheritance procedure is able to treat any event object like its own type. Instead of just relying on the defaults given by an appropriate class definition, `EveCreate` has, in addition, to handle the 'implanting' into the descendant of the current object state that is dependent on recent, interactive method applications. Uninstantiated, inherited roles, however, have to be decoupled from the values to

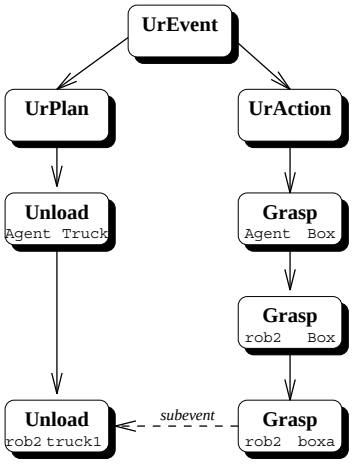


Figure 45: An Example Event Hierarchy

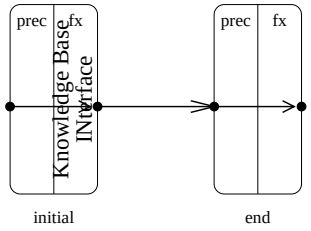


Figure 46: The Default Abductive State

which they are bound in the ancestor to provide a reasonable inheritance scheme. An example hierarchy is given in Figure 8.3.

8.4 The Default Abducibles

In the axiomatisation of the EVENT CALCULUS, we discussed the introduction of `initial` and `end` events to implicitly represent situations (Figure 46). Ubiquitous in the planning problem, they are default sub events for members of class `UrEvent`. The EVENT CALCULI (Section 7.4) and the `UrEvent`'s built-in rules handle their conditions and the default assumptions mentioned in Section 5.2.

8.5 The Knowledge Base Object

EVE expects the knowledge base that represents the initial planning state also to be an object. To guarantee a functional interface between such arbitrary modules and the translated EVENT CALCULUS, a few guidelines have to be followed. They are shown in Figure 8.5.

Whenever the `initial` event is chosen during execution of the EVENT CALCULUS, the problem arises of accessing the content of the propagated knowledge base using the property syntax (`initiates(initial* ?Term)`). A mapping from properties to KB's contents has to be done. This is, of course, up to the user as it depends on the internal structure of his KB. The method `getfactdisj()` is responsible for incorporating the necessary transition disjunctively due to the behaviour of intended `initiates()` semantics.

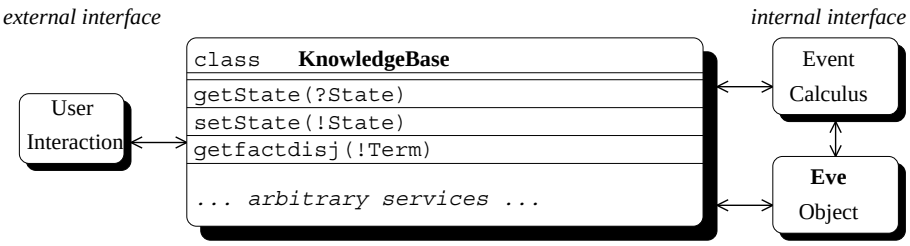


Figure 47: An EVE-Compatible Knowledge Base

The same considerations concerning objects in encapsulated search in the action case arise again with respect to the knowledge base interface (Section 7.4). State related methods are necessary and are also called `getState()` and `setState()`.

8.6 Summary

This section discussed the current object-based representation, its internal structure and services, and its interface to user modules, the knowledge base, and the previously described logical layers of the planning process. We presented the planning service class and the event class whose object system builds a base for interactivity, resource control, and flexible, hierarchical plan representation. Furthermore, the requirements on a knowledge base to be compatible with EVE were explained.

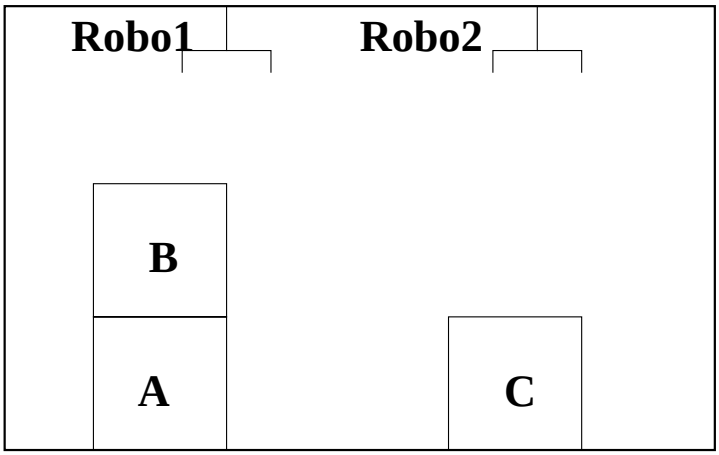


Figure 48: the Multi-Agent Blocksworld

9 EVE in the Multi-Agent Blocksworld Domain

One of the first ever scenarios for robot problem solving and thus planning was the blocksworld [32]. Different blocks arranged on a desk must be rearranged by a robot that is able to grasp exactly one of them. Relations of the blocks are described by the terms `ontable()`, `clear()`, and `on()`, whereas the situation descriptions involving the robot are `handempty()` and `holding()`. Given a start situation and a goal description, the planning task consists of choosing appropriate, sequenced instances of the possible actions of the robot:

- If the robot hand is empty and a certain block is standing on the table, the robot can `Pickup` the block.
- If the robot hand is empty and a certain block is standing on top of another one, the robot can `Unstack` the first block from the second one.
- If the robot hand carries a certain block, the robot can `Drop` the block on the table.
- If the robot hand carries a certain block, the robot is able to `Stack` it onto some other one.

An important characteristic of this time-honoured scenario is the possibility of goal interaction, when actions interfere with one another. Some researchers even regard the problems of the blocksworld as being more complicated than common real-world tasks. To demonstrate the nonlinear facilities of EVE we extend the domain by allowing several robots to be involved (Figure 9).

The concrete axiomatisation of the domain can be found in Section B. Below we list the major findings of our work applying EVE to the test problems of the multi-agent blocksworld. More detailed performance benchmarks and the investigation of the search processes with the latest Oz debugging tools will be the focus of future research.

- Synthesis, modification, analysis and evaluation are as sound and complete as expected by the theoretical considerations.

- Nonlinear functionality is used. Whenever it is reasonable, several agents are engaged in activity at the same time.
- The solution plans in synthesis/modification are enumerated according to the number of steps they contain. Assuming uniform action execution costs, the optimal plan is found.
- Solution management is handled reasonably by the bounded search — no solutions are presented twice.
- If situations are symmetric to several agents, there will be also symmetrical solution plans.
- Interesting problems like the Sussman Anomaly of which there are two versions in the multi-agent blocksworld can be solved with speed.
- The search space has, however, a high branching factor and thus explodes early depending on the number of abductions allowed. The pure, uninformed depth-first approach takes far too long to be useful for hard, practical applications.

10 EVE in a Multi-Agent System

Distributed Artificial Intelligence (**DAI**) is an area within **AI** that is growing in popularity. This is probably due to the common trend in computer science of dividing complex problem solving into several, independent tasks (*Distributed Problem Solving*) that have to interact in order to build a globally correct solution. The special contribution of **DAI** is now to model these *autonomous* problem-solving capabilities as cognitive entities, the *agents*, that interact with their environment (possibly including a human user) and each other. The result are the Multi-Agent Systems (**MAS**) that especially demand *cooperative* and thus *communicative* facilities from their elements in order to acquire the emergent functionality for the top-level problem solution.

The corresponding agent architectures therefore require the integration of most of the rather separate research questions of **AI** in order to model a cognitively complete scheme. This is especially of the task of merging *reactivity*, the ability to make fast decisions fast based on new perceptual data, and *deliberation*, the possibility of obeying more abstract *goals* and developing long-term perspectives for acting. Whereas reactive behaviour seems to be functionality that resides on an unconscious level in cognitive systems — routine tasks that can be understood as procedural knowledge — deliberation poses problem specifications that belong to decision theory and planning. Finally, cooperation and communication seem to complicate these questions by allowing social competence and the ability to exchange beliefs about multi-agent plans.

One major factor in the development of EVE has always been its integration into a MAS. EVE's nonlinear execution model and the flexible, domain-independent implementation make our planning system most suitable for that purpose. Our future research activities will therefore focus on the integration of EVE into the InteRRaP agent architecture, developed within the CoMMA-MAPS project, and the exploration of their interplay. In this section, we briefly present the key concepts of InteRRaP and explain a first approach to placing the planning module within the so-called local planning layer of the agent architecture. First evaluation results from the *loading dock* scenario reveal the flexibility of the new agents and show some of the critical questions that arise when bringing reactivity and deliberation together.

10.1 The INTERRAP Architecture

The major paradigm used in the pragmatic InteRRaP agent architecture [29; 19] is that of a *layered* architecture for integrating reactivity and deliberation (Figure 49). Each layer introduces an extended abstraction level to the reasoning that happens on its lower components:

- the Behaviour Based Layer **BBL** is responsible for performing routine tasks in a procedural manner. Its processes are called patterns of behaviour and they obey the input-output functional scheme that guarantees a reactive system.
- the Local Planning Layer **LPL** introduces the notion of long-term goals and has the ability to develop intentions (plans) that influence the **BBL**'s short-term behaviour in order to match these goals.
- the Cooperative Planning Layer **CPL** represents the social competence of the agent. It extends the agent-centred perspective of the **LPL** with the ability of reasoning about other agents state of mind, communicating about own intentions and developing joint activities.

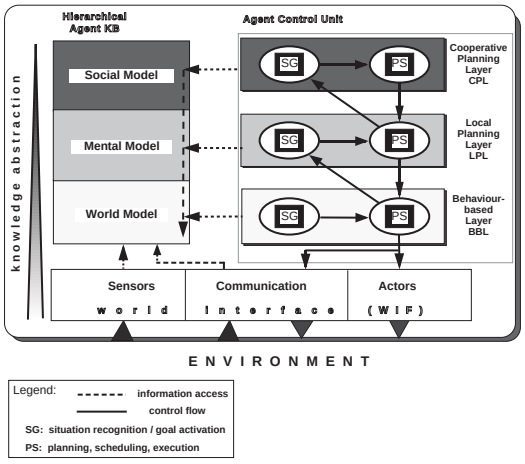


Figure 49: The InteRRaP Architecture

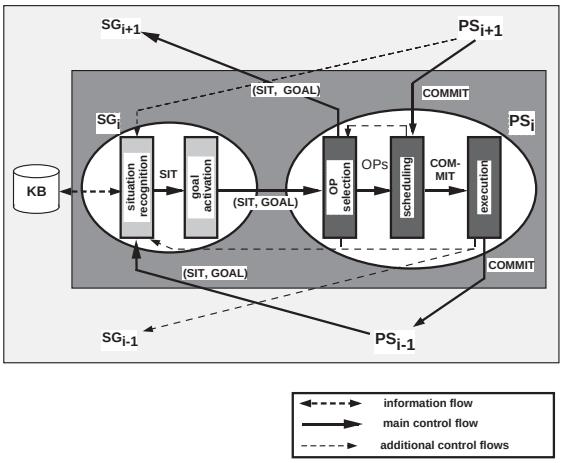


Figure 50: A Layer within InteRRaP

The agent model is supplemented with a hierarchical knowledge base (**KB**) that provides the agents beliefs in several, layer-specific abstraction levels and the world interface (**WIF**), the linking module between the agent and its environment.

The structure of each layer reveals some common design principles (Figure 50). In particular the separation of functionality into the Situation Recognition & Goal Activation (**SG**) and Planning & Scheduling (**PS**) and research into their interaction have been an important topic within the development of this architecture. The **SG** performs the task of receiving layer-specific signals that indicate a situation change and derive new goals from their integration. These are now used to trigger or reconfigure the **PS** process to develop or change the intentional structure within the layer and therefore guide its execution. Raising signals is of course restricted to happen only between a neighbouring layers. They consist of a part that is related to the situation change and a part that carries information about goal activation. In the following, we will see how EVE fits into this framework by integrating it into the LPL. The implementation was carried out within the ALADIN system [28], a concrete InteRRaP skeleton that is also written in Oz.

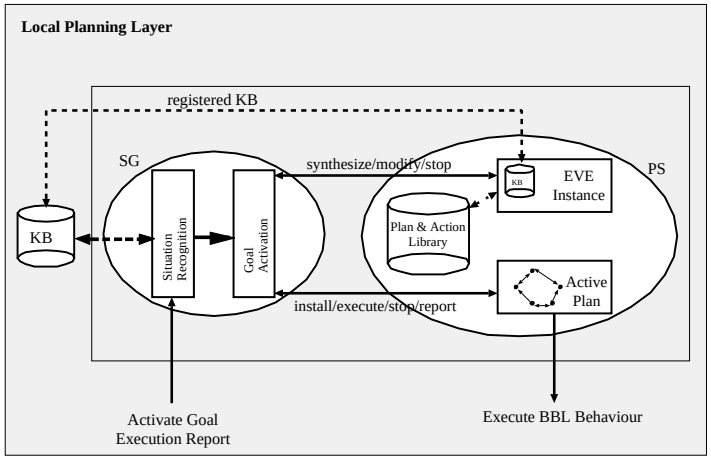


Figure 51: EVE within the **LPL**

10.2 EVE within the Local Planning Layer

As the design of the generic layer (in Figure 50) suggests, a planning module within the local planning layer has to be a part of the **PS** process. We have already mentioned, that the **CPL** has been left out of this initial integration attempt that is schematically shown in Figure 51. The following places are instantiated:

- the **KB** with the mental model of the agent has been extended according to Section 8.5. It is now possible to create clones of it within the search process of the planning module. Initialisation of the agent will register the **KB** at the planning process within **PS**.
- the **SG** process
  - Situation Recognition reacts to changes within the knowledge base as well as messages from the **BBL**. Recognised patterns include contextual information of desires, execution results from commitments to the **BBL** layer and explicit goal activation requests. These patterns are initialised on creation of the agent.
  - Goal Activation takes the output of Situation Recognition, forms new goals expressed in EVE's representation of properties and propagates the commitment results from the **BBL**. It interfaces the **PS** process by controlling the planning module and the active plan. As the current approach does not interleave execution and planning, new goals are queued until the active plan is executed. If the active plan is empty, a new planning service to solve the conjunctive queued goals is activated. As soon as a solution plan is presented, it is introduced as the new active plan and its execution method is called. Results from the commitments are mapped to appropriate method calls to the active plan to continue its work. Success or failure of the active plan are given back to the initiator of the respective goals. Initially, the goal queue is empty.
- the **PS** process
  - a single EVE module (see Section 8.2) builds the heart of the **PS** process. It is concurrent to the rest of the agent's components, but controlled by **SG**. On demand, it will execute a plan service (synthesis or modification)

until either a solution has been generated or the service call is interrupted. By looking up plan proposals in the plan library, predefined plans can be stored indexed according to their intended goals and serve as a base for modification.

- the plan & action library contains the abstracted behaviours that serve as commitments from **LPL** to **BBL** as well as a library of predefined single-agent plans that are designed for the agent's domain. The plans are indexed with their intended goals. All the entries use the object-oriented representation of Section 8.3. The execution method of primitive actions is supplied with an interface to produce commitments to the **BBL**.
- the active plan is the single plan whose execution is in progress at the moment. Its execution follows the procedure described in Section 8.3 and the primitives trigger commitments for the **BBL**.

As the description of loading dock agents in Section 10.3 shows, this approach works well as a straightforward basis for evaluation, but it also reveals some extensions that are necessary in the architecture as well as in the planning system:

- The 'frozen' knowledge base during planning prevents the planning process from being reactive. A dynamic environment or even behaviours that are initiated by the **BBL** may render the agent's situation different from the one represented in the planning process. Solution plans that refer to the situation that was valid when the service was called could be incorrect at execution time. A reactive planning process that is able to guide the planning process according to external signals (Section 12.1) should replace the rather isolated planning module.
- The false linearity assumption claims that conjunctive goals can be reached by sequencing their respective partial solutions. By not interrupting the activated plan and queueing goals, our local planning layer uses this assumption, however. A better scheme of interleaving execution and planning has to be found to get rid of this strong restriction (Section 12.1).
- Cooperative facilities are ignored in modelling only the **LPL**. Future investigations must show how the **CPL** could be constructed to include some temporal reasoning based on EVE. There will be also a need for having several, situation-independent planning processes at the **LPL** that produce multi-agent plans in order to develop and communicate these intentions.
- The Event Calculus and our current EVE system does not take probability, cost and utility measures into account. These are, however, important guidelines for rational agent design. Our goal is to incorporate these notions either into a nonlinear planning process or into the concept of a layer (Sections 12.2 and 12.1).

10.3 EVE in the Loading Dock Domain

As MAS are especially suited to managing complex, but structured systems, such as information networks and traffic, the *loading dock* scenario (Figure 52) provides an appropriate test bed for agent architectures like InteRRaP and multi-agent planning modules like EVE. It is basically an extension of the more abstract blocksworld task to a more practical problem and furthermore introduces issues such as large-scale distribution, robotics, and cooperation. The environment is is comprised of a closed region, the loading dock, that consists of discrete fields. Instead of blocks, the objects to rearrange are boxes characterised by certain colours. Also the shelves indicate

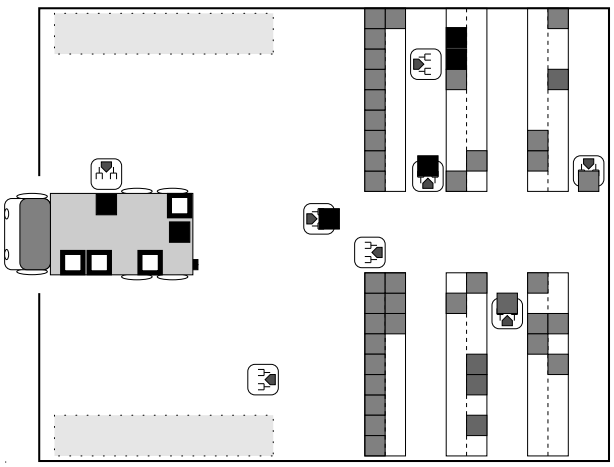


Figure 52: The Loading Dock Scenario

with their colour the type of box that is allowed to be stored here. The only other place, boxes can be placed on is the truck. The entities that are modelled via agents are forklifts. Special parking areas mark the usual positions of agents without tasks. Besides the restriction to the maximal load of one box and a limited perception range, the agents have the ability to execute the the following actions:

- Walk Ahead: If the field in front of the agent is free, i.e., no forklift, truck, shelf, or box is blocking the way, the agent can move from its actual position to the field in front without changing its orientation.
- Turn Left, Turn Right: An agent always has the ability to change its orientation in steps of 90 degrees. The position, however, remains the same.
- Grasp: If the agent is not carrying any box, it is able to move its grippers down to the truck or shelf field ahead, close them, and lift them up again. If there was a box in front, the agent will now be carrying it.
- Drop: This will cause the agent to move its grippers down to the empty truck or shelf field ahead and to open them. The grippers are moved up again afterwards. If the agent was carrying a box, it will now be stored on the field ahead.

Of course, this is not the kind of granularity that is inherent to the **LPL**. There are some routine **BBL** behaviours that provide better abstractions for acting:

- GotoRect: Is a behaviour that moves the agent from its current position to a place within the specified area.
- Turn: Repeated execution of 90-degree turns allows the definition of a behaviour that changes the orientation towards a defined direction.
- SearchBox: This routine will explore a shelf or the truck to identify a certain box. The agent will be placed in front of the box, if this pattern succeeds.
- SearchPlace: Complementary to SearchBox, this procedure looks for a free space on the truck or a shelf.

Tasks of the form *load the truck with a green box*, or *unload one blue box from the truck* can now be posed to the agent society and are negotiated between the agents. Eventually one agent takes temporary responsibility for fulfilling such an atomic task. Since the **BBL** is not able to solve it with its short-term perspective, the **LPL** is responsible for adopting it as a goal. From our axiomatisation (Section C) it can be seen that within this first evaluation we do not handle all the questions that this scenario poses the agent. However our idealised model reveals the following important considerations:

- Planning the above tasks within the loading dock domain works very well. The behaviours provide a robust execution base for the generated solutions.
- Execution failure due to the abstracted problem description in planning is handled adequately.
- The explicit reasoning at the **LPL** is far more flexible than the former method of predefining a complete plan library. The tasks of the forklifts could be easily enlarged by new ones without changing much within the domain axiomatisation.
- Since each action within the agent requires the same resources, the planning process turns out to be a purely linear one.
- Since path planning is a rather special issue, we chose not to use only the primitives within the planning layer. GotoArea is used to navigate to a certain position which is not always possible, but works in many situations.
- We did not model incomplete knowledge due to the limited perception. Section 12.2 shows some way of introducing assumptions to cope with this during planning time. The next point deals with gathering information at execution time.
- Sensing actions like SearchBox and SearchPlace cannot be handled by the current EVENT CALCULUS. Section 12.2 will present some techniques to cope with their semantics.
- Cooperation of agents is ignored for the moment. Future work will try to explore the usability of planning for this extended problem as well (Section 12.1). Nonlinear facilities will play an important role within.

10.4 Summary

With our integration of the planning system EVE into the InteRRaP agent architecture, we have succeeded in bringing the flexibility and deliberation of planning into the promising research of **DAI**. As our evaluation of the agents within the loading dock scenario shows, there are a lot of architectural and algorithmic questions still to be answered in the merging of explicit temporal reasoning with reactive facilities.

11 Related Work

Let us now compare the features of EVE with some well-known, related planning systems in common use. The main criteria for our comparison are efficiency, the application domain, expressivity and flexibility.

11.1 Planning with a logical framework

11.1.1 Situation Calculus-Based Systems

Situation Calculus based planning systems can be divided into logical and procedural approaches. However, it is theoretically possible to build the most powerful logical instance SIT using the clause formulation of Section 3.2 under SLDNF$^+$ (the axiomatisation does not need any abduction). In comparison to SIT, EVE is superior due to its better handling of the frame problem which consequently endows EVE with nonlinear and a far more flexible handling of plans.

Situation Calculus-based systems are restricted to the expressivity of SIT. STRIPS [37], for example, is unable to solve problems such as the register swapping problem, whereas the solutions of EVE's planning procedure are sound as well as complete (see Section 5). Sub optimality caused by the linearity assumption that is inherent to the Situation Calculus is not a problem for EVE.

11.1.2 Planning using Temporal Logics

There are attempts, such as the PHI planner in [26], that use modal logics with a temporal, interval-based interpretation for temporal reasoning purposes. The possible-worlds semantics (Kripke structures) delivers a notion of situation as well as situation transition due to action execution by the reachability-relation. In conjunction with a representational solution for the frame problem (local variables with changing values per-situation), modal logics allow one to realise the planning process through deduction alone (the PHI planner uses a modified S4 calculus).

Beneath the incorporation of tactical theorem proving that is also responsible for guiding the actual planning process, the plan operators in PHI also provide a more extensive framework than EVE. Conditionals and loops are standard tools with which to build up the plan structures.

Since plans correspond to paths within a Kripke structure, they are however condemned to be linear. As in the case of the Situation Calculus-based systems, this is a major drawback in solving multi-agent planning problems. Furthermore, the modification of existing plans to fit certain goals requires more effort than the a priori nonlinear approach of EVE. Interleaving of partial plans to the overall solution is only possible by a two-stage process: [15].

The thesis [34] describes an integration of the PHI planning system into the agent architecture InteRRaP. A comparison to our work that is presented in Section 10 reveals significant overlap (modelling of the loading dock, introduction of assumptions to deal with incomplete knowledge, error handling) as well as fundamental differences (deduction vs. abduction, reactivity of the planning process, resource management, etc.). The choice of EVE as the standard reasoning component for InteRRaP has thus been confirmed.

11.1.3 Event Calculus-Based Systems

A similar Event Calculus-based system to EVE is CHICA [23] served as a model for EVE. CHICA has some special concepts, such as *localised* planning, a way of dividing not mutually depending actions types and properties, and search space reduction strategies (maintenance — see Section 12.2). Localised planning has been

recognised as not being an appropriate technique in common domains. Whereas the adoption of search space reduction is planned for the future, EVE relies on its special improvements to the calculus to get a reasonable behaviour. [23] even mentions some not identifiable, non-termination situations that have occured in their implementation. We strongly suspect the undecidable, mutual destroyer situations that are described in Section 3.4 and handled by our versions of the EVENT CALCULUS to be responsible for this phenomenon. Furthermore the advantages of the OZ platform over the PROLOG base of CHICA include concurrency, object-oriented features and the efficient handling of constraints .

11.2 Procedural, Nonlinear Planning

Conventional, nonlinear planning approaches are most often based on procedural descriptions as those in [16]. Their computation tends to be divided into expansion (insertion of new actions to install the current goal) and conflict resolution (check for conflicts and linearisation to solve them) stages. EVE's behaviour, depending on the version of EVENT CALCULUS used, is very similar as we have outlined in Sections 5.5 and 5.6. Extended functionality, however, is gained by the logical approach of the EVENT CALCULUS and the flexible background of theorem proving.

There are however a lot of improvements to these partial-order approaches to achieve more practical planning: context-dependencies, treatment of sensing actions, probabilistic planning, etc. . Although EVE cannot provide such expressive operations, we aim to integrate these notions into our logical framework (Section 12.2).

11.3 Plan Graph Analysis

A new kind of semi-linear plan algorithms based on *plan graph analysis* is actually beating the records because of astonishing results in well known domains. GRAPHPLAN [2] for example is a forward planning approach that uses a special situation representation that even allows *semi-linearity*. As the actions are organised in several stages that are internally concurrently, but linear with respect to one another, this is a major restriction for multi-agent domains. Furthermore, depending on the domain, forward planning and situation representation lead to an exponential explosion of the search process — multi-agent environments, in particular, seem encounter this problem. Besides, the approach is not, as it stands, able to allow general plan modification, because of the incremental 'from scratch' behaviour that is necessary to build the special situation representation.

11.4 Multi-Agent Planning

Previous considerations related to multi-agent systems showed that decision making is a key functionality of the cognitive entities we would like to model. Much work has, of course, been put into the integration of planning into the agent-oriented programming framework.

One way to look at the problem is the bottom-up perspective of robotics. Since reactivity is the main criterion there, most approaches tend to adopt Operations-Research methods and are based on Input-Output feedback loops. More sophisticated research now tries to build hierarchies of abstractions on top of these rather simple, procedural behaviours. We argue that somewhere within this evolution of abstraction, the right place to integrate planning is found. The layered architecture of Section 10 reflects these considerations. Articles like [10] and [24] seem to share this opinion.

Another top-down perspective, however, likes to look at MAS as distributed problem solvers [42]. The top-level solution for some complex task has to be puzzled together by the individual agent deliberation. Here, the major question is the interaction between localised strategies with regard to the top-level goal. Enabling a cooperative layer within our architecture InteRRaP now merges these two perspectives into a unified framework.

Finally, there is some research from decision theory which continues to deliver meta-theories about agent societies as well as decision guidelines for rationality. A fact that is often left out within the other two approaches is the importance of probability for only partially informed systems. We therefore strongly recommend the incorporation of probability and utility into the agent architecture and/or the temporal reasoning processes.

A special usage of the EVENT CALCULUS with respect to robotics is reasoning about sensor data: [31]. The symmetrical treatment of past and future allows one to unify plan recognition/observation explanation and plan synthesis/intention formation. Although this provides some interesting background, a general abduction-based agent is far more expensive with respect to a deductive, but truth-maintaining one. A mixture of both inference mechanism would seem to offer a good compromise.

11.5 Summary

This section has shown that EVE derives much advantage from the EVENT CALCULUS on the one hand and its flexible architecture and representation on the other. We have demonstrated that our framework is extendible to general multi-agent domains and that it builds up a reasonable base for further extensions, which we hope will provide a unique combination of expressive planning, decision-theoretic reasoning and support for reactivity.

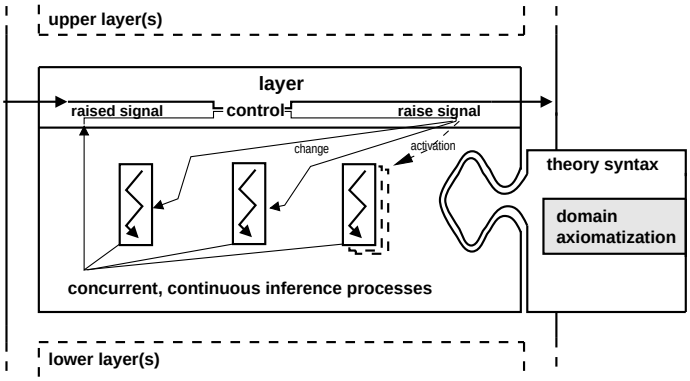


Figure 53: A Layer of an Agent Architecture

12 Future Research

The main reason to choose the nonlinear EVENT CALCULUS as the reasoning strategy for EVE is the intended application in multi-agent systems. Future research will mainly focus on extending the current framework to fit its behaviour into the requirements of this environment. We therefore first outline some considerations regarding the ongoing integration of EVE into InteRRaP and later explain, how the reasoning mechanism can be extended to capture many reasonable extensions from practical planning. Finally, we will discuss some attempts to find tailor-made heuristics in several planning domains and give an idea of how abduction and constraint logic programming can be unified in a much more elegant way to produce new translation functions from clause syntax into constraints.

12.1 Planning in a Multi-Agent Architecture

Section 10 has introduced the fundamental concepts of our underlying agent architecture and a prototypical integration of the EVE planning module. An important extension to the architecture that was implicitly employed there is the notion of concurrency between individual layers. Hence, the planning process itself is loosely linked to the **LPL** and its computation is done concurrently to the rest of the agent. Communication between the different functionalities can now be described as signal sending. We would like to adopt this view for the whole architecture by modelling all the computational facilities within an agent architecture as concurrent processes with signal sending and receiving functionality (Figure 53). This provides a very broad basis for reactive systems although the maintenance of consistency has to be inspected more closely.

Following this picture, the planning process has to develop its signal handling abilities. Possible signals include the creation of a new planning task, resource management information and further changes in the agent's environment. Normally, planning tasks are treated in isolation from a dynamically changing environment. The correctness of their results thus depends on the static information at the start of the computation. A far more reactive planning scheme should be able to guide its search with the supplementary information from the signals it receives. Although this could mean throwing away large parts of the already explored search tree due to some assumption that could not be hold, we argue that this will not happen in all domains and only in special situations which would render the result of conventional systems incorrect. The EVENT CALCULUS, in particular, should allow for mechanisms to support reactivity because the notion of abduction and events together

with constraint programming techniques delivers enough information to deal with incoming changes in the environment (see Section 12.4).

A question that we did not elaborate within our initial approach of bringing planning and agent-oriented programming together is the decision making on the cooperative layer. Of course, the use of plan structures could be adopted here, but they have to be carefully divided into protocols, non-deterministic social rules of communication, and strategies, decision functions that turn a protocol into a deterministic intention to execute. The primitive actions on this level include speech acts as well as synchronisation actions that wait for incoming communication. Both contain the object-level plan structures of the **LPL** enriched with multi-agent facilities as parameters. We propose that reasoning on the protocol/strategy level happens on the **CPL** and is guided by decision-theoretic considerations. The object-level plan structures are generated, modified, and analysed at the **LPL**, possibly with the help of the generic resources approach of Section 12.2. Therefore, the local planning processes are able to be configured with certain contextual information that allows them to switch between multi-agent and single-agent planning.

Execution of multi-agent plans requires the incorporation of translation functions that turn the multi-agent plan into a set of object-level single-agent plans including the necessary synchronisation and error-handling mechanisms. [19] introduces such a function for linear settings. We would like to extend these schemes to nonlinear plan structures with an extended error-handling mechanism.

The symmetrical treatment of past and future by the EVENT CALCULUS even provides a unified framework for realising *plan recognition* and plan generation. [31] presents an impressive example of how our approach could be also used to reason about incoming sensor data. It is a question of efficiency, whether the whole architecture should be based on a global, abductive procedure, or be built within a deductive framework with non-monotonic features to catch inconsistencies. We think that a combination of deduction on the architectural level and abduction within the planning processes is the most promising approach.

12.2 Extensions to the EVENT CALCULUS

12.2.1 Maintenance

Since the translation of the EVENT CALCULUS in EVE has turned out to spawn huge search spaces to handle interaction of events (see Section 9), there are several possibilities for guiding the control flow more accurately. Besides heuristics (Section 12.3), there is also a approach within the calculus that uses the notion of *maintenance* [23] instead of persistence. A problem with persistence is that it insists that property of interest hold throughout a certain time interval. Maintenance however allows the property to be terminated within the interval as long as it is re-established later. Although this would, at first sight, seem to increase the search space, investigation could nevertheless reveal potential for improving the overall depth-first behaviour.

12.2.2 Events with Duration

The current version of the EVENT CALCULUS regards events as ideally having no duration. Real parallel, or at least concurrent, settings, such as multi-agent domains, however, require the modelling of execution time intervals. The assumptions about the lifetime of conditions no longer hold. Preconditions have to last, in the worst case, until the end of event execution. Also the postconditions could be generated from the start on. Their existence, however, cannot be guaranteed until the end of execution. Furthermore, there is the possibility of properties coming into life

and dying again during execution (during conditions — they represent abstracted persistances). We would like to exetnd the EVENT CALCULUS to capture all these concepts. The worst case assumptions about conditions will have to covered as well as the introduction of during conditions and a distinction between *weak* and *strong* conditions will be needed so that the calculus will be able to reason about properties that can be interrupted and reinstalled without affecting the success of an action.

12.2.3 Context-Dependent Conditions

Another extension in expressivity is gained by the introduction of context-dependent conditions. As we have already mentioned in our discussion of the EVENT CALCULUS, postconditions that are strongly tied to certain preconditions could be modelled by several action axiomatisations. The increase in the branching factor can be avoided by using special mechanisms in the planning process. One of these resolves persistence conflicts by simply switching the unwanted effect with one of its associated preconditions off. On the other hand, proving that a certain property holds also requires the installation of its associated preconditions.

12.2.4 Probability in Planning

The action description need not be annotated with the condition dependencies alone. Planning in uncertain environments often requires that certain goals hold with a certain probability after execution of a plan. The notion of probability could therefore be introduced on the action level. One could replace persistence properties with the concept of evidence for certain preconditions. Bayes rule can be used to calculate the correct distributions for the time points. From these numbers, the resulting distributions after action execution can be generated. [21] describes the technical background of the BURIDAN algorithm, a partial-order planner that incorporates these presented techniques.

12.2.5 Sensing Actions

Furthermore, the representation of sensing actions imposes some new requirements on the temporal reasoning component. Sensing actions are special events that will gather some information *at execution time* besides installing their effects. They could serve to complete the knowledge that the planner needs to fulfil its task at planning time. The generation of the plan, however, has to be done for all possible outcomes of the environment sensing. Normally, this introduces several conditional branches within the plan structure. For all combinations of the outcomes of the sensing actions, the planner now has to establish the goal independently. Further resolution techniques for persistence destruction could now be introduced by making the contexts of destroyer and persistence inconsistent. How these mechanisms could be integrated into our logical framework should be a topic of our future research.

12.2.6 Hierarchical Planning

A general heuristic in problem solving is the reuse of already generated solutions. There are two ways of incorporating this heuristic into planning. First, one could have a certain plan as a proposal to the problem solver which then tries its best to modify it to meet the goal. This plan modification or *planning from second principles* has also been recognised as a NP-complete problem [3] and can often be more complex than the corresponding planning from scratch task (the EVENT CALCULUS and abduction show that plan synthesis and plan modification are in general of the

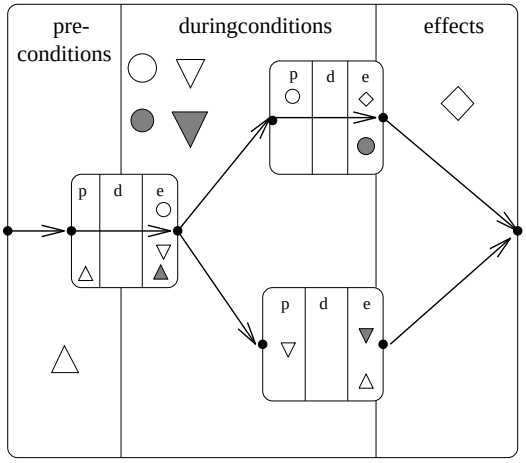


Figure 54: Plan Abstraction

same complexity). EVE supports modification, but an implementation of hierarchical planning by breaking up a plan into its primitive events (see NOAH [39; 40] and SIPE [43]) is therefore often too complex.

A second method, already employed in EVE's object system, is the abstraction of events and plans to be basically the same - whether defined as primitives or hierarchical events should be of no interest to the planner. Certainly one is losing a lot of information depending on the kind of abstraction (and risks conflicts during execution), but this significantly speeds the planning process up. A good rule is to calculate overall preconditions and postconditions for a plan. In the intended strong nonlinear setting, this is too weak, on its own, to handle the dependencies between parallel branches. which concludes in the additional introduction of during conditions. During conditions are furthermore the abstracted version of persistences that are not allowed to be destroyed in order to be successful.

The necessary computation of the overall conditions (Figure 54) is not difficult. The EVENT CALCULUS is again a perfect tool to deliver these. Preconditions are the collected preconditions of the children that have to be installed before the execution of the plan. A registration of successful knowledge base accesses produces these for free during either service involving the calculus. Overall effects consist of the properties provable by the calculus at `start(end)` without those properties that are introduced by the implicit start situation. This could be done by an analysis that is also suited for inspection of properties within the plan and thus exploring the during conditions.

This technique enables EVE to build plan libraries with reuse facilities at relatively little expense. Learning *AI* systems based on planning are also possible.

12.2.7 Generic Resources

As can be seen from the specification of the planning problem, the parts of the resource description relevant to the problem have to be complete at planning time. Decentralised multi-agent systems, however, are often faced with incomplete knowledge about the environment, especially other cooperative agents that could help to solve the goal. The abduction principle is again the key to realising far more flexible behaviour based on hypothetical reasoning. If the facts that are introduced by the start situation, i.e., knowledge base, can also be used as hypotheses, planning could involve a virtual resource amount reasonably treated by the minimality paradigm

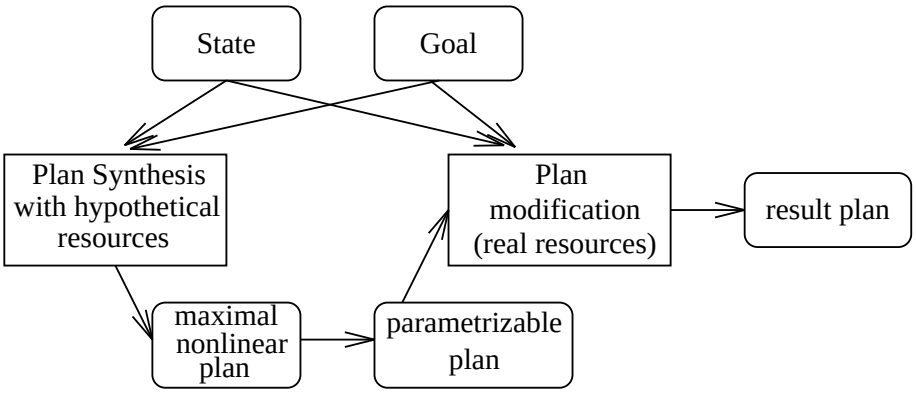


Figure 55: Generic resources

of abduction.

The execution of plans dealing with those *generic resources* [1], however, requires an additional revision with respect to the currently available resource rates. Again the abductive EVENT CALCULUS procedure provides the necessary functionality by its modification ability. First, virtual entities are replaced by roles, i.e., parametrisations to the plan. Afterwards the plan is thrown into modification together with the actual situation and an instantiation as well as the elimination of conflicts is obtained by the theory of time and action (Figure 55).

12.3 Heuristics

Probably best way of obtaining efficiency is of course to permit domain-dependent rules with EVE's general heuristics framework. Development of more generally usable techniques, such as way-planning could be of course part of those efforts mainly focussing on the intended application domain, the *loading dock* (see Section 10), and serve as guidelines for the construction of heuristics within the EVENT CALCULUS.

12.4 Abduction & constraint logic programming

So far, the translation functions that we have introduced in Section 6 use the constraint-based platform as a deduction system and introduce the hypothetical reasoning of abduction via *reification*, i.e. the interpretation of certain terms as sets and thus predicates. This can be seen from the list-based, explicit abduction set that is passed through the constraint abstractions. A closer investigation of the functionality of constraint logic programming approaches, however, reveals a deeper relation between CLP and abduction (Figure 56).

Basically, a constraint-based proof procedure knows some special subset of logic that it can handle efficiently. The data structures that are necessary to handle this reside in the *constraint store*. As soon as some element of this set is chosen (a basic constraint), it is checked whether the constraint store entails (implies) the basic constraint. If not, the constraint is added to the store and will trigger an update of the information within (propagation). If we compare this behaviour to abduction, this is basically the same: an abductive predicate (basic constraint) as a goal is checked to match the already existing assumptions (check on entailment). If not represented in the abductive state, it can serve as an additional hypothesis (added to the store) and thus maintenance (propagation) of the abduction set will produce

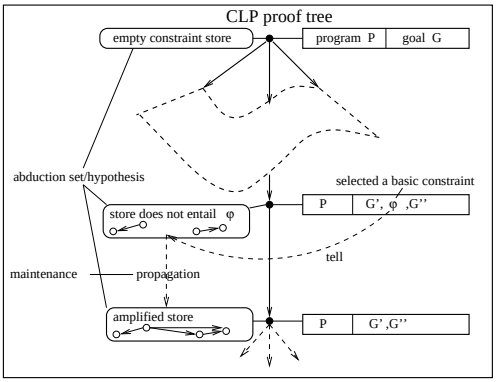


Figure 56: CLP is Abduction

some new hull (the new constraint store).

We therefore propose to get rid of the reification approach and use the constraint-based platform as an abductive inference procedure. Due to efficient constraint handling, new translation functions will produce more efficient code that is also more natural. The constraint metaphor can also be used as the basis for dynamically influencing the planning search process. Environmental change during planning could be expressed as additional constraints (perhaps through the introduction of some new event) that could be posted to the local computation spaces already spawned. Inconsistent assumptions within the abductive framework will cause the respective computation spaces to collapse. The remaining front represents all the paths that are consistent with the dynamic change and further expansion respects the new situation. The search has thus been dynamically guided.

12.5 Summary

With its intended integration into the agent model InteRRaP, there are several issues to explore within EVE apart from the design of the agent architecture. Investigating the suitability of the Event Calculus in multi-agent, distributed and real-time planning requires sophisticated techniques to be evaluated. Also extensions to the planning procedure, such as during conditions, improved time treatment, hierarchical features, and generic resources, turn out to belong to these methods. Finally, abduction and constraint logic programming seem to be related in a very close way. Improved clause-to-constraint translations with abductive facilities will be possible.

13 Conclusion

Comprising the results of our reported work with the analysis of our test applications (Sections 9 and 10), EVE has proved several facts related to planning, logic programming, and multi-agent systems:

- The EVENT CALCULUS is, together with an abduction principle, highly suited for nonlinear planning.
- The calculus can be improved to catch strong nonlinearity and to cope with worst case assumptions.
- The abduction principle is a rich extension to theorem proving that allows hypothetical reasoning for many purposes, such as planning.
- The EVENT CALCULUS behaves well under a theorem proof procedure, possibly given some restrictions.
- It is possible to implement full theorem proving facilities even with hypothetical and nonmonotonic reasoning on a constraint-based platform.
- Modern, procedural paradigms and logical programming do not contradict even in tightly logic-based approaches. New techniques, such as object-oriented abduction, are possible.
- OZ provides a perfect base for distributed, nonlinear planning attempts.
- Object-orientation is the key for reasonable data structures, also in planning.
- EVE can be extended to an efficient, distributed, multi-agent planning module. The incorporation into agent architectures arises interesting **DAI** research problems.
- Heuristics are necessary to guarantee efficiency as the calculus in its current formulations requires the construction of a rather large search space.

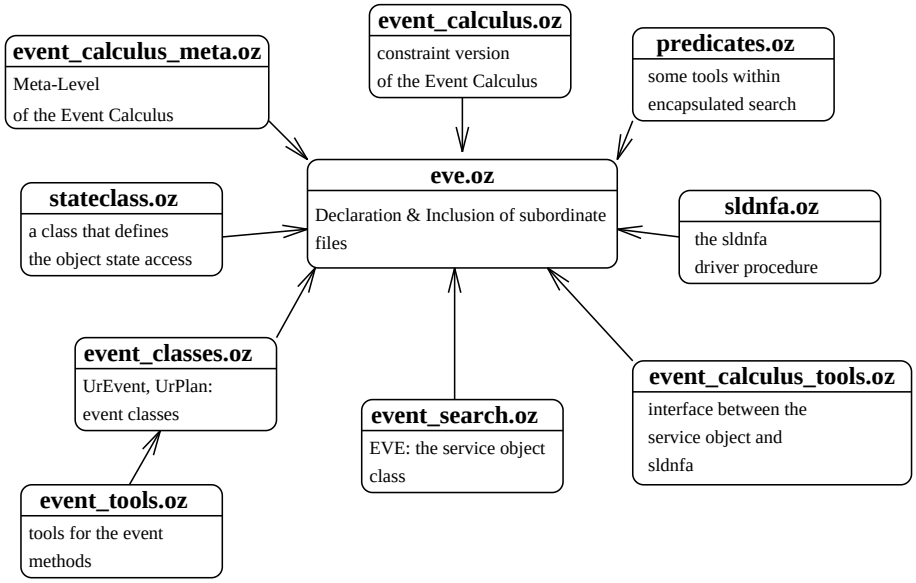


Figure 57: File Hierarchy

A EVE: Installation

After the theoretical and implementational issues, this section tries to give a short overview and guide to the installation and the customisation of the EVE module.

A.1 Dependencies and Compilation

The files that come with the EVE distribution, their content and mutual dependencies are shown in Figure 57. The only administrative task left to the user is an optional precompilation of `eve.oz` and its inclusion into his source code: `\feed ´eve.oz´`. From then on, the important constants `UrEvent,UrPlan,UrAction,EveCreate` and `Eve` are defined and allow interaction with the EVE system that is exemplary described in the next section A.2. A special significance is ascribed to `StateClass` that should serve as the parent for the user's knowledge base (see Sections 8.5 and 7.4) to provide the state accessing methods demanded for the interaction in encapsulated search.

A.2 A Counting Example

The following example allows to get insight into the basic steps of user interaction with EVE. Let us construct a counting domain in which the planning process is used to modify a counter. The beginning is made by inclusion of the EVE system:

```
\feed eve
```

The next step should be the customisation of the chosen knowledge base. EVE's test environment therefore defines a very simple, list-based approach. Its implementation could of course be far more sophisticated, but the functionality suffices to show the basic abilities. Remember that `StateClass` introduces the basic state access methods:

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

```
% KnowledgeBase
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% a simple list based knowleddge base
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
declare KnowledgeBase

class KnowledgeBase from UrObject StateClass
   attr
      facts:nil

   meth add(Fact)
      if {IsList Fact} then
         facts<-{Append
                 @facts
                 Fact}
      else
         facts<-{Append @facts [Fact]}
      fi
   end

   meth remove(Fact)
      if {Member Fact @facts} then
         facts<-{Filter @facts proc{$ X}
                                  if X=Fact then true
                                  else false fi
                               end}
      fi
   end

   meth getfactdisj(Fact)
      {MemberDisj Fact @facts}
   end

   meth reset()
      facts<-nil
   end

end
```

`KnowledgeBase` stores its content in a simple list. There can be additions (`add()`), deletions (`delete()`) and a total `reset()` of the knowledge base. `getfactdisj()` is responsible for interfacing the KB within the Event Calculus — it matches the members of the list disjunctively against the requested property. Now let us introduce a KB object and its representation of the actual counter's state that is set to 1:

```
% new knowledgebase
declare ActualKnowledgeBase
create ActualKnowledgeBase from KnowledgeBase
end

{ActualKnowledgeBase [reset() add([actual(1)])]}
```

Still lacking is the definition of a counting action. Its roles are the counter state before (`fromn`) and after (`ton`) the counting operation and the direction in which it counts (`direction`).:

```
declare Count

{EveCreate UrAction Count}
{Count addRoles(roles(fromn:_ ton:_ direction:_))}
{Count setId('Count')}
{Count addExecutionParameters([count evevar(fromn:_) eve-
var(ton:_)])}
```

The conditions for `Count` are straightforward. Before its execution, the actual counter must contain the `fromn` value. Afterwards, its content will be `ton` and no longer `fromn`:

```
{Count addConditions(posprecondition [actual(evevar(fromn:_))])}
{Count addConditions(pospostcondition [actual(evevar(ton:_))])}
{Count addConditions(negpostcondition [actual(evevar(fromn:_))])}
```

The relation between counting direction and counter's states is given by the following constraint that will furthermore restrict the values to be in range from $1 \ldots 5$ and the directions `up` and `down`:

```
{Count addConstraints(proc{$ Roles}
                         Roles.fromn::1#5
                         Roles.ton::1#5

                         or
                            Roles.direction=up
                            {FD.'+' Roles.fromn 1 Roles.ton}
                         []
                            Roles.direction=down
                            {FD.'-' Roles.fromn 1 Roles.ton}
                         ro

                      end|nil)}
```

Note that the disjunction is really disjoint. Whether the action is built on the top-level or in encapsulated search, the behaviour in instantiations is reasonable. In encapsulated search, `Count` even turns out to implicitly represent two types `CountUp` and `CountDown` at once due to the disjunction semantics. Left to trigger the planning process remain instantiation of planning service object, its connection to the KB, registration of `Count` as a type of the planning domain and the synthesis request to obtain a solution plan that counts from 1 to 3:

```
declare Planner  TestPlan  Busy
% new planner object
create Planner from Eve with init end

{Planner setKnowledgeBase(ActualKnowledgeBase KnowledgeBase)}

{Planner setAllowedActionTypes([Count])}

{Planner synthesis([actual(3)] Busy TestPlan)}
```

A.3 Summary

Installation and customisation of EVE is very simple. The given example shows the basic steps to perform in order to connect the planning module to the application

domain. To get a complete overview about programming with EVE, we highly recommend [5].

B Axiomatisation of the Multi-Agent Blockworld

The following source code represents the multi-agent blocksworld domain and exemplary defines a situation description using the list-based knowledge base object known from the previous installation introduction from Section A:

```
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   % Action classes of the the blocksworld scenario
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

   % the blocksworld generic actions
   declare  Pickup Stack Unstack Drop

   {EveCreate UrAction Pickup}
   {Pickup setId('Pickup')}
   {Pickup addExecutionParameters([pickup evevar(agent:_) eve-
var(block:_)])}
   {Pickup addRoles(roles(agent:_ block:_))}
   {Pickup addConditions(pospostconditions
                         [holding(evevar(agent:_) eve-
var(block:_))])}
   {Pickup addConditions(negpostconditions
                         [handempty(evevar(agent:_))
                          clear(evevar(block:_))
                          ontable(evevar(block:_))])}
   {Pickup addConditions(pospreconditions
                         [handempty(evevar(agent:_))
                          clear(evevar(block:_))
                          ontable(evevar(block:_))])}


   {EveCreate UrAction Drop}
   {Drop setId('Drop')}
   {Drop addExecutionParameters([drop evevar(agent:_) eve-
var(block:_)])}
   {Drop addRoles(roles(agent:_ block:_))}
   {Drop addConditions(pospostconditions
                       [ontable(evevar(block:_))
                        clear(evevar(block:_))
                        handempty(evevar(agent:_))])}
   {Drop addConditions(negpostconditions
                       [holding(evevar(agent:_) evevar(block:_))])}
   {Drop addConditions(pospreconditions
                       [holding(evevar(agent:_) evevar(block:_))])}


   {EveCreate UrAction Stack}
   {Stack setId('Stack')}
   {Stack addExecutionParameters([stack evevar(agent:_)
                                  evevar(block1:_) eve-
var(block2:_)])}
   {Stack addRoles(roles(agent:_ block1:_ block2:_))}
   {Stack addConditions(pospostconditions
                        [clear(evevar(block1:_))
                         on(evevar(block1:_) evevar(block2:_))
                         handempty(evevar(agent:_))])}
```

```
   {Stack addConditions(negpostconditions
                        [clear(evevar(block2:_))
                         holding(evevar(agent:_) eve-
var(block1:_))])}
   {Stack addConditions(pospreconditions
                        [clear(evevar(block2:_))
                         holding(evevar(agent:_) eve-
var(block1:_))])}

   {EveCreate UrAction Unstack}
   {Unstack setId('Unstack')}
   {Unstack addExecutionParameters([unstack
                                    evevar(agent:_)
                                    evevar(block1:_)
                                    evevar(block2:_)])}

   {Unstack addRoles(roles(agent:_ block1:_ block2:_))}
   {Unstack addConditions(pospostconditions
                          [holding(evevar(agent:_) evevar(block1:_))
                           clear(evevar(block2:_))])}
   {Unstack addConditions(negpostconditions
                          [handempty(evevar(agent:_))
                           clear(evevar(block1:_))
                           on(evevar(block1:_) evevar(block2:_))])}
   {Unstack addConditions(pospreconditions
                          [handempty(evevar(agent:_))
                           clear(evevar(block1:_))
                           on(evevar(block1:_) evevar(block2:_))])}

   declare Planner ActualKnowledgeBase TestPlan

   % new planner object
   create Planner from Eve with init
   end

   % new knowledgebase
   create ActualKnowledgeBase from KnowledgeBase
   end

   {Planner setKnowledgeBase(ActualKnowledgeBase KnowledgeBase)}

   % The KnowledgeBase delivers the start situation!
   {ActualKnowledgeBase [reset() add([on(blockb blocka)
                                      ontable(blocka)
                                      ontable(blockc)
                                      clear(blockc)
                                      clear(blockb)
                                      handempty(robo1)
                                      handempty(robo2)]) ]}
```

C Axiomatisation of the Loading Dock Scenario

This section presents the current axiomatisation of the loading dock domain. `PbcGeneric` builds the parent type of all available actions. It incorporates an execution procedure that will transmit commitments to the **BBL**. Inherited from this type are the agent-independent action types that also could serve as a base axiomatisation for multi-agent planning. Finally, the agent-specific actions are defined as their subtypes.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Action classes of the loading dock scenario
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

local

   %%% The finite domains
   PbcGeneric
   Move
   Turn
   Grasp
   Drop

in

   %%% The generic pbc-action
   {EveCreate UrAction PbcGeneric}
   {PbcGeneric setExecution(proc{$ Parameters Success}
                               local Self ActivationParameter in
                                  Parameters=Self|ActivationParameter|nil
                                  {PBC_Control trigger(Self
                                                       ActivationPa-
rameter
                                                       Success)}
                               end
                            end)}


   %%% general Move command
   {EveCreate PbcGeneric Move}
   {Move setId('Move')}
   {Move addRoles(roles(agent:_ x1:_ y1:_ x2:_ y2:_ direction:_))}
   {Move addExecutionParameters([activate(beh:'Goto_rect'
                                          args:rect(evevar(x2:_)
                                                    evevar(y2:_)
                                                    evevar(x2:_)
                                                    evevar(y2:_)))
                                ])}
   {Move addConditions(prespreconditions
                       [ agent(agentname:evevar(agent:_)
                               x:evevar(x1:_)
                               y:evevar(y1:_))
                         agentdirection(agentname:evevar(agent:_)
                                        direc-
tion:evevar(direction:_))
                         empty(x:evevar(x2:_)
                               y:evevar(y2:_))
```

```
                      ])}
   {Move addConditions(pospostconditions
                      [ agent(agentname:evevar(agent:_)
                              x:evevar(x2:_)
                              y:evevar(y2:_))
                        agentdirection(agentname:evevar(agent:_)
                                       direction:unknown)
                        empty(x:evevar(x1:_)
                              y:evevar(y1:_))
                      ])}
   {Move addConditions(negpostconditions
                      [ agent(agentname:evevar(agent:_)
                              x:evevar(x1:_)
                              y:evevar(y1:_))
                        empty(x:evevar(x2:_)
                              y:evevar(y2:_))
                        agentdirection(agentname:evevar(agent:_)
                                       direction:evevar(direction:_))]
                     )}

   %%% specialised for our agent
   {EveCreate Move LocalMove}
   {LocalMove setId(´LocalMove´)}
   {LocalMove getRole(agent {KB get(my_name $)})}


   %%% the Turn command
   {EveCreate PbcGeneric Turn}
   {Turn setId(´Turn´)}
   {Turn addRoles(roles(agent:_ direction1:_
                        direction2:_))}
   {Turn addExecutionParameters([activate(beh:´Turn´
                                          args:evevar(direction2:_))
                                ])}
   {Turn addConditions(pospreconditions
                      [
                        agentdirection(agentname:evevar(agent:_)
                                       direc-
tion:evevar(direction1:_))])}
   {Turn addConditions(pospostconditions
                      [agentdirection(agentname:evevar(agent:_)
                                      direc-
tion:evevar(direction2:_))])}
   {Turn addConditions(negpostconditions
                      [
                        agentdirection(agentname:evevar(agent:_)
                                       direc-
tion:evevar(direction1:_))])}

   %%% specialised for our agent
   {EveCreate Turn LocalTurn}
   {LocalTurn setId(´LocalTurn´)}
   {LocalTurn getRole(agent {KB get(my_name $)})}
```

```
   %%%  the Grasp command
   {EveCreate PbcGeneric Grasp}
   {Grasp setId('Grasp')}
   {Grasp addRoles(roles(agent:_ agx:_ agy:_ direction:_
                         box:_ boxx:_ boxy:_ type:_))}

   {Grasp addExecutionParameters([activate(beh:'Get_box'
                                           args:nil)
                                  ])}

   {Grasp addConditions(pospreconditions
                        [
                          agent(agentname:evevar(agent:_)
                                x:evevar(agx:_)
                                y:evevar(agy:_))
                          handempty(agentname:evevar(agent:_))
                          agentdirection(agentname:evevar(agent:_)
                                         direc-
tion:evevar(direction:_))
                          box(boxname:evevar(box:_)
                              boxtype:evevar(type:_)
                              x:evevar(boxx:_)
                              y:evevar(boxy:_))
                        ])}
   {Grasp addConditions(pospostconditions
                        [
                          holding(agentname:evevar(agent:_)
                                  boxname:evevar(box:_)
                                  boxtype:evevar(type:_))
                          free(x:evevar(boxx:_)
                               y:evevar(boxy:_))

                        ])}
   {Grasp addConditions(negpostconditions
                        [
                          handempty(agentname:evevar(agent:_))
                          box(boxname:evevar(box:_)
                              boxtype:evevar(type:_)
                              x:evevar(boxx:_)
                              y:evevar(boxy:_))
                       ])}
   {Grasp addConstraints([proc{$ Roles}
                            or
                               Roles.direction=north
                               Roles.agx=Roles.boxx
                               {FD.'-' Roles.boxy 1 Roles.agy}
                            []
                               Roles.direction=south
                               Roles.agx=Roles.boxx
                               {FD.'+' Roles.boxy 1 Roles.agy}
                            []
                               Roles.direction=east
                               Roles.agy=Roles.boxy
                               {FD.'-' Roles.boxx 1 Roles.agx}
```

```
                              []
                                 Roles.direction=west
                                 Roles.agy=Roles.boxy
                                 {FD.´+´ Roles.boxx 1 Roles.agx}
                              ro
                           end])}

   %%% specialized for our agent
   {EveCreate Grasp LocalGrasp}
   {LocalGrasp setId(´LocalGrasp´)}
   {LocalGrasp getRole(agent {KB get(my_name $)})}



   %%% the Drop command
   {EveCreate PbcGeneric Drop}
   {Drop setId(´Drop´)}
   {Drop addRoles(roles(agent:_ agx:_ agy:_ direction:_
                        box:_ boxx:_ boxy:_ type:_))}

   {Drop addExecutionParameters([activate(beh:´Put_box´
                                          args:nil)
                                 ])}

   {Drop addConditions(pospreconditions
                       [
                         agent(agentname:evevar(agent:_) x:evevar(agx:_) y:evevar(agy:_))
                         agentdirection(agentname:evevar(agent:_) di-
rection:evevar(direction:_))
                         holding(agentname:evevar(agent:_) boxname:evevar(box:_) box-
type:evevar(type:_))
                         free(x:evevar(boxx:_) y:evevar(boxy:_))
                       ])}
   {Drop addConditions(pospostconditions
                       [
                         box(boxname:evevar(box:_) boxtype:evevar(type:_) x:evevar(boxx:_)
                             y:evevar(boxy:_))
                         ])}
   {Drop addConditions(negpostconditions
                       [ holding(agentname:evevar(agent:_) boxname:evevar(box:_) box-
type:evevar(type:_))
                         free(x:evevar(boxx:_) y:evevar(boxy:_))
                       ])}
   {Drop addConstraints([proc{$ Roles}
                            or
                               Roles.direction=north
                               Roles.agx=Roles.boxx
                               {FD.´-´ Roles.boxy 1 Roles.agy}
                            []
                               Roles.direction=south
                               Roles.agx=Roles.boxx
                               {FD.´+´ Roles.boxy 1 Roles.agy}
                            []
                               Roles.direction=east
                               Roles.agy=Roles.boxy
```

```
                        {FD.´-´ Roles.boxx 1 Roles.agx}
                     []
                        Roles.direction=west
                        Roles.agy=Roles.boxy
                        {FD.´+´ Roles.boxx 1 Roles.agx}
                     ro
                  end])}

%%% specialized for our agent
{EveCreate Drop LocalDrop}
{LocalDrop setId(´LocalDrop´)}
```

List of Figures

1 Normalised Clauses C,$\tilde{C}$ . . . 12
2 The Task of Plan Generation . . . 14
3 Situation, Action Type, Goal, and Plan . . . 15
4 Nonlinear Multi-Agent Plan . . . 15
5 Hierarchical Plan . . . 16
6 The Task of Plan Analysis . . . 16
7 Computational Architecture of EVE . . . 17
8 Action Type Definition/Planning Domain Axiomatisation . . . 18
9 The Situation Calculus . . . 20
10 Propagation of Properties in the Situation Calculus . . . 20
11 The Event Calculus by Shanahan . . . 21
12 Propagation of Properties in the Event Calculus . . . 21
13 Strong Nonlinear Incorrectness . . . 22
14 Event Calculus by Missiaen . . . 22
15 Nontermination . . . 23
16 A Not Strong Nonlinear Solution . . . 23
17 The Simple Event Calculus of EVE . . . 24
18 The Extended Event Calculus of EVE . . . 26
19 The SLD Step . . . 31
20 The SLDNF Step . . . 32
21 In-soundness of Non-ground NF . . . 32
22 Incompleteness of Ground NF . . . 33
23 The SLDCNF Step . . . 34
24 The De Morgan's Laws . . . 34
25 The SLDNF$^+$ Step . . . 36
26 Example Proof with SLDNF$^+$ . . . 37
27 The SLDA Step . . . 38
28 The SLDNFA Step . . . 40
29 The SLDCNFA Step . . . 41
30 The SLDNFA$^+$ Step . . . 42
31 Translation Scheme T_1 – Resolution . . . 54
32 T_2 – Negation As Failure . . . 55
33 T_3 – Constructive Negation . . . 57
34 T_3^+ – Constructive Negation . . . 57
35 T_3^- – Constructive Negation . . . 58
36 The `MetaAbduction` Predicate . . . 59
37 T_4 - Abduction . . . 60
38 T_4^+ – Abduction . . . 61
39 T_4^- – Abduction . . . 62
40 Driver for Scheme T_4 . . . 63
41 Object-Oriented Abduction . . . 66
42 the Plan Service Class `Eve` . . . 69
43 The Event Class `UrEvent` . . . 70
44 A `display()` Result . . . 71
45 An Example Event Hierarchy . . . 72
46 The Default Abductive State . . . 72
47 An EVE-Compatible Knowledge Base . . . 73
48 the Multi-Agent Blocksworld . . . 74
49 The InteRRaP Architecture . . . 77
50 A Layer within InteRRaP . . . 77
51 EVE within the **LPL** . . . 78
52 The Loading Dock Scenario . . . 80

53 A Layer of an Agent Architecture 85
54 Plan Abstraction 88
55 Generic resources 89
56 CLP is Abduction 90
57 File Hierarchy 93

References

[1] M. Alicia Pérez. Multiagent planning in Prodigy. May 1991.

[2] Avrim L. Blum and Merrik L. Furst. Fast Planning Through Planning Graph Analysis.

[3] Bernhard Nebel and Jana Koehler. Plan Modification versus Plan Generation: A Complexity-Theoretic Perspective. pages 1436 – 1441.

[4] Christian Schulte, Gert Smolka, and Jörg Würtz. Encapsulated Search and Constraint Programming in Oz. March 1994.

[5] Christoph G. Jung. *the* EVE *User Guide*, February 1996.

[6] Christoph G. Jung, Klaus Fischer, and Alastair Burt. Resolution, Constructive Negation, and Abduction over Finite Domains in Higher Order Constraint Programming. In Andreas Abecker, Harald Meyer auf'm Hofe, Jörg Müller, and Jörg Würtz, editors, *Proceedings of the* DFKI *Workshop on Constraint Based Problem Solving*, DFKI Document, Saarbrücken, Germany, February 1996. DFKI GmbH.

[7] David Chang. Constructive Negation based on the Complete Database. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proceedings of the 5th Int. Conf. on Logic Programming, Seattle, 1988*, pages 111–125. The MIT Press, September 1988.

[8] K. Eshghi and R. A. Kowalski. Abduction compared with negation as failure. In *Proceedings of the 6st Int. Conf. on Logic Programming.* The MIT Press, 1989.

[9] C. Evans. Negation as failure as an approach to the hanks and mcdermott problem. 1989.

[10] R. James Firby. Building symbolic primitives with continuous control routines. 1994.

[11] Gert Smolka. *An Oz Primer.* DFKI Oz Documentation. Saarbrücken, Germany, April 1995.

[12] Gert Smolka. The Oz Programming Model. In Jan van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.

[13] Gert Smolka and Christian Schulte. Logische Programmierung. 1993. Skriptum zur gleichnamigen Vorlesung.

[14] S. Hanks and D. McDermott. Default reasoning, nonmonotonic logic, and the frame problem. pages 328 – 333, 1986.

[15] Jana Koehler. Towards a logical treatment of plan reuse. pages 285–286, 1992.

[16] Joachim Hertzberg. *Planen. Einfuehrung in die Planerstellungsmethoden der Kuenstlichen Intelligenz*, volume 65 of *Reihe Informatik.* Wissenschaftsverlag, Mannheim - Wien - Zuehrich, 1989.

[17] John McCarthy. Situations, Actions and Causal Laws. 1957.

[18] John W. Lloyd. *Foundations of Logic Programming. 2nd ext. Edition.* Symbolic Computation. Springer, Berlin - Heidelberg - New York, 1987.

[19] Jörg P. Müller. *An Architecture for Dynamically Interacting Agents.* Dissertation, Universität des Saarlandes, 1996.

[20] Keith L. Clark. Negation as Failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322, New York, 1978. Plenum press.

[21] N. Kushmerik, S. Hanks, and D. Weld. An algorithm for probabilistic planning. 1994.

[22] J. Lloyd and R. Topor. Making PROLOG More Expressive. *Journal of Logic Programming*, 1(3):225 – 240, 1984.

[23] Lode Missiaen. *Localized Abductive Planning with the Event Calculus.* PhD Dissertation, K.U. Leuven, Leuven, September 1991.

[24] D.M. Lyons and A.J.Hendricks. A practical approach to integrating reaction and deliberation. 1996.

[25] Marc Denecker, Lode Missiaen, and Maurice Bruynooghe. Temporal reasoning with Abductive Event Calculus. January 1992.

[26] Mathias Bauer, Susanne Biundo, Dietmar Dengler, Jana Koehler, and Gabriele Paul. PHI - A Logic-Based Tool for Intelligent Help Systems. Technical report, Saarbr"ucken, 1992.

[27] Melvin Fitting. *First Order Logic and Automated Theorem Proving.* Texts and Monographs in Computer Science. Springer Verlag, New York, September 1990.

[28] Michael Rosinus. ALADIN - A Language for Designing InteRRaP Agents. Diplomarbeit, Universität des Saarlandes, 1996.

[29] Jörg P. Müller and Markus Pischel. The Agent Architecture InteRRaP: Concept and Application. Technical report, Saarbr"ucken, 1993.

[30] Murray P. Shanahan. Representing continuous change in the event calculus. In *Proceedings of the ECAI 90*, pages 589–603, August 1990.

[31] Murray Shanahan. Robotics and the Common Sense Informatic Situation. Budapest, Hungary. to appear.

[32] Nils J. Nilsson. *Principles of Artificial Intelligence.* Tiago Publishing Company, Palo Alto, CA, 1980.

[33] Peter J. Stuckey. Constructive Negation for Constraint Logic Programming. In *Proceedings LICS*, 1991.

[34] Ralf Wingenter. Ein Deduktiver Planungsansatz für das Verladehofszenario. Diplomarbeit, Universität des Saarlandes, 1996.

[35] Robert A. Kowalski. *Logic for Problem Solving*, volume 7 of *Artificial Intelligence Series.* Elsevier Science Publisher B.V. (North-Holland), 1979.

[36] Robert A. Kowalski and Marek Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1):67–95, 1986.

[37] Robert E. Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artifical Intelligence*, 2(3/4):189–208, 1971.

[38] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[39] E.D. Sacerdoti. Planning in a Hierarchy of Abstraction Spaces. In *Proceedings of the IJCAI 1975*, 1975.

[40] E.D. Sacerdoti. *A Structure for Plans and Behaviour*. Elsevier, North Holland, 1977.

[41] M. Shanahan. Prediction is deduction but explanation is abduction. In *Proceedings of the IJCAI 89*, page 1055, 1989.

[42] F. von Martial. Interactions among autonomous planning systems. pages 105–120, 1990.

[43] D.E. Wilkins. Hierarchical Planning: Definition and Implementation. In *Proceedings of the ECAI 1986*, 1986.

[44] Yoav Shoham. What's the frame problem ? In Michael P. Georgeff and Amy L. Lansky, editors, *Reasoning about Actions and Plans*, pages 83–98. Morgan Kaufmann Publishers, Inc., Los Altos, CA, June 1986.

Index

C, 30, 53
C,$\tilde{C}$, 12
CET, 30
$\bot$, 11, 30
$\bot^*$, 18, 25
$\equiv$, 11
$\exists$, 11
$\forall$, 11
$\models|$, 11
$\neg$, 11
$\supset$, 11
$\tilde{C}$, 30
$\tilde{\exists}$, 11
$\tilde{\forall}$, 11
$\top$, 11, 30
$\top^*$, 18, 25
$\vdash$, 11
$\vee$, 11
$\wedge$, 11
select, 29
variant, 29
BBL, 76
CPL, 76
KB, 77
LPL, 76
PS, 77
SG, 77
WIF, 77
choose, 29
disunify, 33
fail, 30
maintain, 39
reprove, 39
unify, 29
generic resources, 89
`addConditions()`, 70
`display()`, 70
`getState()`, 71
`initiates()`, 18
`postcondition()`, 48
`precondition()`, 18
`setExpense()`, 69
`setHeuristic()`, 69
`FiDo`, 56
`act()`, 45
`continue()`, 68
`fails()`, 19
`getState()`, 73
`halt()`, 68
`happens()`, 46
`holds()`, 25, 47
`initial`*, 22
`initiates()`, 45
`negprecondition()`, 45
`on()`, 74
`posprecondition()`, 45
`start()`, 20, 22
`terminates()`, 45
$\neg^*$`()`, 18, 25
`=`, 12, 29
`Drop`, 74
`Eve`, 68
`KnowledgeBase`, 94
`Maintain`, 59, 66
`MemberDisj`, 59, 60
`MetaAbduction`, 56, 60
`MetaClause`, 65
`Pickup`, 74
`SolveCombinator`, 56
`Solver`, 69
`Solve`, 53
`Stack`, 74
`StateClass`, 93
`Unstack`, 74
`UrAction`, 71
`UrEvent`, 69, 72
`UrPlan`, 71
`act()`, 19, 46, 47
`addConstraints()`, 70
`addEvent()`, 70
`addRole()`, 70
`analysis()`, 68
`before()`, 20, 45, 46
`clear()`, 74
`clipped()`, 20, 25, 47, 48, 50
`end`*, 47
`end()`, 20, 22, 47
`end`, 47, 67
`evaluation()`, 68
`evevar()`, 70
`execute()`, 70
`executestop()`, 71
`fails()`, 23, 25, 47, 48, 50
`getAllAfter()`, 70
`getAllEvents()`, 70
`getConditions()`, 70
`getEvent()`, 70
`getRole()`, 70
`getState()`, 67
`getfactdisj()`, 67, 72
`handempty()`, 74
`happens()`, 20, 45, 48

`happens()` , 47
`holding()`, 74
`holds()`, 19, 47, 49, 50
`holdsfalse()`, 24, 67
`holdstrue()`, 24
`in()`, 21, 22
`initial`*, 46, 67
`initial`, 22, 47, 67
`initiates ()`, 67
`initiates()`, 72
`member()`, 18
`modification()`, 68
`negpreconditions()`, 24
`next()`, 68
`nil`, 18, 25
`ontable()`, 74
`out()`, 22, 47, 49, 50
`pospreconditions()`, 24
`precondition()`, 47
`result()`, 19
`setActionBounds()`, 68
`setAllowedActionTypes()`, 68
`setBefore()`, 70
`setDebugLevel()`, 69
`setEvent()`, 70
`setKnowledgeBase()`, 68
`setRoleMap()`, 70
`setState()`, 67, 71, 73
`start()`, 47
`stop()`, 68
`synthesis()`, 68
`terminates()`, 18, 47, 67
`false`, 11
`true`, 11
`if else fi`, 52
`local in end`, 52
`or [] ro`, 52
`proc end`, 52
NP-hard, 13
SIPE, 88
SLDNFA$^+$, 46
CHICA, 82
DFKI Oz, 52
EVE, 9
Event Calculus, 45
Event Calculus, 9, 16, 24, 25, 33, 82
GRAPHPLAN, 83
InteRRaP, 53
NOAH, 88
NP-completeness, 87
NP-hard, 43
Oz, 9, 13, 16, 52, 83
PROLOG, 83
SIT, 82
SLDCNFA, 39
SLDCNF, 33
SLDNF$^+$, 34
SLDNFA$^+$, 9, 12
SLDNFA, 39
SLDNFA$^+$, 41, 56
SLDNF, 12, 30, 55
SLDNF$^+$, 45, 56
SLD, 12, 29, 53
STRIPS, 20, 82
Situation Calculus, 9, 19, 21, 82

abducible, 35
abducible maintenance, 39
abducible maintenance, 45
abduction, 9, 35, 45
abduction set, 35, 45, 47
abduction state, 35, 66
abductive predicate, 35, 45, 46
action, 9, 13, 18, 45, 69
action type, 18
action type, 13
action type, 45, 47, 69
action/plan execution, 45
action/plan execution, 14, 24, 70, 86
agent, 10, 53, 76
ALADIN, 77
anti-symmetry, 46
application, 53
architecture, 9, 16
Artificial Intelligence, 13
ask operator, 52
assumption, 35, 46
atom, 11
autonomy, 76
axiomatisation, 9, 18, 45

backtracking, 34
Backus-Naur Form, 11
backward planning, 15, 20
body, 12
boolean value, 18

calculus, 9, 11
cell, 52
choice, 13, 29, 33, 39, 45, 59
choice point, 9
choice rule, 9, 29, 45, 47
Clark's Completion form, 11
Clark's equality theory, 30
class, 68, 69
classical logic, 11
clause, 9, 11, 13, 18, 29, 30, 53

clause program, 9, 12, 18, 29
closed formula, 11
communication, 76
completeness, 9, 11, 14, 16, 18, 39, 45
complexity theory, 13
computation space, 9, 52
concurrency, 9, 15, 17, 19, 23, 52
condition, 13, 70
conditional, 52
conjunction, 11, 52
consistency, 11, 18, 33, 39, 46, 56
constraint, 9, 13, 17, 52, 83
constraint logic system, 13
constraint abstraction, 9, 52, 65
constructive negation, 33
constructive negation, 9, 41, 56, 67
context-dependent action/event type, 25
contour, 35
cooperation, 76
cooperative domain, 25, 88
correctness, 11, 21, 30
cost, 13, 48

DAI, 76
De Morgan's laws, 56
De Morgan's laws, 34
decidability, 11
deduction, 35
deliberation, 76
denotational semantics, 13
depth bound, 48, 68
depth-first search, 47, 53, 86
destroyer, 21, 22, 24
disjunction, 11, 29, 52
distributed planning, 10
disunification, 33
domain, 9, 11, 13, 18, 45
DPS, 76
during condition, 25, 87, 88

effect, 14, 18, 45
encapsulated search, 9, 17, 52, 66
encapsulated search driver, 9
entity, 11
equality, 12, 29
equality maintenance, 33
equivalence, 11
equivalence semantics, 11
equivalence transformation, 12
event, 13, 20, 25, 69
event duration, 20, 25, 86
event type, 13
execution, 18
existential quantification, 11, 29, 34, 35
explicit representation, 18
explicit representation, 20
explicit representation, 19
exponential complexity, 13, 35, 83
extended least commitment, 48

fact, 14
fact clause, 12, 35
failure, 24, 25
fairness, 13, 30, 45, 47, 53
finite domain, 34
finite domain, 9, 35, 41, 45, 47, 56
first-order logic, 11, 30
formula, 11
forward planning, 15, 83
four-valued logic, 11
free variable, 11
fuzzy logic, 11

general problem solving, 13
generic resources, 10
goal, 12, 13, 18, 29, 76
goal set, 29
GPS, 13
graph theory, 13
groundness, 30, 33

head, 12
heuristics, 13, 25, 43, 47, 60, 65, 69, 89
hierarchical planning, 10, 15
higher-order logic, 9, 17
hypothesis, 9, 35
hypothetical reasoning, 88
hypothetical reasoning, 9, 12, 19, 35

implication, 11, 14
implicit representation, 19
incomplete temporal knowledge, 22
incomplete temporal knowledge, 19
incomplete temporal knowledge, 15, 19, 21
inequality, 33, 56
inference, 9, 11
infix, 18
initiator, 20
interpretation, 11
InteRRaP, 76
iterative deepening, 48

KB, 14, 19, 22
knowledge base, 14
knowledge base, 19, 22, 24, 67, 68, 72

layer, 76
leaf, 13
least commitment, 41
least commitment, 39, 46
linear planning, 9, 15, 19, 21
linearisation, 15, 22, 71
linearity assumption, 19, 82
list, 18, 25, 52, 68
loading dock scenario, 76
loading dock scenario, 10
localised planning, 82
logic, 11, 13, 18
logical connective, 11
logical programming, 16

maintenance, 50, 82, 86
Malc'ev's Lemma, 33
MAS, 76
means-end analysis, 13
meta programming, 56, 65
meta-logic, 11
method, 52, 66, 68
MGU, 13
model, 11
monotonicity, 13
most general unifier, 13, 29
multi-agent blocksworld, 74
multi-agent domain, 10, 15, 53, 83

negation, 11, 30, 39, 45, 47
negation as failure, 30
negation as failure, 9, 24, 55
negation normal form, 34
negative property, 24
negative abductive goal, 41
negative precondition, 24
NF, 30
nonlinear planning, 9, 15, 19, 21, 82
nonmonotonicity, 9, 13, 18, 19, 25, 30, 46
number, 52, 68

object orientation, 9, 17, 52, 66, 83
object state, 66
object-oriented abduction, 9, 66

parallelism, 15, 19
path, 13
pattern of behaviour, 76
persistence, 86
plan, 13, 18, 76
plan evaluation, 16
plan abstraction, 10, 15, 88
plan analysis, 9, 16, 45
plan evaluation, 9, 45
plan graph analysis, 83
plan modification, 9, 14, 16, 45, 83
plan synthesis, 9, 14, 45
plan/action execution, 53
planning, 9, 13
planning from scratch, 14
planning from second principles, 87
positive precondition, 24
positive property, 24
postcondition, 9, 14, 18, 45, 86
precondition, 9, 14, 45, 86
predicate, 11
predicate logic, 11
program verification, 53
proof, 29
property, 14, 18, 19, 45
proposition, 11, 14

quantification, 11
quantifier, 11, 29

rational tree, 52, 68
reactive architecture, 53
reactivity, 76
reasoning, 9, 11, 18
record, 52, 68
reflexivity, 35
refutation, 29
regression, 20
reification, 89
relation, 11
representation, 9, 11, 13, 18
residue, 35
resolution, 9, 12, 29, 45
resource specification, 13
role, 14, 69
role constraint, 26

satisfiability, 11, 33
scheduling, 13
scope, 11
search, 13
search tree, 13
selection rule, 9, 29, 47, 59, 65
semantics, 11
semi-linear planning, 83
situation, 13, 18, 19, 45
situation abstraction, 16
skolem term, 35
solution, 11
soundness, 9, 11, 14, 16, 18, 39, 45
start situation, 14, 19, 22, 24, 46, 48, 67

state, 13, 18, 52
statement, 11
strong nonlinearity, 19
strong condition, 87
strong nonlinearity, 9, 15, 21, 23, 25, 53
structure, 11
sub formula, 11
subgoal, 14
substitution, 9, 12, 29, 33, 45
subsumption, 14, 19
success, 14, 19, 21, 22, 25
suspension, 55, 59
symmetry, 35
syntax, 11

term, 11
terminating computation, 24
terminating computation, 23, 25, 47, 48
the frame axiom, 21
the persistence axiom, 24
the abduction step, 35
the abduction step, 9, 56
the extended Event Calculus of EVE, 69
the extended Event Calculus of EVE, 50
the extended Event Calculus of EVE, 16
the extended Event Calculus of EVE, 25, 47
the frame axiom, 19
the frame problem, 9, 18, 82
the Halting Problem, 11
the persistence axiom, 20, 22, 48
the qualification problem, 14
the resolution step, 29
the simple Event Calculus of EVE, 49
the simple Event Calculus of EVE, 16
the simple Event Calculus of EVE, 69
the simple Event Calculus of EVE, 47
the simple Event Calculus of EVE, 24, 25
the Sussman Anomaly, 19
the travelling salesman problem, 13
the Turing Machine, 11
the Event Calculus by Missiaen, 22
the Event Calculus by Shanahan, 20
theorem, 11
theorem proof procedure, 45
theorem proof procedure, 9, 11
theorem proving, 9, 16, 45
theory, 9, 11, 18, 45
thread, 52
time, 9, 18, 20, 45
transformation function, 9, 12
transition function, 13, 18
transitivity, 35, 46
truth, 11
tuple, 68
type hierarchy, 14

unification, 12, 16, 29, 33, 68
unifier, 12, 29
universal quantification, 11, 29, 34
universe, 11, 68

variable, 11
variant, 29

weak condition, 87
weak nonlinearity, 15, 21
worst case assumption, 19, 24, 87