Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Research Report
RR-01-01

Theory and Practice of Hybrid Agents

Christoph G. Jung and Klaus Fischer

November 2001

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Postfach 20 80
67608 Kaiserslautern, FRG
Tel.: + 49 (631) 205-3211
Fax: + 49 (631) 205-3210
E-Mail: info@dfki.uni-kl.de

Deutsches Forschungszentrum für Künstliche Intelligenz

DFKI GmbH

German Research Center for Artificial Intelligence

Founded in 1988, DFKI today is one of the largest nonprofit contract research institutes in the field of innovative software technology based on Artificial Intelligence (AI) methods. DFKI is focusing on the complete cycle of innovation — from world-class basic research and technology development through leading-edge demonstrators and prototypes to product functions and commercialization.

Based in Kaiserslautern and Saarbrücken, the German Research Center for Artificial Intelligence ranks among the important "Centers of Excellence" worldwide.

An important element of DFKI's mission is to move innovations as quickly as possible from the lab into the marketplace. Only by maintaining research projects at the forefront of science can DFKI have the strength to meet its technology transfer goals.

DFKI has about 115 full-time employees, including 95 research scientists with advanced degrees. There are also around 120 part-time research assistants.

Revenues for DFKI were about 24 million DM in 1997, half from government contract work and half from commercial clients. The annual increase in contracts from commercial clients was greater than 37% during the last three years.

At DFKI, all work is organized in the form of clearly focused research or development projects with planned deliverables, various milestones, and a duration from several months up to three years.

DFKI benefits from interaction with the faculty of the Universities of Saarbrücken and Kaiserslautern and in turn provides opportunities for research and Ph.D. thesis supervision to students from these universities, which have an outstanding reputation in Computer Science.

The key directors of DFKI are Prof. Wolfgang Wahlster (CEO) and Dr. Walter Olthoff (CFO).

DFKI's six research departments are directed by internationally recognized research scientists:

- Information Management and Document Analysis (Director: Prof. A. Dengel)
- Intelligent Visualization and Simulation Systems (Director: Prof. H. Hagen)
- Deduction and Multiagent Systems (Director: Prof. J. Siekmann)
- Language Technology (Director: Prof. H. Uszkoreit)
- Intelligent User Interfaces (Director: Prof. W. Wahlster)

In this series, DFKI publishes research reports, technical memos, documents (eg. workshop proceedings), and final project reports. The aim is to make new results, ideas, and software available as quickly as possible.

Prof. Wolfgang Wahlster
Director

Theory and Practice of Hybrid Agents

Christoph G. Jung and Klaus Fischer

DFKI-RR-01-01

This work has been supported by a grant from The Federal Ministry of Education, Science, Research, and Technology (FKZ ITW-01IW810 4).

ISSN 0946-008X

Theory and Practice of Hybrid Agents

Christoph G. Jung and Klaus Fischer

November 20, 2001

Abstract

Hybrid agents integrate different styles of reactive, deliberative, and cooperative problem solving in a modular fashion. They are the prime device of (Distributed) Artificial Intelligence and Cognitive Science for realising a broad spectrum of simultaneous functionalities in application domains such as Artificial Life, (Tele-)Robotics, Flexible Manufacturing, and Automated Transportation.

In this report we propose a design methodology for hybrid agents which combines complementary approaches of Software Engineering and declarative Cognitive Robotics at five interconnected specification stages: Architecture, Computational Model, Theory, Inference, and Implementation.

The design methodology is then applied to the reconstruction of the layered agent model InteRRaP ('Integration of Reactivity and RAtional Planning'). InteRRaP, in spite of its practical success in the past, suffers from its originally architecture-centred and informal description. The result is an agent model, InteRRaP-R (the additional 'R' stands for 'Resource-adapting'), which maps its layered architecture onto the formally described interplay of concurrent processes whose runtime is scheduled by meta-control: The processes of a higher layer control the processes of the subordinate layer by the allocation of (computational) resources. Processes are continuous computational activities that are realised as situated inference procedures which implement well-defined subsets of a common logic of time and action.

Three representative scenarios are chosen for an evaluation of InteRRaP-R: the Automated Loading Dock, the RoboCup Simulation League, and the ROTEX space robot. These case studies confirm the applicability of our slogan "**Agent = Logic + Architecture**" to the theory and practice of hybrid agents.

Contents

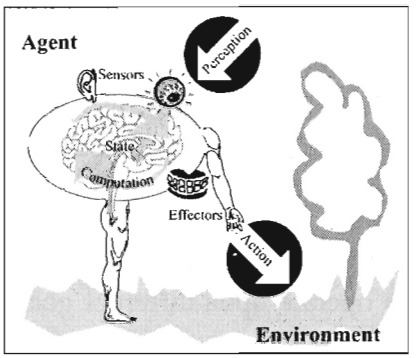

Figure 1: Agent and Environment

1 The Design Space of Agents

There is currently a growing interest in basic research and industrial applications of *intelligent agents*. Following modern textbooks on (Distributed) Artificial Intelligence (DAI) and Cognitive Science [RN95] the term agent describes a self-contained computational structure, i.e., a state and a corresponding calculation which exist in a separate *environment*. The agent perceives the environment through *sensors* and acts upon the environment through *effectors*. Figure 1 illustrates this view which is close to the definition of a *robot* [Rei78], but is not necessarily the view of a hardware realisation (see the *softbots* of, e.g., [Etz93]).
This generalisation makes sense in that many physical and virtual environments share the same requirements for decentralisation, for handling an inherent complexity, and for coping with open and heterogeneous settings. The intelligent agent subsumes many ideas that, for example, originate in the early Shakey project [Nil84], and provides a natural metaphor for addressing these requirements.
Due to the broadened perspective and due to new insights into the agent as a *situated* entity, an enlarged set of agent properties, initially proposed by Wooldridge & Jennings [WJ95], is nowadays commonly agreed upon:

- **Autonomy:** Agents should be able to perform the majority of their problem solving tasks without the direct intervention of humans or other agents. Hence, they should have a degree of control over their own actions and their own internal state. *Autonomy* is the central property also according to [RN95].

- **Responsiveness, Reactivity:** Agents should respond in a timely fashion to changes that occur in their environment, i.e., they are *reactive* and perform in real-time.

- **Pro-activeness and Deliberation:** Agents should not simply act in response to their environment, but also exhibit *goal*-directed behaviour to take initiative where appropriate. We speak of *deliberative* abilities in this respect and presume *rationality*: From its current *belief* (or state), the agent *decides* (chooses) *intentions* (*plans* as sets of basic actions) in order to achieve its goals. The agent avoids any measure that it believes to conflict with its goals. This is the starting point for more restrictive and formal agent definitions in the literature [CL90, RG91].

- **Social Abilities:** Agents should be able to interact with other artifacts and humans in order to complete their own problem solving. In particular, they should be able to help others with their activities if *cooperation* is part of their objective (as it is the case in *Distributed Problem Solving*). This requires that agents can *communicate* their requirements to others. This also requires agents to employ internal mechanisms for deciding when interaction and communication are necessary. These *social* abilities are the key to design *open* systems in which heterogeneous artifacts operate upon different goals and on behalf of different users in a common setting.

- **Adaptivity** Agents should be able to modify their behaviour according to changing environmental and computational conditions. These conditions we shall call *resources*, such as fuel, space, tools, but also CPU-time, memory, etc. *Adaptivity* has a short-term component in compensating dynamic resource changes and a long-term component in *learning* particular *domain* characteristics. The aim of *adaptivity* is to approximate an optimal behaviour with respect to available resources.

  This coincides with the popular principle of *bounded* rationality [Goo76, Sim82] that deviates from the intractable requirement of *perfect* rationality and optimal behaviour regardless of available resources. There are three options to realise bounded rationality [Zil95]: We distinguish (i) *resource-adapted* systems that are pre-designed to the fixed resources of a domain. *Resource-adaptive* systems (ii) are 'somehow' able to react to changing resources of a domain at runtime. Generic agent *models*, however, should be customisable to a range of domains and should cope with a variety of resources. Hence, they favour the third option of *resource-adapting* systems (iii) that incorporate explicit representation and management[1] of resources.

Autonomy is the key requirement for any agent where it is often sufficient that only one further property is satisfied, i.e., either reactivity, pro-activeness, social abilities, or adaptivity. Other agent features, which are in our opinion not primary, are *mobility* and *benevolence*.

1.1 Motivation: Broad Agents

With the growing importance of intelligent systems, especially in the Internet, the application areas for agent technology become considerably larger: According to figures from Ovum Ltd. [Gui95], agent-based software will comprise up to 20 percent from an estimated 4.000.000.000$ segment of the year 2000's software market containing Groupware, Personal Digital Assistants, User Interfaces, Workflow Management, Network Management and Information Retrieval. Subsequently, almost all major companies such as Microsoft, IBM, Apple, AT & T, Siemens, Anderson Consulting and Daimler-Benz, have launched strategic developments in these areas.

[1] Russell & Subramanian argue that any such management is itself subject to resource consumption and thus prevents optimality [RS95]. We define the task of a resource-adapting system only to approximate optimality — as already mentioned in [RW91], its results can often be compiled into simpler, implicit forms of control afterwards.

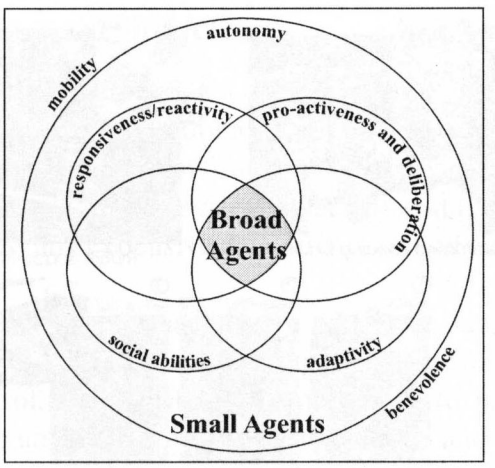

Figure 2: Small versus Broad Agents

These applications are usually based on *small* agents, i.e., software models with a small band-width of functionalities. Small agents have a relatively uniform design that is well-researched in theory and practice. Small agents are currently heading towards commercialisation as ready-made tools which are just to be filled with additional domain knowledge (or application code, in the narrow sense). This is called Agent-Oriented Programming (AOP) [Sho90].
The Ovum study does not look into 'real-world' applications, such as in Artificial Life, (Tele-)Robotics, Flexible Manufacturing, and Automated Transportation. These areas will play an important role in future Information Technology (IT) applications in which the strict border between virtual and physical settings will dissappear, for example in intelligent buildings. In comparison to traditional softbot domains, these areas have a significantly more demanding profile and require *broad* agents as defined by Bates [BLR92] (Figure 2).
Originally, the term broad agent describes life-like inhabitants of synthetic worlds as entities that simultaneously display a whole range of shallow (cognitive) abilities, rather than exhibiting a single ability exceedingly well. For our purposes, we would like to extend this definition into the direction of bounded rationality: Broad agents are computational entities that solve a range of cognitive problems, from reactive behaviour over deliberative tasking up to social interaction, in an approximately optimal manner. Depending on available resources, they trade-off the quality of their decisions versus the cost of computation and interaction.
Since Ferguson's seminal work on *hybrid* Touring Machines [Fer92], broad agents constitute a very active scientific field where the term 'hybrid' characterises a design to integrate several small-agent methods. Many prototypical designs, such as InteRRaP [Mül96] ('Integration of Reactivity and RAtional Planning'), have been built and analysed, many design methods have been proposed, and many representative application scenarios have been defined (see, e.g., Figure 3). In order to settle the field and to enable an industrial impact, it is now time to filter the gamut of methods into a common design *methodology*[2] which bridges theory and practice. It is time to obtain

[2]Webster's Dictionary defines methodology as 'a body of methods, rules, and postulates employed by a discipline; the analysis of the principles or procedures of inquiry in a particular field'. In the

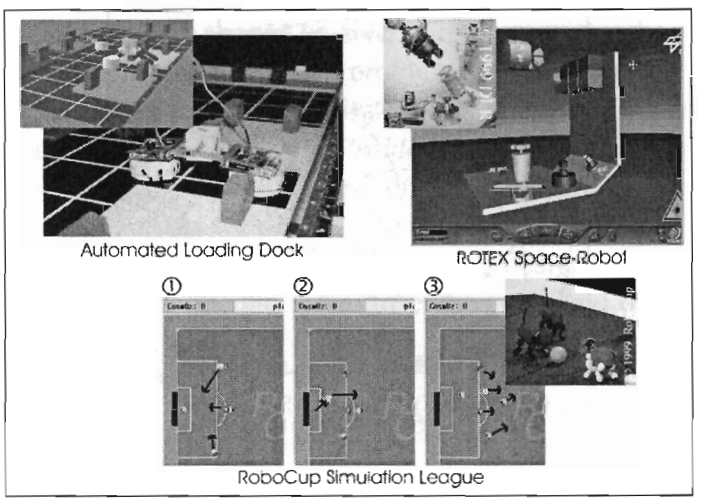

Figure 3: Representative Scenarios for Broad Agents

methodologically-funded models that successfully operate in a range of domains.
This report represents a step towards such a design methodology, the *Design Space of Agents*, and provides a particular design substrate, the InteRRaP-R model (the additional 'R' stands for 'Resource-adapting'), whose rational reconstruction is based on that methodology. In doing so, we do not stick to a high-level analysis and integration of design patterns. Instead, our work is equally driven by the needs of a concrete *implementation* that, following the AOP-paradigm, is practically customisable to demanding domains (Figure 3):
The Tele-Robotics experiment ROTEX [BLSH95] of the 'Deutsche Luft- und Raumfahrtgesellschaft' (DLR) has been on-board of the Columbia space shuttle and is an example of both a reactive and a deliberative control problem. Also the Khepera forklifts in the Automated Loading Dock, a testbed for InteRRaP as defined in [Mül96], have to combine problem solving with sensor-motor feedback. In addition, their social aspect is emphasised: Without coordinated transportation, the loading dock is not to be served optimally by the robots.
Virtual versions of robotic scenarios can be at least as demanding, sometimes even more sophisticated than many physical domains. An example is the official simulation league of the RoboCup initiative [KTS+98] (Figure 3) which is based on a simulation of conventional robot soccer tournaments. Each RoboCup player is modelled as a separate agent with imperfect sensors and effectors. The timing constraints for reactive soccer skills, such as catching, kicking, aiming, tracking, and positioning, are close to real-time (100 milli-seconds). Deliberative capabilities are needed to realise tactical behaviour, i.e., to perform reasonable soccer moves in attack or defence. Social capabilities are necessary for a strategic team-play, thus for coordinated tactics that assign particular roles to the players. Finally, RoboCup players must compensate changes in environmental resources, e.g., the limited stamina of their simulated body, and computational resources, e.g., the limited time that is available to adapt the agent's computation according to the rapidly altering status of the game.

context of broad agents, a design methodology hence comprises the different languages and notations with which we describe their structure and communicate about it.

1.2 Agent = Logic + Architecture

Before bounded rationality developed to a common denominator for AI and Cognitive Science, the appropriate notion of rationality, and hence the choice of agent design methods, had been a highly controversial subject. While early symbolists concentrated on building perfect knowledge-based systems (see Nilsson [Nil84]), the *New AI* community has argued against any expensive data-structures and computations (see Brooks [Bro91]).
Both research streams can be seen as extreme, resource-adapted instances of small agent design. Because of making particular types of decisions, i.e., either high-level tasks or low-level control, their systems are optimised to particular classes of domains. Because of being unable to adjust to varying needs and resources, their systems show severe drawbacks in broad domains, such as the Automated Loading Dock, the RoboCup simulation, or the ROTEX work-cell: It is difficult to force an inherently myopic reactive system to goal-oriented behaviour. It is equally difficult to force an inherently complex planning algorithm to responsive behaviour.

Agent Engineering Hybrid agents have been developed to integrate the reactive, but myopic mechanisms initiated by Rodney Brooks with optimal, but expensive deliberation facilities, such as planning. A particular example of such a modular design is InteRRaP (Figure 4). InteRRaP has a *layered* structure for the combination of a reactive *Behaviour-Based Layer*, a deliberative *Local Planning Layer*, and a *Social Planning Layer*. Each layer is associated with computations on a particular level of representation. Each layer supplements its subordinate layers in order to put through more abstract and more persistent goals and decisions.
With respect to bounded rationality, hybrid agents provide a resource-adaptive trade-off between computational costs and solution quality, i.e., between reactive, deliberative, and social abilities. As such, they have already proven quite successful in constructing broad agents for real-world and virtual-world domains (see the assessment of [Mül99a]).
In order to fill their designated role in industrial-strength systems, however, hybrid models face a fundamental engineering problem in that they lack a clear design methodology. Up to now, their description is usually given in an informal *architectural* manner. This pragmatic method of specification introduces very crude and abstract concepts and leaves many design issues open. Hence, the space of possible *implementations* does not necessarily reflect the original objectives, such as a practical trade-off between reactivity, deliberation, and social abilities.
Moreover, by integrating a variety of modules from various backgrounds, hybrid models are not easily comprehensible. This complicates the identification of appropriate programming constructs and impedes their customisation to various domains. We have experienced these difficulties with previous InteRRaP implementations.

Cognitive Robotics Formal *logic* has always been used in the tradition of *theories* of rationality. For example, Cohen & Levesque [CL90], Shoham [Sho90], and Rao & Georgeff [RG91] describe agents in temporal and epistemic logic, thus deviating from earlier informal descriptions in [GL87]. Kowalski & Sadri [KS96b] rely on the

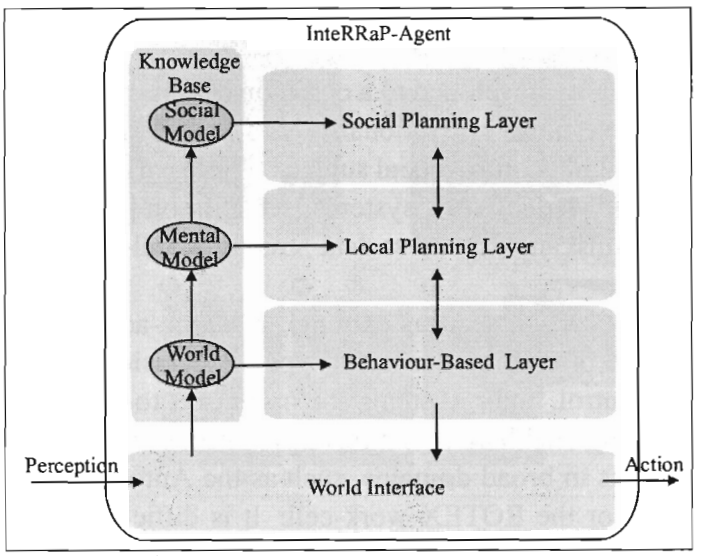

Figure 4: Hybrid InteRRaP Agents (Outline)

power of first-order logic augmented with *abduction* as the declarative basis of a *unified* agent. Especially the latter approach envisages a logic programming (LP) perspective [Kow79] in which the high-level agent axiomatisation is straightforwardly implemented by a special *inference procedure.*

The logic-based specification of agents[3] comes nowadays under the umbrella title Cognitive Robotics [Bow87]. It aims at a coherent, concise, and verifiable design whose declarative concepts can immediately serve as intuitive programming constructs. However, the conceptual level is too high for deriving practical systems: Straight implementations via inference procedures are either not feasible or build on restricted expressiveness; the operational considerations to 'make the theory run' are seldom discussed. To our knowledge, no such monolithic 'rationality engine' has ever been able to master settings that are comparable to those of hybrid systems.

The Design Space of Agents From what we have just discussed, it is apparent that Agent Engineering and Cognitive Robotics are rather complementary: Both ways of specification introduce useful concepts for agents, either on the theoretical side — the logic representations of Cognitive Robotics — or on the architectural side — the modular structures of Agent Engineering. Both lack design issues, either in declarative or in operational respect. Hence, a design methodology that reconciles both approaches in a preferably formal setting seems promising.

Such a methodology helps to set up a well-understood collection of interrelated methods (or *specification stages*) bridging theory and practice. In doing so, such a methodology moreover addresses a matter that both Agent Engineering and Cognitive Robotics have largely neglected up to now, namely how to derive sound implementations in effective programs. The methodology that we are looking for runs under the

[3]There are differences between logic theories for specifying agent computations, such as [Kow79], and logic theories just for describing and verifying agent behaviour, such as [RG91]. We do not engage into a discussion of this issue here, rather stay with the first perspective for our purposes.

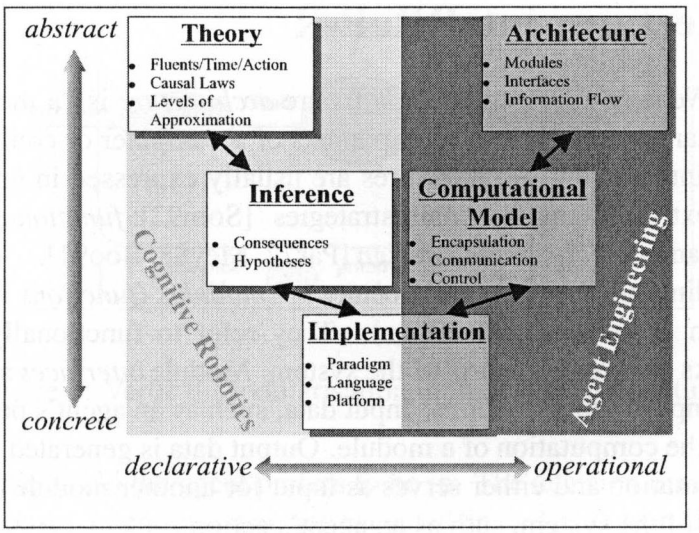

Figure 5: The Design Space of Agents

slogan "**Agent = Logic + Architecture**"[4] and is the basis for reconstructing InteRRaP into an AOP-tool for broad applications.

In Figure 5, we have arranged the specification stages in the common *Design Space of Agents*. This design space is spawned by two independent *dimensions of specification*, namely the degree of abstraction and the degree of declarativity. Architectural Engineering turns out to be a rather abstract and operational enterprise while Cognitive Robotics covers the declarative side of the design space. Agent implementations are most concrete; although they are a too low-level medium for research, their connection to the higher-level specification is nevertheless of justified interest.

To complete the Design Space of Agents, the point of concern is to find an operational complement to the inference stage that is able to capture the architectural features of hybrid agents, e.g., their *modularisation*, in formal and computational terms. We call such descriptions *computational models* as inspired by formal programming [Hoa69]. Computational models are written in dedicated specification languages, such as Z [Spi92]. They describe the state and the operation of a kind of agent 'interpreter' running inferences in a particular logic. As *formal specifications*, they already became a successful tool in modern Software Engineering and are just about to enter (D)AI and Cognitive Science.

Computational models are the missing 'puzzle-piece' to our design methodology because they are sufficiently **abstract** to formally connect to high-level conceptualisations, sufficiently **concrete** to derive concise agent implementations, sufficiently **declarative** to integrate a theoretical perspective, and sufficiently **operational** to capture architectural considerations.

[4] Derived from the path-setting motto of Kowalski: "**Algorithm = Logic + Control**"

2 Architecture: InteRRaP-R

According to Webster's Dictionary, a software *architecture* is "a method or style of building; the manner in which the components of a computer or computer system are organised and integrated". Architectures are usually expressed in graphical notation and informal text following two main strategies [Som92]: *functional design* [CY79, Wir71, Wir76] and *object-oriented design* [Par72, Mey88, Boo91].
The basic building blocks of an architecture are *modules* (*functions* in functional design; *objects* in object-oriented design). They refer to functionally self-contained ([Som92] speaks of *cohesive*) parts of the system. Module *interfaces* are usually given in the form of input-output relations: Input data, such as an agent's perception, is necessary to drive the computation of a module. Output data is generated as a result of the module's computation and either serves as input for another module or comprises the ultimate output of the system, such as an agent's action.
Primitive modules are black-boxes; their computation and their inner state are not visible to the outside and usually remain not fully specified. In contrast, *compound* modules can be recursively decomposed into more primitive sub-modules and corresponding interfaces. Compound architectures are the result of an iterated design process which starts with the overall computer system as a single primitive module.
With respect to agents, such a starting point as illustrated in Figure 1 is given by the *physical symbol hypothesis* of Newell & Simon [NS76] which postulates cognitive behaviour as a symbol-manipulating activity and which is the basis of classical AI systems. Their 'architecture' hence consists of a single module carrying the agent's mental state (*knowledge base*). The mental state receives perceptual input and computes output actions by a rational reasoning procedure. This is too coarse a design for practical purposes and poses the problem to the agent designer: Which refinement of this picture is needed in order to cohesively address the functionalities that are required from broad systems running in, e.g., Automated Loading Dock, RoboCup simulation, and ROTEX work-cell?

2.1 Deliberative Agents and Vertical Modularisation

The refinement of compound modules imposes structure onto the system, thus determines the possible *interactions* (*coupling* according to [Som92]) between modules. For example, if we combine the output of one module with the input of another, we introduce a sequential chain in which the former module provides a service to the latter. In general, modules servicing each other introduce a *vertical* modularisation of a system, where the direction of service provision, e.g., which module is the client and which is the server, determines the *flow of information*. Usually, *bidirectional* interactions allow for feedback (reporting errors, mutually accessing data) between modules and are a more flexible choice than *unidirectional* chains.
Vertically arranged structures have been rather prominent in agent design and are for example demonstrated by the 'Procedural Reasoning System' agents (PRS) of [GL87] (Figure 6). PRS is organised along 'mental categories', such as believes, goals and intentions, which coincide with different functional roles in a deliberative agent. The belief data base or knowledge base is responsible for maintaining a representation of the external world from incoming perceptual data. Goals are spawned to indicate which

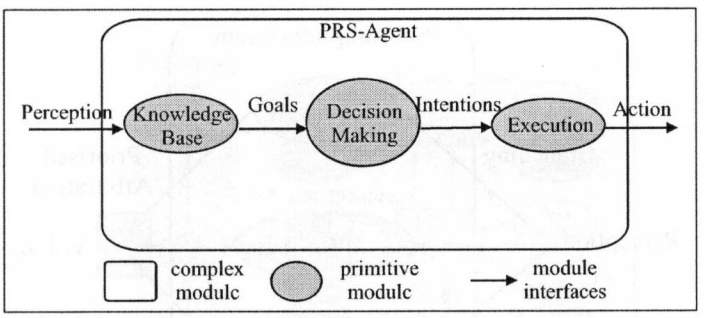

Figure 6: Vertical Modularisation in the PRS Architecture

future states of the world the agent prefers to be in. The decision making or planning module then chooses intentions that are executed in order to turn the world into a desired state. Hence, PRS is a unidirectional architecture.
Many variants of this design have been built [BIP87, Sho90], for example by exchanging some of the modules by others with different computational properties. Bidirectional module interfaces have also been introduced, since the manifested goals and plans can now be communicated back to the knowledge base in order to, e.g., reason about failure. Such nested accessibility of mental categories, i.e., believes about goals and intentions, is a central proposition of BDI ('Belief, Desire, Intention') theories [CL90, RG91, RG95, Woo96] of rational agents. Nevertheless, PRS has been theoretically and empirically shown to be close to BDI [KG91]. For these and other reasons (there exist a well-defined computational model[dKLW98] and several efficient implementations, such as dMARS of the Australian Artificial Intelligence Institute and UM-PRS of the University of Michigan), PRS is today one of the most successful agent architectures. It offers a convenient way to structure, e.g., the task planning of forklift robots in the Automated Loading Dock as well as the tactical planning of soccer agents in the RoboCup simulation.

2.2 Reactive Agents and Horizontal Modularisation

Brooks [Bro86, Bro91], Agre and Chapman [AC87], Kaelbling [KR90], Maes [Mae90] and others voiced a fundamental criticism on deliberative agents questioning the value of an explicit representation of the world ("Let the world be its own representation."). They argue that the aggregation of complexity within a sequential reasoner is intractable for most practical domains, such as robotics. The trick to severely restrict the expressiveness of a decision making module is regarded as an impasse.
Their alternative method envisages a *horizontal* modularisation of the agent where input is sent in parallel to several modules that independently compute possibly conflicting output. These modules hence compete for providing a service ([Som92] speaks of *concurrent* systems design). Since the computational complexity of each module is bound, the *Subsumption* architecture of Brooks [Bro86] and its derivates are proposed as a suitable scheme for building reactive agents.
Brooks' architecture decomposes the agent according to its activities (or *competences*). Figure 7 shows a possible realisation of a robot with three competences, namely driving along a road, avoiding collisions with obstacles, and approaching other robots.

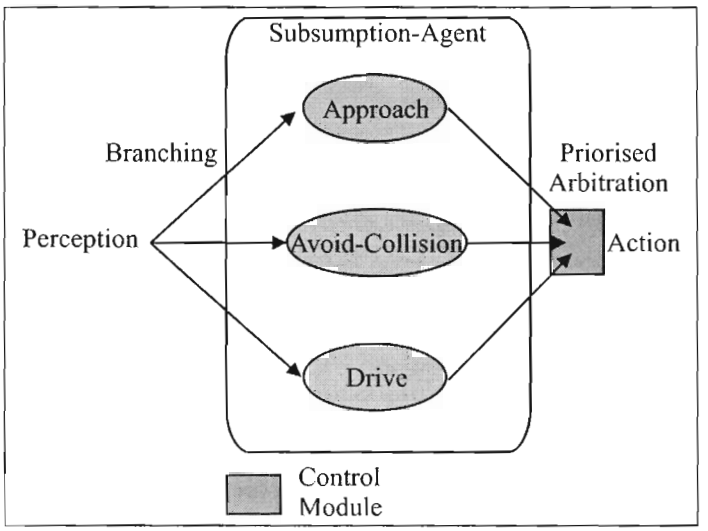

Figure 7: Horizontal Modularisation in the Subsumption Architecture

These competences are evaluated in parallel or concurrently at different levels of a hierarchy where each level is immediately coupled to sensor input (the input *branches* into several modules) and effector output. For example, the drive module computes steering commands from perceptual data of the road ahead.
In general, we distinguish between an *uncontrolled* and a *controlled* form of horizontal modularisation. In its uncontrolled form, there is no priority for any module. The *arbitration* of output, i.e., the choice of the module that is granted to provide the service, is unspecified. In its controlled form, we identify higher-level modules that have priority over lower-level ones. Hence, their output overwrites or at least influences the final action in any case.
In order to combine the output of different competences in the Subsumption architecture, higher-level competences overwrite the output of lower-level competences in an arbitration module. For example, if an obstacle appears, the avoid-collision module *suppresses* the driving commands and stops the motor. If the obstacle is a cooperative agent and should be approached, the approach module is getting priority over avoid-collision in turn. Control is given to the lower-level modules only as long as there is no output of higher-level modules. As a result, the functionality of the lower-level modules is thus *subsumed* by the higher-level modules.
Controlled horizontal modularisation turns out to be very successful for the realisation of real-time robot behaviour (even in hardware) and has stimulated a whole sub-discipline of AI. Subsumption-type of architectures are useful to handle, e.g., the sensor-motor feedback of forklift robots in the Automated Loading Dock as well as the reactive soccer skills of RoboCup agents.

2.3 Hybrid Agents

Deliberative and reactive architectures, i.e., the vertical and horizontal ways of structuring agents, have different advantages and drawbacks. Vertical arrangements are good for realising rationality, but their inherent complexity is problematic for installing rapid

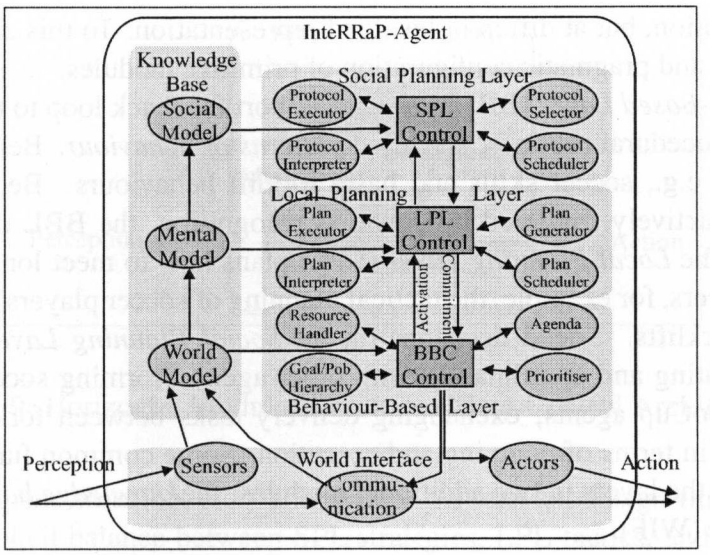

Figure 8: Vertical and Horizontal Modularisation in InteRRaP

feedback loops. Horizontal arrangements are good for responsive and real-time behaviour, but they have difficulty to make persistent and goal-oriented decisions. Hence, both types of design deal with different aspects of broad agents, such as either soccer skills or soccer tactics[5], either low-level control of forklifts or their task-oriented problem solving.

Consequently, *hybrid* and *layered* agents [Fer92, Fir92, LH92, Dab93, BKMS95, SP96] integrate both methods in order to benefit from a horizontally as well as a vertically structured architecture. A typical arrangement[6] is realised in Firby's RAP [Fir92] and Bonasso et al.'s 3T [BKMS95]. They distinguish a *planning layer*, a *sequencing layer*, and a *behaviour layer* which are organised in a *layered hierarchy*. This is similar to the organisation of competences in the Subsumption architecture, but only the behaviour layer carries reactive computations. It is triggered by the sequencing layer's execution of plans which have been synthesised by the planning layer.

For different applications, other constellations have been developed. The Touring Machines of Ferguson [Fer92] have a *plan modification layer* which modifies the plans of a planning layer according to environmental conditions. Lyons & Hendricks [LH92] proposed a planning layer which modifies the behaviour of a reactive layer. Sloman & Poli's SIM_AGENT [SP96] has a *management process* which operates upon plans and reflexes. A non-hierarchical design with reactive, deliberative and diagrammatic (quasi-analogue) computations, all of the same priority, has been proposed in [Pia99]. InteRRaP [Mül96] ('Integration of Reactivity and Rational Planning' — Figure 8) aims at bringing the hybrid and layered design closer to BDI, at the same time addresses the need of coordination among multiple agents. The three InteRRaP layers, the reactive *Behaviour-Based Layer*, the deliberative *Local Planning Layer*, and the *Social Planning Layer* each integrate the BDI-cycle of goal activation, decision making, and

[5]That the BDI design of [BHW98] became champion in 1997 was due to its highly domain-dependent implementation.

[6]Interestingly, a similar design has already been proposed for the Shakey robot back in the 60's [Nil84].

intention execution, but at different levels of representation. To this end, each layer is given a special and pragmatic configuration of primitive modules.

The *Behaviour-Based Layer* (BBL) provides a short feedback loop to the environment by applying procedural routines, so-called *patterns of behaviour.* Behaviour patterns correspond to, e.g., soccer skills and basic forklift behaviours. Because behaviour patterns are reactively triggered by situation recognition, the BBL decision making is quite fast. The *Local Planning Layer* (LPL) plans how to meet long-term, abstract goals. This covers, for example, the tactical planning of soccer players and the delivery planning of forklifts. Social decisions (at the *Social Planning Layer* — SPL) that involve negotiating and coordinating with other agents (forming soccer strategies in a team of RoboCup agents; exchanging delivery tasks between forklift robots) are also expressed in terms of planning and execution. Some common functionalities are separated from the layers in two additional modules: the*knowledge base* (KB) and the *world interface* (WIF).

Hybrid architectures have been successful in a range of demanding domains, especially in building service robots [Mül99a]. For example, [Mül96, Ros96] demonstrated InteRRaP to handle the low-level motion and manipulation of loading dock forklifts via the BBL, to plan the delivery tasks within the LPL, and to coordinate transportation conflicts in the SPL. Still, InteRRaP and hybrid designs in general face two architectural problems which impede their usefulness as industrial-strength tools.

First, they are complex beasts including a variety of heterogeneous modules. Figure 8 shows that the structure of each InteRRaP layer is rather arbitrary and incomprehensible at first sight. Moreover, structures often change over the different layers. As a result, major code changes are even needed in the agent kernel in order to run the implementation of [Ros96] (that was very much tight to the loading dock), e.g., in the RoboCup. Hence, for developing an agent programming language à la AOP, a cohesive and congruent architectural structure is a necessary prerequisite.

Secondly, layering is often regarded as being equivalent to horizontal modularisation. In InteRRaP, layers interact via particular control modules and based on two mechanisms: *upward activation* and *downward commitment.* If a goal cannot be resolved by a lower layer, it is posted upwards to be handled by the next upper layer. In turn, if the upper layer has made some decision, it will send particular commitments down to the next lower layer. The result is quite close to the interplay of competences in the Subsumption architecture [Bro86]: priorised decisions of the higher-layer modules compete for execution with the lower-layer decisions; lower layers are integrated into the functionality of the higher layers.

Due to the different computational needs of its layers, InteRRaP is an instance of a resource-adaptive system (see our definition in the previous section). Changing resources in the environment and the computation device influence the quality of actually executed decisions: If the environment is reasonably calm, it is likely that the LPL can timely influence the fast decisions of the BBL. If the environment becomes more dynamic, the BBL will constantly act without the LPL being able to intervene. This is implicitly encoded into the architecture and makes it quite difficult for the higher-layer computations to have a meaningful impact in real-time domains. [Mül96, Ros96], for example, 'abused' the WIF to install another, hidden forklift layer.

Moreover, demanding environments, such as the Automated Loading Dock and the RoboCup simulation, are full of changing resources (time, state of the body, battery

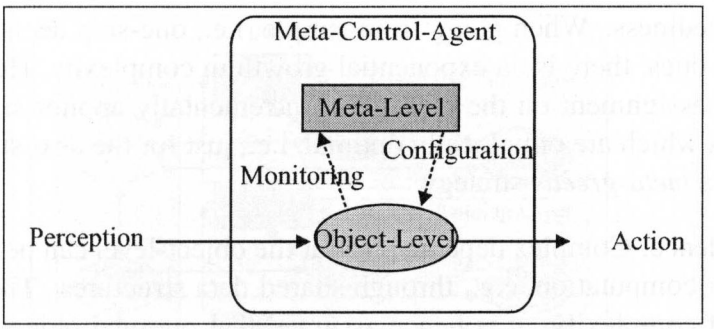

Figure 9: Horizontal Modularisation in a Meta-Control Architecture

load, stamina, game situation, etc.) and require the relevant design-time conventions, such as the implicit balance between SPL strategies, LPL tactics, and BBL skills, to be a part of the run-time decisions of the agent. In order to turn this trade-off into an explicit parameter of the InteRRaP architecture and in order to make the higher-level computations feasible within real-time domains (100 milliseconds in the automated loading dock and the RoboCup simulation), the module interaction between layers has to be revisited.

2.4 Meta-Control Agents

In parallel to hybrid agents, a special form of controlled horizontal modularisation has been put forward by people from AI and Cognitive Science [And93, Hor86, BD94, RW91, RS95]. Their ideas are based on the observation that one major ingredient to human intelligence is the ability to reflect the changes in environmental and computational resources. Hence, resource-adapting mechanisms are proposed as an architectural principle to trade-off situated computations, such as the balance between strategies, tactics, and skills inside a RoboCup agent, according to overall system constraints.

Following [Goo76, Sim82, RS95], a boundedly rational agent has to solve two nested optimisation problems. The first problem is to optimise its external behaviour with respect to environmental resources. The second problem is to modify its internal computation with respect to computational resources. Russell and Wefald [RW91] developed a representative *meta-control* architecture (Figure 9) in which each of those optimisation problems is assigned a dedicated module: The *object-level* module works like a traditional planner that makes complex decisions how to act in the environment. It is *monitored* and *configured* by an additional *meta-level* module. For this purpose, the meta-level assigns computational resources to the various computational options at the object-level.

In contrast to the branching & arbitration in reactive architectures and the activation & commitment in layered architectures, the monitoring & configuration interfaces decouple the meta-level from the timing that the environment imposes onto the object-level. Nevertheless, the meta-level also consumes computational resources and so it should be of neglectable complexity. In Russell and Wefald's design, this is accomplished by the following assumptions:

- Meta-greediness: When going from simple, i.e., one-step decision problems to complex ones, there is an exponential growth in complexity. Hence, a tractable resource assignment on the meta-level incrementally applies simple allocation decisions which are only locally optimal, i.e., just for the next step in time. This is called a *meta-greedy* strategy.

- Independence: Complex dependencies at the object-level can be caused by side-effects of computation, e.g., through shared data structures. They are a further source of complexity. It is hence assumed that optional object-level computations are approximately *independent* with respect to computational resources, i.e., choosing one particular option does not influence the outcome of other possibilities.

- Sequential Object-Level: Russell and Wefald's meta-level restricts to assigning a single computational resource, namely computation time, to the object-level computations. Hence, time serves as a unified representation for all interdependencies between object-level options and is allocated sequentially.

Russell and Wefald report the successful application of their architecture to game-playing under time constraints, such as chess. Other prominent meta-control approaches share several of their ideas: Anderson's *ACT-R* architecture [And93] uses a *rational analysis* (meta-)module to control the operation of a rule-based production system and to learn parameters and utilities of object-level rules. *Flexible algorithms* [Hor86] provide a special form of object-level whose outcome monotonically improves over time. *Anytime algorithms* [BD94] are flexible algorithms that are continuously interruptible and have been given dedicated meta-controllers in terms of *deliberation scheduling* algorithms. Deliberation scheduling is empirically shown to provide a quasi-optimal behaviour in navigation planning and targeting.
However, meta-control architectures have drawbacks when it comes to the construction of broad agents. Real-time conditions are difficult to handle with a single, sequential object-level module. Moreover, the simplifications of the meta-level are not always applicable: Interactions with other agents (between the soccer players of a RoboCup team; between the forklift robots in the loading dock) and interdependencies between optional object-level computations (such as different soccer skills and different low-level forklift behaviours) go beyond the consumption of time. For example, aiming at the opponent's soccer goal improves the result of kicking afterwards. Finally, there is a conceptual problem with the separation into computational and environmental resources, since they are substitutable: By withdrawing computation time from a module, its taking actions in the external world can be prevented.

2.5 The InteRRaP-R Architecture

The architectural contribution of this thesis, the InteRRaP-R[7] architecture (Figure 10), is a generalisation of the InteRRaP model, the design of Russell & Wefald, and the

[7]This name is inspired by Anderson's switch from ACT* to ACT-R. Where the 'R' in ACT-R however emphasises the impact of meta-control to bounded 'Rationality', it should refer in InteRRaP-R rather to its 'Resource-adapting' architecture.

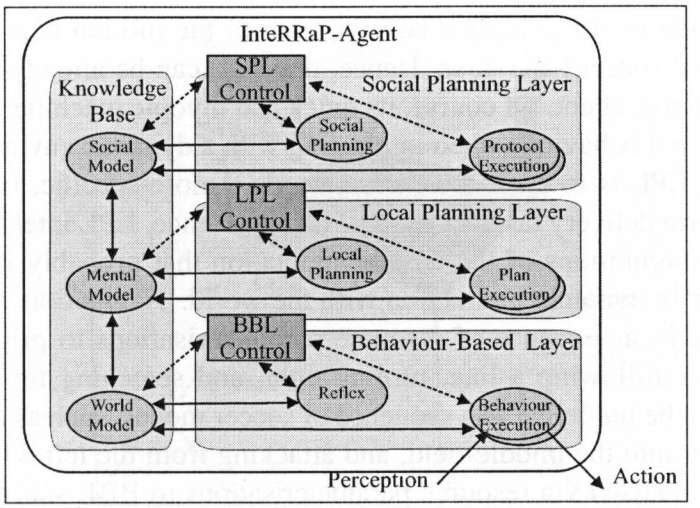

Figure 10: InteRRaP-R: Layering as Meta-Control

PRS. It builds upon the observation that meta-control provides the desired resource-adapting principle for layering in hybrid agents.
InteRRaP-R resembles the original InteRRaP being divided into a reactive, a deliberative, and a social layer. Each layer, however, is now congruently structured like a Russell & Wefald agent into a (meta-level) control module and several object-level modules. The object-level modules are arranged similarly to the PRS design.
The important difference to InteRRaP is that upper layers do no more immediately interact with the environment, but their decisions parameterise the control module of their subordinate layer towards a more rational (in the case of the LPL configuring the BBL) and social (in the case of the SPL configuring the LPL) behaviour. Lower layers are thus no longer subsumed by the upper layers in the sense of Brooks. Upper layers no longer operate under the same timing constraints as their lower companions which renders their more expensive computations more feasible.
The important difference to the Russell & Wefald agents is that InteRRaP-R introduces the explicit management of computations according to situative resources at several levels of the architecture, i.e., at the reactive, the deliberative, and the social layer. InteRRaP-R computations are not simply bounded by time, but they are controlled with respect to general interdependencies that are grounded in environmental as well as computational restrictions. The representation device which we use to this end is called *abstract resources.*

2.5.1 Layering as Meta-Control

The decisions of an upper layer in InteRRaP-R guide or restrict the resource allocation of the control module of the subordinate layer. Using the illustrating example of RoboCup: Deliberative soccer tactics and moves at the LPL are realised as computational guidelines for the executing reactive soccer skills inside the BBL; team strategies that are negotiated and decided by the SPL in turn represent constraints on the planning and execution of soccer tactics within the LPL.
A lower layer, such as the BBL, therefore provides in-principle all the functionality

of the agent, such as the complete basis to control the motion of a forklift robot or to exhibit skillful soccer behaviour. Hence, the BBL can be already thought of as an autonomous reactive agent. Of course, its quick and myopic machinery allows a whole corridor of external behaviour when interacting with a dynamic environment.
The aim of the LPL is to push the BBL towards a more specific, rational function, such as to perform delivery tasks or to score a goal. Hence, LPL intentions are realised as mid-term configurations of the BBL computation that probably lead to a desired 'symbolic' state in dynamic interaction with the world. Navigation, for example, can be implemented as a sequence of resource parameterisations to primitive behaviour patterns, such as following a line, turning right, and searching for a landmark. A soccer attack can be modelled as a sequence of soccer moves, such as defending on the left side, passing into the middle-field, and attacking from the left wing. Each soccer move in turn is realised via resource parameterisations to BBL soccer skills, such as locating, positioning, tracking, kicking, etc.
Similarly, the SPL operates as the supervisor of the LPL with respect to the social coordination issues. In negotiating with other agents, the SPL monitors the state of the local decision making, e.g., to communicate active goals of the LPL planning module and to reason about the currently executed LPL plans. If the commitment to adopt a new goal or the agreement on a particular multi-agent plan is made during a negotiation, the SPL influences the LPL computation accordingly. In the Automated Loading Dock, a computational guideline that the SPL imposes to the LPL could enable the planning for exchanging a box. In the RoboCup domain, the SPL lets the LPL incarnate a tactical role (goal keeper, left defender, etc.) inside a global team strategy.
The BBL is thus no longer subsumed, but supported by its super-layer LPL; layers do not stand in competition, but in a structured, cooperative relation with their super-layers. This form of resource-adapting horizontal modularisation decouples the higher-level reasoning from the critical timing of dynamic environments and renders its representations, such as LPL plans, more persistent and more abstract. For example, the typical timing of soccer moves in the RoboCup is between ten seconds and one minute. This can be reasonably handled by AI planning algorithms. Team strategies are even active for major parts of the game such that a sporadic reasoning and negotiation is possible. Almost identical timings can be stated for the forklift robots in the Automated Loading Dock — this hints to the general scope of our design.

2.5.2 Abstract Resources

The InteRRaP-R control modules operate in a similar fashion to the simple Russell & Wefald meta-level in order to refine the higher-layer guidelines into concrete resource assignments to the supervised modules, such BBL behaviours and LPL plans. The complex reasoning inside LPL and SPL modules can thus focus on long-term conflict resolution, while the simple control modules are responsible for a short-term optimisation. For this purpose, we propose a resource representation which integrates features of both computational and environmental restrictions to the agent's functioning. We call this representation *abstract resources*, because of denoting general interdependencies between or constraints on the (object-level) modules within a single layer.
The clear distinction between architectural and environmental resources does not seem to be reasonable in the broad agent case: modules that, e.g., implement different soccer

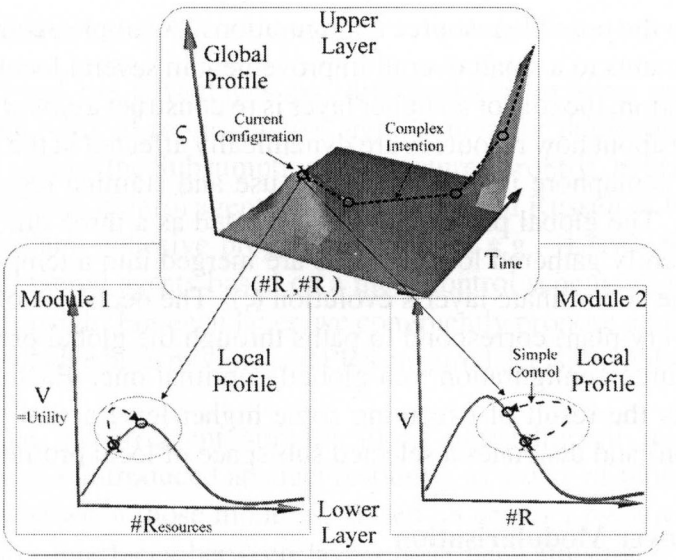

Figure 11: Decision-Theoretic View of Abstract Resources and Layering

skills affect each other in a similar fashion no matter if this happens internally, e.g., by running on the same computing device (aiming to the goal versus aiming to a team mate), or if this happens externally by performing actions in the physical world (positioning versus chasing the ball). If we influence the internal computation of a module, this also has an effect to the external environment.

In the Russell & Wefald architecture, this has been addressed by exclusively and sequentially allocating computation time. However, InteRRaP-R modules, such as positioning, kicking, and tracking the ball, exhibit interdependencies which go beyond the consumption of processing time. At the same time, these modules should be able to compute interleaved in order to support responsiveness (see the following section).

As an alternative representation, we regard an abstract resource as a limited set of 'items' for which several modules apply. For example, the BBL 'landmark' resource is accessed by all location skills; the BBL 'stamina' resource is accessed by all movement behaviours, and the LPL 'role' resource is accessed by the LPL planner and the LPL plan execution modules. Hence, the task of each control module is to decide about the distribution of the items to its subordinate modules.

A rather primitive example of an abstract resource is a unary set. This corresponds to the well-known construct of a *semaphore* in multi-threaded programming languages. The semaphore forces certain activities that apply for it to be executed exclusively, because only one activity is able to get a hold on it and is allowed to compute and act. Different modules can interact through different semaphores, thus a meta-controller is able to schedule several, mutually non-interacting computations at the same time. The semaphore is thus a representation of a selected subspace of computations; assigning it is equivalent to putting guidelines on the object-level modules.

The allocation of abstract resources is based on decision-theoretic considerations about how useful a particular resource assignment is. Each object-level module therefore statistically monitors a so-called *local profile* illustrated as a two-dimensional graph in Figure 11. The local profile describes the current performance or utility of a module

(V) in relation to the possible resource configurations. A simple decision of the control module thus amounts to a small overall improvement in several local profiles.
In this interpretation, the role of an upper layer is to construct a *global profile* by higher-level knowledge about how resources are dynamically affected in the subordinate layer, e.g., a released semaphore is free for further use and stamina regenerates while not running around. The global profile can be illustrated as a three-dimensional space in which the frequently gathered local profiles are merged into a temporal projection or hypothesis of the subordinate layer's evolution (ς). The decisions of a planning layer (a tactic; a delivery plan) correspond to paths through the global profile leading from the current resource configuration to a globally optimal one. Each intermediate step in such a path is the result of executing some higher-level action (a soccer move, a navigation action) and denotates a selected sub-space of local profiles.

2.5.3 Inner-Layer Modularisation

The secondary objective of our redesign of InteRRaP has been to develop a more cohesive architecture. A first step in this direction has been to remove the redundant World Interface (WIF) which originally served to lift sensing and acting onto a more 'symbolic' level. Conceptually, but, the separation into BBL and WIF is not necessary. Our experience has shown that an efficiently implemented BBL can master close-to-real-time settings without an intermediate level of processing (see Section 7) and provides a transparent control through LPL and SPL. Even special-purpose hardware can be integrated by envisaging a distributed implementation model.
InteRRaP has been inspired by BDI logic and its complicated inner-layer structure can thus be replaced by a design that is much closer to PRS and includes a knowledge base module, a decision-making module, and several competing intention execution modules per layer. In the case of the BBL, the decision making module performs a quick computation which we call a reflex; planning inferences are applied in the case of LPL and the SPL. BBL intentions are procedural behaviour patterns much like the competences in the Subsumption architecture turning perceptual sensor data into external actions. LPL and SPL intentions are plans and protocols, respectively, whose execution finally affects the allocation of resources on the next-lower layer.
As opposed to PRS, InteRRaP-R layers are not realised as unidirectional chains. Instead they resemble the design of *blackboard systems* [EM88] in which a central blackboard, the knowledge base module, is bidirectionally coupled to several other modules, here the decision making module, the intention execution modules, the control module (via monitoring & configuration), and the knowledge module of the next higher layer. This design allows sensor data, goals, plans, profiling data, and resource parameterisations to be a processible part of the knowledge base and thus to be accessible from the upper-layer modules. For example, the LPL of our RoboCup agents is able to take the current amount of available stamina into account when setting up a tactic. For example, the SPL of forklift robots is able to communicate about the delivery tasks that a robot intends to perform in its LPL.

2.6 Bottom Line

This section analysed the advantages and drawbacks of existing patterns of Agent Engineering on the basis of representative architectures for deliberative agents (PRS [GL87]), reactive agents (the Subsumption architecture [Bro86]), hybrid agents (InteRRaP [Mül96]), and meta-control agents (the architecture of Russell & Wefald [RW91]). As a consequential and cohesive progression also to, e.g., [LH92, SP96, JC97], we postulate to build layered agents based on a meta-control scheme in order to obtain a resource-adapting design. For each layer, we congruently propose a blackboard design with a central knowledge base module and PRS-inspired decision making and intention execution modules.
In order to escape the stringent simplifications of traditional resource management [BD94], we have introduced abstract resources as a way of representing general interdependencies between those modules. Based on abstract resources, the monitoring & configuration interfaces between layers have been illustrated using decision-theoretic terms. The resulting InteRRaP-R architecture fits naturally the particular needs of, e.g., the Automated Loading Dock and the RoboCup simulation, and will be formally specified in the following.

3 Computational Model: COOP

The problem of describing systems via architectures, i.e., graphical notations and informal texts, is that this introduces ambiguous concepts [Som92]. Thus, many design decisions, such as the scheduling of computations inside a module or the algorithmic realisation of primitive modules, are left to the programmer. Hence, an implementation neither necessarily nor provably reflects the original goals of design, such as the practical integration of deliberation and reactivity in the case of InteRRaP. For example, previous implementations of the original InteRRaP architecture [FMP95, Ros96] show significant performance differences due to different scheduling strategies. This led us to investigate a more concrete and formally agreed way of conceptualising InteRRaP-R in order to remove unwanted ambiguities while upholding a decent degree of generalisation.
Historically, the first detailed *software specifications* were written using proprietary pseudo-code or 'Program Description Languages' (PDL) close to the implementation level. However, as implementation languages have become more abstract, detailed specifications can now be written using a formal mathematical notation. The area of Formal Methods has now reached a stage where it can be used in industrial systems in order to increase system quality and reduce development costs [Hal90].

3.1 Computational Models and Z

There are two types of formal specifications: *algebraic specifications* and model-based specifications (henceforth called *computational models*). Algebraic specifications, such as Hoare's process calculi [Hoa69], focus on the operations that are done upon the modules which makes them particularly useful in object-oriented design. Computational models construct a mathematical model of the system's transition as well as its state. Computational models are useful in both object-oriented and functional design.

A computational model is like an abstract interpreter built according to the system architecture. In order to refine the given module structure, the computational model explores *encapsulation*, i.e., how to divide the system's state and its computation into separate computational *processes*. Encapsulation determines whether there could be *inconsistencies* between processes and also determines whether processes could compute *concurrently* to one another in which case the overall behaviour of the system is composed out of the interleaved or even simultaneous computation of processes. If a computational model guarantees each concurrent process to eventually being able to compute, it is said to be *fair*.
Process *communication* is responsible for the exchange of data between processes and can be used to realise module interfaces. There are *explicit* forms of communication, such as directed *signals* and undirected *alarms*. *Shared memory* is an *implicit* form of communication. A communication channel that looses information is said to be *unsafe*.
If we look at processes, a design option is to choose between *complex* and *simple* algorithms for their realisation and therefore between complex and simple functions as the basic building blocks of the system. Complex results, such as a plan, denote optimal long-term structures that consist of several primitive actions (or output) of the system. Usually, the corresponding procedures, such as planning, turn out to be complex, too, in the sense that their amount of computation increases exponentially in problem size. Simple results, e.g., a single action, are primitive measures that maximise the system's performance just for a single step in time. Often, they can be computed using fairly un-demanding, therefore simple procedures.
The earliest model-based specification technique was VDM (the 'Vienna Development Method' [Jon80, Jon86]) which was developed in the late 1970's and refined during the 1980's. Its sometimes unintuitive semantics has been the reason for basing the language Z [Hay87, Spi92] developed at the University of Oxford on classical typed set theory. The standardisation and the widespread use of Z led to a number of specification tools, such as editors, presentation modules, type-checkers, animators, and theorem provers, and extensions, such as special support for object-oriented design (Object-Z). Z will be used to specify a computational model of InteRRaP-R in the current section.

3.2 A Crash-course in Z

Z is particularly well-suited for incrementally building formal specifications, because they are presented in small, legible structures called *schemas* and are distinguished from associated commentary using graphical highlighting. Operations such as *references*, *renaming* and *hiding* allow schemas to be manipulated in their own right and provide a powerful tool for system specifiers.

DataDictionaryEntry[*DataType*]

$kind : EntryType$	[signature]
$description : \text{seq } DataType$	[sequence]
$\#description \leq 2000$	[predicate]

The preceding schema of a *DataDictionaryEntry* (the schema's *name*) describes a part of the system state, e.g., the state of a process, as a mathematical expression. Its upper, *signature* part defines the structure of entities of type *DataDictionaryEntry*. It introduces two sub-entities under the identifiers *kind* and *description*. In Z, types are seen as sets; operations on types thus are operations on sets. This makes it easy to describe that the *kind* slot must be filled by an enumeration type *EntryType*, thus can take one of the atomic values *data_flow*, *data_store*, *user_input* and *user_output*:

$$EntryType ::= data\_flow \mid data\_store \mid user\_input \mid user\_output \qquad \text{[definition]}$$

Not only types, but nearly all Z-constructs, such as *relations*, *functions*, and subsequently *sequences* (which are represented as functions from the natural numbers to a target type) are couched in set-theoretic terms. Sequences, as well as our *DataDictionaryEntry*, need some extra care, because they are *generic* types which are defined upon arbitrary targets (sequences of characters, sequences of natural numbers, dictionary entries storing real numbers, etc.). They are defined as *generic schemas* with additional type arguments (*DataType*). Conceptually, a generic type is a function which maps the target to an instantiation of the generic type. Hence, the application 'seq *DataType*' delivers the sequences over *DataType*. In order to define a *StringDataDictionary* which maps ($\pfun$ denotes a partial function) a not further specified index set *NAME* to a string-based *DataDictionary*, we thus simply write in an alternative notation:

$$[NAME, CHAR] \qquad \text{[given sets]}$$
$$StringDataDictionary \mathrel{\widehat=} [ddict : NAME \pfun DataDictionaryEntry[CHAR]]$$

The lower part of a schema, the *predicate* part, includes statements which constrain the mathematical structure that is built by the signature. In a logical interpretation, the predicate part shrinks the number of valid models: For example, the sequence length of a *DataDictionaryEntry*'s *description* field must not exceed 2000. Length (#) is one of the many predefined operations on sequences (others are the empty sequence $\langle\rangle$, the head of a sequence *head*, the tail of a sequence *tail*, or the concatenation of sequences $\frown$). In Z, such built-in support is already present for types, such as pairs, numbers, bags, grammars, etc.

The predicate part of the *StringDataDictionary* schema is empty which means no further restriction to the signature, such as a maximal size for *ddict*. This demonstrates the advantage of a model-based approach to formal specification: the specification is only restricted where necessary; it remains open and abstract where choices do not matter. In this respect, the possibility to incrementally refine a previously under-specified system provides a convenient prototyping environment.

Computational models do not stick with modelling the state of the system. They specify as well operational considerations how this state changes over time while running the system. In Z, system operations are modelled by particular schemas, such as the following *MakeNewUserInput* operation on our previously defined *StringDataDictionary*:

MakeNewUserInput

$\Delta StringDataDictionary$	[delta operator]
$entry : DataDictionaryEntry[CHAR]$	
$name? : NAME$	[input to the transition]
$data? : \text{seq } CHAR$	
$name? \notin \text{dom } ddict$	[constraints to the transition]
$entry.kind = user\_input \land entry.description = data?$	[build entry]
$ddict' = ddict \oplus \{name? \mapsto entry\}$	[override]

The Δ decoration to *StringDataDictionary* means that there exists a corresponding substructure *ddict* before the state transition that is related to a resulting changed data structure $ddict'$ of the same type. System operations are performed in the light of additional input decorated with a question mark (*name*? and *data*?). System operations can also produce output that is decorated with an exclamation mark (!).
Similarly to states, transitions are constrained by the predicate part of a Z schema. In the case of *MakeNewUserInput*, the resulting dictionary $ddict'$ is not an arbitrary entity, but is derived from the former *ddict* by mapping the input *name*? to a freshly created string-based *DataDictionary* (using the override operator $\oplus$ and the pair constructor $\mapsto$). *MakeNewUserInput* also requires the input name not being in the domain (dom) of the initial dictionary. Fields of substructures are accessed as *entry.kind* and *entry.description*.
State transitions, such as for specifying processes, can hide unnecessary information, such as the exact determination which of several processes should be run at which time. For example, on of the last steps in a Z specification collects the alternative operations of a system into an overall schema:

$StringDataDictionaryProcess \mathrel{\widehat=}$ [schema collection]
$MakeNewUserInput \lor MakeNewUserOutput \lor MakeNewDataFlow \lor$
$MakeNewDataStore \lor MakeNewError \lor RetrieveData \lor$
$RetrieveError \lor RemoveData \lor RemoveError$

For a detailed overview of all the Z constructs and their semantics, we refer to reference manuals, such as [Spi92] from which we lend above *DataDictionary* example.

3.3 COOP: A Computational Model of InteRRaP-R

3.3.1 Formal Specifications in Agent Design

The first efforts to pin down agent architectures with more formalised descriptions used pseudo-code notations. For example, [Mül96] gave an object-oriented sketch of InteRRaP modules and their interfaces. However, such languages have no computational semantics and leave many questions open, especially the issue of concurrency. Moreover, taking a purely object-oriented design stance, similar to the use of formal transition systems, algebraic specifications, and process logics [HdBvdHCM98, Fis93b], delivers a too operational view onto an agent. This is especially because of leaving a large gap to the declarative methods of Cognitive Robotics [Bow87] in which logic theories and inferential frameworks are developed that determine how to intuitively represent an agent's environment and how to perform reasoning on these representations.

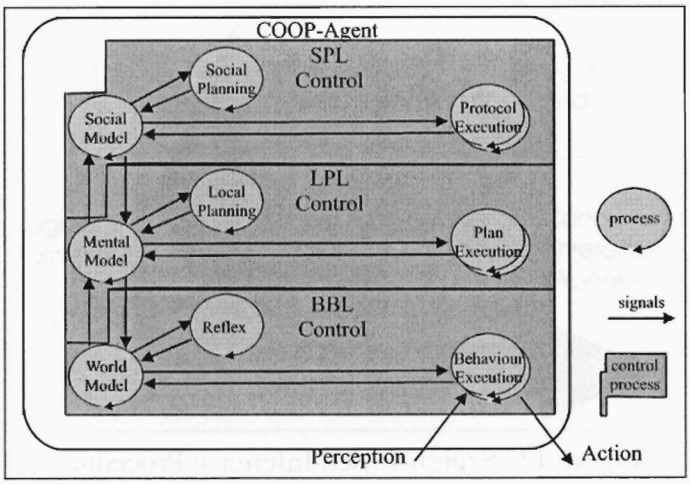

Figure 12: Specifying InteRRaP-R via Processes, Signals, and Control

Hence, for the purpose of the Design Space of Agents, we argue that computational models, such as those specified in Z, are the missing link between Cognitive Robotics and Agent Engineering as well as between conceptualisation and implementation, since they are sufficiently **abstract** to formally connect to high-level conceptualisations, **concrete** to derive concise agent implementations, **declarative** to integrate a theoretical perspective, and **operational** to capture architectural considerations.
This has been first recognised by [Woo95] who gives a computational model of the experimental MyWorld architecture using VDM. Because of its standardisation and support, the Z language has been used to specify variants of the PRS architecture [dL98, dKLW98]. While these efforts focused on unified agents, i.e., models with a low degree of modularisation, DESIRE [DKT94] is a proprietary formalism for describing arbitrary compound agents.

3.3.2 A Computational Model of InteRRaP-R

In the remainder of this section, we develop a consistent computational model of the InteRRaP-R architecture. We use the Z notation [Spi92] which makes the model easy to read (due to Z typesetting), clearly to understand (due to a standard interpretation), and accessible to further processing (due to the available Z tools).
Being guided by the requirements of, e.g., the Automated Loading Dock and the RoboCup simulation, we shall use constructs that were developed in computational models of modern programming environments [Smo95, AG96] and reactive systems [AZ87, BW96]. In particular, we introduce a concept of encapsulated processes which compute continuously and concurrently to each other and this is why our computational model is called COOP ('COncurrent, cOntinuous Processes' — Figure 12). Processes are able to communicate explicitly via information-transmitting signals and implicitly via shared memory. Their *exception handling* facility allows a flexible reaction to new, possibly inconsistent information.
COOP processes, such as knowledge base processes, planning processes, and behaviour execution processes, are closely related to *threads* in concurrent programming. They are more special in that they do not represent arbitrary one-shot computations,

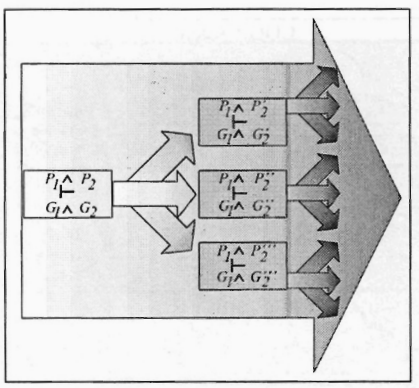

Figure 13: Search-Based Inference Procedure

but are mapped to well-defined subsets of a logic of action and time. Hence, primitive InteRRaP-R modules are modelled as persistent and interactive COOP processes which are in turn regarded as encapsulated inferences in a common logic theory.
The only exception to this logic interpretation are the InteRRaP-R control modules. They are realised as simple allocation algorithms for abstract resources, so-called *control processes*. Ordinary processes, such as behaviour execution and local planning, consume abstract resources while computing. The control process monitors their resource consumption and their performance. By its management of a particular type of *trigger* signals, the control process is able to frequently optimise the allocation of fresh abstract resources to processes, hence to optimise the active computations inside its associated layer. The complete formal specification which has some superficial relations to labelled deductive systems [Gab96] is given in Subsection 3.4.

3.3.3 Concurrent Inferences

The first requirement for a computational model of an intelligent agent is the necessity to model rational inferences, such as for performing navigation in the loading dock and for setting up a tactic in the RoboCup. The lesson that we learn from Cognitive Robotics is that even reactive ingredients, such as collision-avoidance or ball tracking, can be reasonably described as special-purpose inferences operating on a low level of representation [KS96b]. Therefore, the modules of InteRRaP-R can be expressed as inference procedures that operate on logic representations. We call these procedures *inference processes*.
There exist many inference procedures in the literature. All of them can be described as a step-wise rewriting of logic formulae (Figure 13). The state of an inference procedure consists of a program formula P (an assumption, a theory) and a goal[8] formula G (an observation, a call). The task of an inference procedure is to find extensions P' to the program P from which G follows logically, i.e., G is a consequence of P' ($P' \models G$). For this purpose, the inference procedure constantly strengthens the information in P

[8]It should be noted that we use the term goal in two related, but different meanings. An *agent goal* as described in Section 2 is a data structure with a particular functional role inside the perception-action cycle. A *logic goal*, in contrast, plays a representational role inside an inference process within the perception-action cycle. Thus, we can use logic goals to encode agent goals, but logic goals also realise other concepts, such as observations and requests. We will come back to this issue in Section 5.

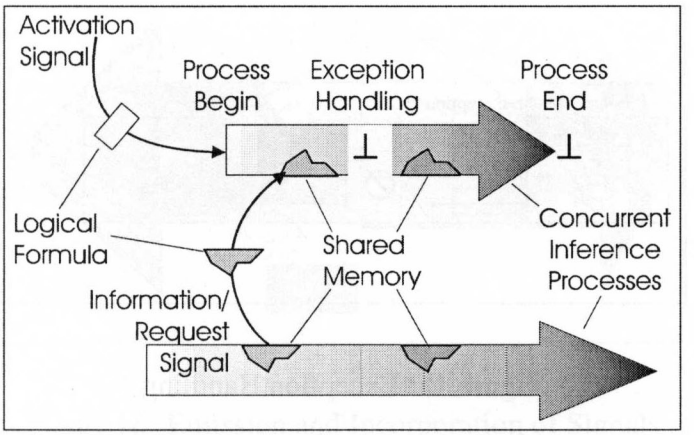

Figure 14: COOP: COncurrent, cOntinuous Processes

and G while keeping the logic meaning. G' that results from a process transition thus implies G under the extended theory P'. If the goal evaluates to true ($\top$), the procedure has been successful.

It is not always possible to find all extensions of P and instantiations of G without changing this canonical representation. Hence, we allow inference processes to pursue several *options* of rewriting the initial formula whose combination, i.e., disjunction, keeps the information of the predecessor expression. This is a technique which is widely used in logic programming and combinatorial search. An inference process thus maintains a set of optional programs and goals. In each step of the process, one option is selected one sub-formula of which is transformed into a set of successor options.

We propose a single logic and a corresponding inference framework to formally specify InteRRaP-R modules in terms of inference processes (Sections 4 and 5). Each process refers to a restricted sub-language of the logic and hence implements a well-defined subset of the inference framework.

However, COOP distinguished from unified, logic-based agents such as proposed by Cognitive Robotics in that its state and its computation are encapsulated into rather independent parts (Figure 14). Independence in state is manifested by COOP processes operating on different and partially inconsistent formulae. Independence in computation is manifested by COOP processes operating concurrently to each other. Hence, the state of InteRRaP-R is a possibly conflicting combination of the state of its inference processes and its operation amounts to the interleaved operation of these.

Both issues are important features of reactive systems [AZ87, BW96] for supporting the overall responsiveness of the agent: The BBL, for example, must be able to meet quick decisions interleaved and even inconsistent with the computations of the higher layers. In the loading dock, collision-avoidance must be able to dodge the robot even if this renders the planned movement impossible. In the RoboCup, the kick behaviour must be able to clear the ball quickly out of the field, even if not getting into ball possession.

This is not to say that a sequential, consistency-preserving computational model cannot exhibit such interactive behaviour. High-level reasoning and consistency-checking are costly tasks. Hence, the programmer of such a system has to add 'scheduling

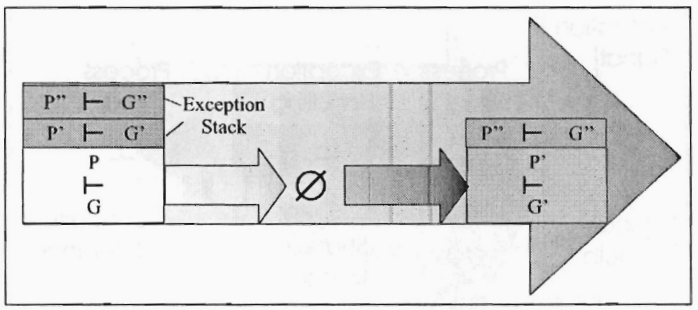

Figure 15: Exception Handling

knowledge' into his domain representations, such as the behaviours, the plans, and the protocols. Analogue to multi-purpose programming platforms, however, we argue that a good agent model should integrate those facilities which are required in most envisaged domains in order to ease the programmer's task.

3.3.4 Continuous Processes and Exception Handling

There is a danger in allowing too much concurrency and inconsistency which is the lack of persistence. In the mid-term perspective, an agent should behave rational and work constantly on particular tasks. Hence, processes, such as local planning and behaviour execution, are not one-shot operations which vanish after having provided a service, i.e., after having inferred $\top$. Instead, they are continuous computations that are active over a longer period of time. This period could even comprise the complete life-cycle of the agent which are 10 minutes in RoboCup and up to one day in the Automated Loading Dock.

For example, the local planner does not disappear after having decided about a single goal, but waits for new goals to extend its current state. For example, an avoid-collision reflex is not finished after having dodged the robot once, but computes further on until a particular obstacle should be deliberately approached. Hence, COOP processes will not die unless they finally fail, i.e., there exist no more inference options anymore.[9]

Inference processes are situated inside the agent just as the agent is situated inside its environment. Hence, they must compute with incomplete information and under dynamic changes of the environment. In particular, they must make preliminary assumptions which turn out to be inconsistent afterwards. Instead of simply failing as a result of foreseeable inconsistencies, COOP processes own mechanisms to restore their operation. Advanced inference principles are one way of achieving this kind of incrementality. In addition, COOP processes employ an *exception handling* mechanism that shields dangerous computations against predictable failures (Figure 15).

Exception handling has been introduced in multi-threaded programming languages [BW96] in order to introduce fault recovery. It is a convenient way to separate the regular way of program execution from the recovery actions. In a logical setting, recovery actions can be established as a set of optional inferences ready to replace the

[9] We regard the distinction into successful and failed operations as an unfortunate concept for situated agents because their outcome can only be evaluated in the eye of the beholder, such as the LPL which surveys the operation of BBL processes.

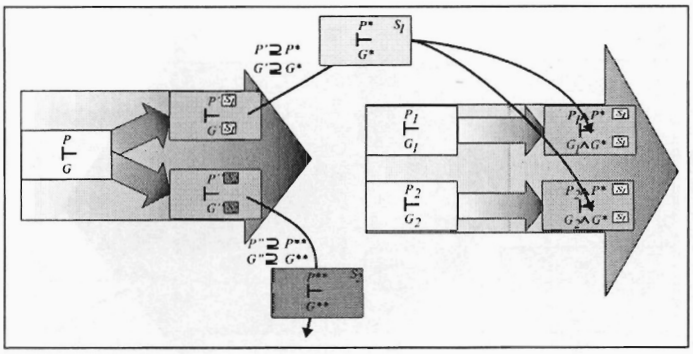

Figure 16: Emission and Incorporation of Signals

regular state of the process. While running deeper into the solution of a problem, an *exception stack* is provided with more and more specific information about possible failures. In the case of failure, the exception handler then determines the most specific continuation for the process.
Continuity is important to ensure persistent behaviour. Nonetheless, for subduing inconsistencies and concurrency between task planning and low-level robot control or between active tactics and conflicting soccer skills, additional measures have to be introduced. These are the exchange of information, i.e., formulae, between processes and particular control processes to optimise active computations within a layer.

3.3.5 Signals and Shared Memory

COOP signals transmit information from one process to another in the form of logic formulae. Depending on their content, we distinguish different types of signals, such as *information signals* which transmit a part of a logic program, and *request signals* which transmit a logic goal. Signals are generated as a by-product of inference. Figure 16 illustrates that they have to be consistent with their emitting inference option, i.e., they carry a a subset of its program and its goal.
Signals represent commitments of the process to its environment: Information signals commit to certain hypotheses or assumptions about the world, such as a plan that has been decided by the local planning process. Request signals commit to particular tasks to be performed, such as a desire that is activated by a knowledge base process. As such, signals are incorporated into every option of the recipient process.
Further types of signals are *trigger signals* that are able to resume the activity of a suspended recipient process and *activation signals* which are used to initialise new processes. Activation signals are sent out by knowledge bases processes in order to create new intention execution processes (behaviours, plans, protocols).
Due to its logic content, a signal could only outlinea particular fact or a particular task. For example, the knowledge base could activate the desire to deliver an arbitrary box within the planner where the box is not fully determined. In return, the knowledge base would like to know the result of calling the planner's service, e.g., the identification of the selected box, or it would like to specify the box in more detail, when getting additional evidence. Using logic formulae as shared and incrementally refined data structures, an implicit way of information exchange can be introduced for that purpose

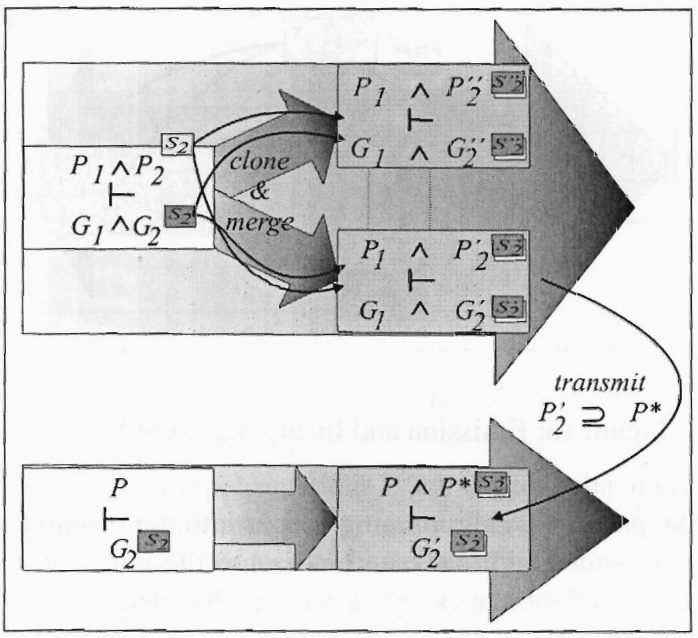

Figure 17: The Shared Memory Model

(Figure 17).
In COOP processes, the formulae are annotated with the signals by which they have been communicated. This means that there are shared references within the state of signal sender and signal recipient. During the performance of an inference upon such a shared formula, all referring processes are automatically transmitted a part of the rewritten information: Not every option, for example every possible instantiation of the delivery desire, has to be communicated, but possibly a non-empty subset thereof. Not every hypothesis, for example every detail of a generated plan, is communicated, but only a part of it. Shared goal formulae, in contrast, are completely communicated and rewritten, because they represent the 'shared tasks'.
Inference steps rewrite several sub-formulae of selected option. Inference steps also have several options as their result. An appropriate labelling mechanism *merges* and *clones* signals to ensure that process annotations always refer to identical formulae.

3.3.6 Control Process, Internal Profiling, and Resource Allocation

Control processes in COOP differ from deliberation schedulers [BD94, Hor86] in that they do not develop a costly long-term scheduling policy which would be ineffective in the real-time dynamics of, e.g., robotic domains. Similar to the Russell & Wefald's meta-level [RW91], control processes in COOP envisage a simple optimisation of the ongoing computations, i.e., they temporally minimise mutual conflicts while maximising performance. The concurrent setting, however, requires extended representations and scheduling algorithms:

1. We have already discussed that, unless sequentially scheduling processes, computation time is a bad resource representation. Hence, we use *abstract resources*, such as soccer 'stamina' and 'aim', for describing the interdependencies of processes both with respect to environmental restrictions — positioning and chasing

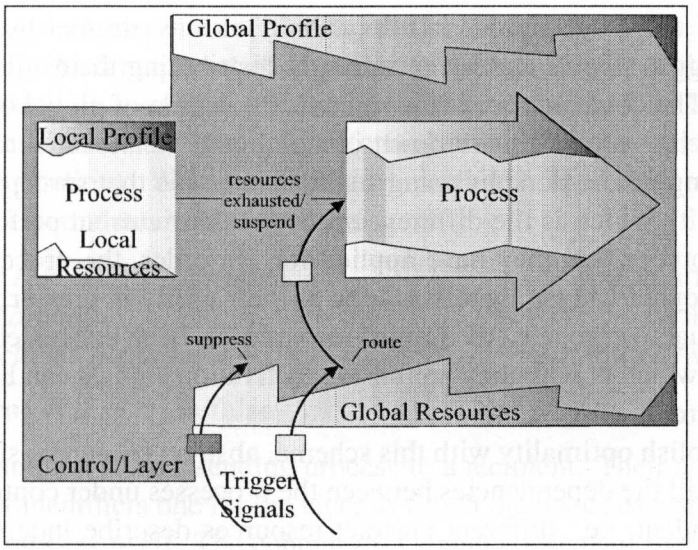

Figure 18: Trigger Signals and Control Process

the ball meet conflicting action decisions and diminish stamina — as well as with respect to computational restrictions — aiming to the goal and aiming to a team mate redundantly consume time and memory. Abstract resources are modelled as sets of items denoting the available amount of that particular resource. Processes 'consume' these items by each step of inference. Once a process, such as chasing the ball, has exhausted its local resources, such as its amount of stamina, it is suspended. To resume activity, a new resource allocation from the control process is required.

2. To make useful allocation decisions, we need to estimate the performance of processes. *External profiling* mechanisms evaluate computations at hand of sporadic feedback from sensor data (scoring a goal or loosing the ball in the RoboCup) and are suitable to guide such short-term decisions. Instead, we propose a built-in self-evaluation of processes. Their transitions produce performance values from which an average local profile is generated. For example, a RoboCup aim process that has to evaluate the distance and the direction to the opponent's goal will frequently determine the likelihood of a successful scoring. This is called *internal profiling.*

3. To uphold responsiveness, we cannot synchronise a scheduler with each step of computation in COOP. Instead, the control processes operate asynchronously and interleaved. To this end, they are able to control the computations via controlling their communication. Control processes collect all outgoing and incoming signals to the processes under their control. Particular trigger signals must be frequently routed to a process in order to transmit new allocations of resources, thus to resume its processing. For example, to resume an aim process, a change of the player's relative position to the goal has to be transmitted from the knowledge base. As long as a process is active, trigger signals are collected. As soon as the process suspends, the stored signals are used to reallocate resources and to resume the process.

In each step of the control process, allocation decisions are met by filtering the set of collected trigger signals and by accordingly distributing the available resources to the processes. The control process uses a static allocation of global to local resources per process, such as a particular amount of stamina to be allocated to positioning. In this case, filtering can be done by using a simple heuristic that sorts processes according to their utility which is the difference between their current performance and the cost of the resources that they have applied for. In order, the processes are granted resources and routed trigger signals as long as their utility is greater than zero. Once the utility is equal to zero, e.g., an applied resource has been exhausted, trigger signals are suppressed which is why they are unsafe; activation signals can be routed without assigning resources.
In order to establish optimality with this scheme, abstract resources should be exhaustive, i.e., cover all the dependencies between the processes under control. They should also be independent, i.e., different abstract resources describe independent conflicts between processes. Our experience has shown, however, that the scheme is tolerant with respect to small changes in the modelling and installs an approximately optimal short-term control.
To achieve global optimality, control processes are guided by the decisions of the next upper layer. Therefore, resource representations, profile data, and parameterisations of the control process including static allocations, process*emphasis*, and process *discount* are shared between control processes and their associated knowledge base process. This allows, for example, to decompose the soccer moves in a RoboCup agent's LPL into configurations for its BBL control process. The goal keeper's defending move , for example, sets a particular home position (in terms of belief) and assigns small portions of stamina for frequent positioning (in terms of allocation). At the same time, chasing the ball receives a high priority (in terms of emphasis) and a large portion of stamina for sudden ball interception (in terms of allocation, again).

3.4 Formal Specification

We now formalise the outlined agent interpreter by using the Z notation. First, we concentrate on defining the state of InteRRaP-R by mathematical definitions and schemas (Subsection 3.4.1). Afterwards, we turn to the transitions of InteRRaP-R modelled as schematic relations between these states (Subsection 3.4.2). The following specification has been successfully type-checked using the ZTC tool [Xia95]. Some auxiliary definitions can be found in Appendix A.

3.4.1 The State of InteRRaP-R

The most primitive constructs within InteRRaP-R are *identifiers* of type *Identifier* and (logic) formulae of type *Formula*. Identifiers are constants that are used to address processes and layers inside the agent. For the moment, formulae shall belong to an arbitrary (first-order) logic injected into the COOP model. In Section 4, we present a dedicated first-order theory for that purpose. Identifiers and formulae are not further specified and are thus introduced as given sets.

$[Formula, Identifier]$ [given sets]

Next, we define the conjunction and the disjunction of formulae. They are realised as relations *conj*, *disj* which map a set of formulae to the equivalents of their conjunct, or disjunct respectively. In the following definition, we use the notation $\mathbb{P}\,Formula$ which is the power set of *Formula*. The unique formula **true** (**false**) represents the empty conjunction (disjunction). We also distinguish particular identifiers *perception*, *action* that address the external environment of the agent.

$$conj, disj : (\mathbb{P}\,Formula) \leftrightarrow Formula \qquad \text{[boolean combinators]}$$
$$\mathbf{true}, \mathbf{false} : Formula \qquad \text{[unique formulae]}$$
$$perception, action : Identifier \qquad \text{[unique environment id's]}$$

$$conj(\varnothing) = \mathbf{true} \land disj(\varnothing) = \mathbf{false} \qquad \text{[logic laws]}$$

A *Signal* is transmitted from a sending process to a recipient. Their addresses are encoded as pairs of identifiers one for the layer at which the respective process is located and one for the process itself. The content of a signal consists of two conjunctions of formulae (in the *range* of the *conj* relation, $\operatorname{ran} conj$), the first denoting a logic program and the second denoting a logic goal. Any signal can be *cloned* with respect to a particular process *Option* — this is a forward reference to an upcoming schema. Cloning means to copy the signal into a different, but unique instance of *Signal* and is realised as an injective function.

Signal

$$sender, recipient : Identifier \times Identifier \qquad \text{[addresses]}$$
$$program, goal : \operatorname{ran} conj \qquad \text{[two-fold content; conjunctions]}$$
$$clone : Option \rightarrowtail Signal \qquad \text{[clone function]}$$

A clone always carries the same content and the same addresses as its original. Furthermore, the transitive closure of cloning ($clone\star$) must be anti-reflexive in order to exclude cycles and its must map different signals to disjoint clones.

$$clone : Signal \leftrightarrow Signal \qquad \text{[cloning as a relation]}$$

$$\forall s : Signal \bullet clone(\!|\, \{s\} \,|\!) = \operatorname{ran} s.clone \qquad \text{[turn function into relation]}$$
$$\forall s : Signal;\ b : Option \bullet \qquad \text{[clones behave like their original]}$$
$$((s.clone)(b)).recipient = s.recipient \land ((s.clone)(b)).sender = s.sender \land$$
$$((s.clone)(b)).program = s.program \land ((s.clone)(b)).goal = s.goal$$
$$\forall s : Signal \bullet s \neq clone\star(s) \qquad \text{[closure is anti-reflexive]}$$
$$\forall s_1, s_2 : Signal \bullet clone\star(s_1) = clone\star(s_2) \Leftrightarrow s_1 = s_2 \qquad \text{[disjoint clones]}$$

The following definitions introduce special sub-types of signals: *TriggerSignal* contains a selection of signals that will be used to control the computation of processes. *InformationSignal*, *RequestSignal* and the environmental signals *PerceptionSignal* and *ActionSignal* are characterised through their content and addresses. We use the Z set construction ($Set == \{Signature \mid Predicate \bullet Result\}$) for that purpose.

$$TriggerSignal : \mathbb{P}\,Signal \qquad \text{[is a selected subtype of signal]}$$
$$RequestSignal == \{s : Signal \mid s.goal \neq \mathbf{true} \bullet s\} \qquad \text{[particular signal subtypes]}$$
$$InformationSignal == \{s : Signal \mid s.program \neq \mathbf{true} \bullet s\}$$
$$ActionSignal == \{s : Signal \mid s.recipient = (action, action) \bullet s\}$$
$$PerceptionSignal == \{s : Signal \mid s.sender = (perception, perception) \bullet s\}$$

We now give the schema for a single *Conjunct* which is the COOP representation of a formula that is annotated with a set of signals. Straightforwardly, any set of *Conjunct* is translated to a regular formula by stripping off annotations (*translateConjunct*).

$$Conjunct \mathrel{\widehat=} [state : Formula;\ annotation : \mathbb{P}\, Signal] \qquad \text{[annotated formula]}$$

$$translateConjunct : \mathbb{P}\, Conjunct \to Formula \qquad \text{[translate set of conjuncts]}$$

$$\forall pc : \mathbb{P}\, Conjunct \bullet translateConjunct(pc) = conj(\{c : pc \bullet c.state\})$$

InferenceState is the structure behind a particular process *Option* and an exception, i.e., an alternative continuation of inference. *InferenceState* consists of two sets of *Conjunct*, one for the logic program and one for the logic goal. These sets are interpreted as conjunctions, thus must exist in the domain dom of *translateConjunct*. An *Option* is an *InferenceState* which additionally maintains an exception stack, i.e., a sequence of alternative *InferenceState*s. An initial option *InitOption* has empty content and empty exception stack.

$$InferenceState \mathrel{\widehat=} [program, goal : \mathrm{dom}\ translateConjunct] \qquad \text{[two-fold state]}$$

$$Option \mathrel{\widehat=} [InferenceState;\ exceptions : \mathrm{seq}\, InferenceState] \qquad \text{[shielded inference]}$$

$$InitOption \mathrel{\widehat=} [Option \mid program = \varnothing \land goal = \varnothing \land exceptions = \langle\rangle]$$

Before turning to process design, we need to introduce abstract resources, process performance, and utility in advance. Abstract *Resources* will be modelled in terms of particular resource items *Item*, such as the *role* resource in a soccer team consisting of several *goalies*, *defenders*, *mid – fielders*, and *attackers*, or such as the *stamina* resource of a soccer player consisting of several stamina *unit*s. The continuous *UtilityScale* from which we draw performance and utility measures ranges from 0.0 to 1.0.

$$[Resource, Item] \qquad \text{[given sets]}$$

$$UtilityScale == \{r : \mathbb{R} \mid r \geq 0.0 \land r \leq 1.0 \bullet r\} \qquad \text{[continuous utility scale]}$$

A *ResourceValue* is a partial mapping of *Resource* to multi-sets of *Item*, such as a current line-up of a soccer team having 1 *goalies*, 4 *defenders*, 3 *mid – fielders*, and 3 *attackers*, or such as the current *stamina* comprising 2.000 *unit*s. Multi-sets can be modelled as functions from *Item* to natural numbers $\mathbb{N}$. A *ResourceAllocation* could consume as well as produce resource items and maps each *Resource* to a function from *Item* to integers $\mathbb{Z}$. Hence we can use integer operations to apply a set of *ResourceAllocation* to a *ResourceValue* (*allocate*). We also specify the reverse operation *deAllocate* to free non-allocated resource items.

$$ResourceValue : \mathbb{P}(Resource \to (Item \nrightarrow \mathbb{N}))$$
[each resource carries a set of countable items]

$$ResourceAllocation : \mathbb{P}(Resource \to (Item \nrightarrow \mathbb{Z}))$$
[consumed or produced amount of items per resource]

$$allocate, deAllocate : ResourceValue \to \mathbb{P}(ResourceAllocation) \to ResourceValue$$
[applying allocations to a valuation]

$$\forall r : Resource;\ i : Item \bullet \qquad \text{[resource operations are]}$$
$$(\forall rv : ResourceValue;\ pra : \mathbb{P}\, ResourceAllocation \bullet \qquad \text{[derived from integers]}$$
$$allocate(rv)(pra)(r)(i) = (rv(r)(i) - \sum(\{ra : pra \bullet ra(r)(i)\})) \land$$
$$deAllocate(rv)(pra)(r)(i) = (rv(r)(i) + \sum(\{ra : pra \bullet ra(r)(i)\})))$$

The utility of a process is the difference between its current performance and the cost of resources (*ResourceCost*) that it currently applies for. *ResourceCost* depends on the amount of items that are needed by the process. It also depends the amount of items that are globally available, for example, a certain amount of stamina is more expensive, if there are not many units left. If an resource is exhausted, i.e., the available amount of items is less than allocated, resource costs are maximal (1.0).

$$ResourceCost : \mathbb{P}(Resource \rightarrow (Item \nrightarrow \mathbb{N}) \rightarrow (Item \nrightarrow \mathbb{Z}) \rightarrow UtilityScale) \qquad \text{[allocation cost relative to valuation]}$$

$$\forall r : Resource;\ i : Item \bullet (\forall rc : ResourceCost;\ rn : (Item \nrightarrow \mathbb{N});\ rz : (Item \nrightarrow \mathbb{Z}) \bullet rn(i) - rz(i) < 0 \bullet rc(r)(rn)(rz) = 1.0) \qquad \text{[exhausting is invaluable]}$$

In the following, let *Id* be the subset of *Identifier* that is reserved for internal addressing, i.e., for identifying processes and layers. The enumeration type *ProcessType* (*knowledge_base* | *reflex* | *behaviour_execution* | ...) determines which inferences are to be performed by a respective process.

$$Id == Identifier \setminus \{perception, action\} \qquad \text{[internal id's]}$$

$$ProcessType ::= knowledge\_base \mid reflex \mid behaviour\_execution \mid local\_planning \mid plan\_execution \mid social\_planning \mid protocol\_execution \qquad \text{[process types]}$$

Now we are ready to construct the *Process* schema. Its *state* consists of a set of options. Its *active* flag is a boolean $\mathbb{B}$ that indicates whether the process is suspended or active. The local resources *resourceValue* describe the amount of resource items that have already been granted to the process. The local profile, i.e., the average performance of the process, is maintained within its*performance* slot. Moreover, a process has two signal buffers, a sequence of input signals and a set of output signals, whose addressing must be consistent. For example, action and perception signals are only allowed to appear in the respective buffers of a process of type *behaviour_execution*.

Process

$$layer, id : Id \qquad \text{[id of layer, own id]}$$
$$type : ProcessType \qquad \text{[the type of process]}$$
$$state : \mathbb{P}\, Option \qquad \text{[inference options]}$$
$$active : \mathbb{B} \qquad \text{[ready to compute?]}$$
$$resourceValue : ResourceValue \qquad \text{[local resources]}$$
$$performance : UtilityScale \qquad \text{[average performance]}$$
$$signals\_in : \text{seq}\, Signal \qquad \text{[a signal queue]}$$
$$signals\_out : \mathbb{P}\, Signal \qquad \text{[a signal buffer]}$$

$$\forall s : \text{ran}\, signals\_in \bullet s.recipient = (layer, id) \wedge s \notin ActionSignal \qquad \text{[constraints]}$$
$$\forall s : signals\_out \bullet s.sender = (layer, id) \wedge s \notin PerceptionSignal \qquad \text{[on]}$$
$$(\exists s : Signal \bullet s \in (ActionSignal \cap signals\_out) \cup (PerceptionSignal \cap \text{ran}\, signals\_in)) \Rightarrow type = behaviour\_execution \qquad \text{[addressing]}$$

A newly created *InitProcess* has empty signal buffers, has no local resources, consists of a single *InitOption*, and has a neutral performance estimation set to 0.7.

InitProcess

Process	[a particular process]

$$signals\_in = \langle\rangle \land signals\_out = \varnothing \land (\exists\, b : InitOption \bullet state = \{b\})$$
$$resourceValue \in Resource \rightarrow (Item \nrightarrow \{0\}) \land performance = 0.7$$

In COOP, we identify *Layer*s of InteRRaP-R with their associated control process. Hence, each *Layer* controls a set of *processes* with unique identifiers under which we find exactly one of type *knowledge_base*. Since control processes survey the process communication, they also maintain buffers in which they collect incoming and outgoing signals from and to the process under control: *input* contains signals which come from a different layer or from outside the agent and which are addressed to an internal process. *output* contains signals which have been emitted internally and are to be sent into other layers or the outside of the agent. *signalBuffer* stores every communication that is addressed to the inside. Finally, *triggerStore* is able to temporarily store the trigger signals that are dedicated to already active processes. In the following, the auxiliary projection functions π_i^n with $i \leq n$ are used to extract particular values out of n-ary pairs.

Layer

$processes : \mathbb{P}\, Process;\ kb\_process : Process$	[all subordinate processes]
$id : Id$	[own id]
$signalBuffer, triggerStore, input, output : \mathbb{P}\, Signal$	[signal buffers]
$resourceValue : ResourceValue$	[global resources]
$costs : ResourceCost$	[cost specification]
$allocations : Id \rightarrow ResourceAllocation$	[allocations to processes]
$processUtility : Id \rightarrow UtilityScale$	[current utility of process]
$processEmphasis, processDiscount : Id \rightarrow UtilityScale$	[priority]
$discountFactor : Id \rightarrow \{r : \mathbb{R} \mid r > 0.0 \bullet r\}$	[curiosity]

$$kb\_process \in processes \land kb\_process.type = knowledge\_base \quad \text{[constraints]}$$
$$\forall\, p : processes \bullet p.layer = id \land (\forall\, p_2 : processes \bullet p.id = p_2.id \Rightarrow p = p_2)$$
$$signalBuffer = \bigcup \{p : processes \bullet \quad \text{[route signals]}$$
$$\{s : p.signals\_out \mid (\pi_1^2\ s.recipient) = id \bullet s\}\} \cup input \cup triggerStore$$
$$output = \bigcup \{p : processes \bullet \{s : p.signals\_out \mid (\pi_1^2\ s.recipient) \neq id \bullet s\}\}$$

The above defined *Layer* also has a global *resourceValue* depot from which static *allocations* can be given to the controlled processes. *Layer* depends on a *costs* specification, the measured utility of processes in the global profile (*processUtility*), the process emphasis (*processEmphasis*), and the process discount (*processDiscount*). *discountFactor* determines for each process the degree of penalising its frequent activation, i.e., how curious the agent will be in trying to activate other, supposedly worse computations.
The complete InteRRaP-R agent consists of three uniquely identified layers which disjointly encapsulate all internal processes. The types of processes are assigned according to the architecture, i.e., at the BBL, there is a *reflex* process and several *behaviour_execution* processes; at the LPL, there is a *local_planning* process and several *plan_execution* processes, etc. Signals that are transmitted between the layers are collected and distributed through *interLayerSignals*. Two additional sets, the

perceptionBuffer and the *actionBuffer*, handle the interaction with the external environment during the agent's processing.

InteRRaP − R

$$
\begin{array}{ll}
bbl, lpl, spl : Layer;\ processes : \mathbb{P}\, Process & \text{[three layers; all processes]} \\
interLayerSignals : \mathbb{P}\, Signal & \text{[signals between layers]} \\
perceptionBuffer : \mathbb{P}\, PerceptionSignal & \text{[perception signals are buffered]} \\
actionBuffer : \mathbb{P}\, ActionSignal & \text{[action signals are buffered]} \\
\hline
bbl.id \neq lpl.id \land bbl.id \neq spl.id \land lpl.id \neq spl.id & \text{[unique layers]} \\
\exists_1 p : bbl.processes \setminus \{bbl.kb\_process\} \bullet p.type = reflex \land & \text{[determ. modules]} \\
\quad (\forall p_2 : (bbl.processes \setminus \{bbl.kb\_process, p\}) \bullet p_2.type = behaviour\_execution) & \\
\exists_1 p : (lpl.processes \setminus \{lpl.kb\_process\}) \bullet p.type = local\_planning \land & \\
\quad (\forall p_2 : (lpl.processes \setminus \{lpl.kb\_process, p\}) \bullet p_2.type = plan\_execution) & \\
\exists_1 p : (spl.processes \setminus \{spl.kb\_process\}) \bullet p.type = social\_planning \land & \\
\quad (\forall p_2 : (spl.processes \setminus \{spl.kb\_process, p\}) \bullet p_2.type = protocol\_execution) & \\
processes = bbl.processes \cup lpl.processes \cup spl.processes & \text{[collect procs]} \\
interLayerSignals = bbl.output \cup lpl.output \cup spl.output \cup perceptionBuffer & \\
\forall l : \{bbl, spl, bbl\} \bullet & \text{[route signals]} \\
\quad l.input = \{s : interLayerSignals \mid \pi_1^2 s.recipient = l.id \bullet s\} & \\
actionBuffer = interLayerSignals \cap ActionSignal &
\end{array}
$$

3.4.2 The Operation of InteRRaP-R

We have now incrementally constructed the compound state of InteRRaP-R. Straightforwardly, specifying its overall operation will be composed by more primitive transitions upon its substructures, starting at the process level. For this purpose, we instantiate the general transition $\Delta Process$ to a more specific $\Delta ProcessOperation$ schema in which some parts of the process state stay constant, such as identifiers and type. Other parts are subject to change, such as signal buffers, local resources, local profile and the activation flag; for the moment, we do not place any restrictions on their evolution.

ΔProcessOperation

$$
\begin{array}{ll}
\Delta Process & \text{[a process transition]} \\
optionMap : Option \rightarrowtail \mathbb{P}\, Option & \text{[several option transitions]} \\
process, process' : Process;\ context : Process \rightarrowtail Process & \text{[the process context]} \\
\hline
layer' = layer \land id' = id \land type' = type & \text{[stable parts]} \\
\mathrm{dom}\, optionMap = state \land state' = \bigcup(\mathrm{ran}\, optionMap) & \text{[do option transition]} \\
\forall b_1, b_2 : Option \bullet optionMap(b_1) \cap optionMap(b_2) \neq \varnothing \Rightarrow b_1 = b_2 & \text{[unique]} \\
process = \theta Process \land process' = \theta Process' \land context(process) = process' &
\end{array}
$$

During $\Delta ProcessOperation$, the available inference options are rewritten according to the concept of search procedures underlying COOP: The *optionMap*, a partial injective function, allows each option to evolve into a unique set of successors. Moreover, due to shared memory, each process operation happens in the light of operations on other processes. These are simultaneously collected in the partial *context* mapping.

The additional $process, process'$ slots are used to make the process under consideration explicit and are bound by the dedicated θ operator of Z.

A *Layer* evolves in a $\Delta LayerOperation$ as all of its embedded processes evolve in a corresponding $\Delta ProcessOperation$. As before, the shared *context* is a partial injective function which allows processes to have no successor, i.e., they have died because of containing no more options. During $\Delta LayerOperation$, a set of new processes *newProcesses* of type *InitProcess* can be created.

$\Delta LayerOperation$

$\Delta Layer$	[a layer transition]
$newProcesses : \mathbb{P}\, InitProcess$	[newly activated processes]
$layer, layer' : Layer$	[the layer under consideration]
$context : Process \rightarrowtail\!\!\!\!\!\!+\; Process$	[process context]
$\forall p : (processes \cup newProcesses) \cap \operatorname{dom} context \bullet$	[process transition]
$\quad (\exists op : \Delta ProcessOperation \bullet op.process = p \wedge op.context = context)$	
$\forall p : (processes \cup newProcesses) \setminus \operatorname{dom} context \bullet p.state = \varnothing$	
	[failed processes]
$processes' = context(\!| processes \cup newProcesses |\!)$	[do transitions]
$kb\_process' = context(kb\_process) \wedge id' = id \wedge costs' = costs$	[these remain]
$layer = \theta Layer \wedge layer' = \theta Layer'$	[stable]

At the topmost level, the InteRRaP-R agent operates ($\Delta InteRRaP - ROperation$) by letting all its layers perform a $\Delta LayerOperation$. Hereby, the agent interfaces the environment via input $perception?$ and output $action!$ that are modelled as sets of signals. Input signals are added to the perception buffer; output signals are taken out of the action buffer. The overall *context* is defined in terms of the particular *context* mappings of the layers.

$\Delta InteRRaP - ROperation$

$\Delta InteRRaP - R;\ newProcesses : \mathbb{P}\, InitProcess$	[an InteRRaP-R transition]
$perception?, action! : \mathbb{P}\, Signal$	[perception and action signals]
$context : Process \rightarrowtail\!\!\!\!\!\!+\; Process$	[process transitions]
$actionBuffer' = \varnothing \wedge action! = actionBuffer$	[clear action buffer]
$perceptionBuffer' = perceptionBuffer \cup perception?$	[fill perception buffer]
$\exists blo, llo, slo : \Delta LayerOperation \bullet$	[layer operations]
$\quad blo.context = context \wedge blo.layer = bbl \wedge blo.layer' = bbl' \wedge$	
$\quad llo.context = context \wedge llo.layer = lpl \wedge llo.layer' = lpl' \wedge$	
$\quad slo.context = context \wedge slo.layer = spl \wedge slo.layer' = spl' \wedge$	
$\quad newProcesses = blo.newProcesses \cup llo.newProcesses \cup slo.newProcesses$	
$\operatorname{dom} context \subseteq processes \cup newProcesses$	

Inferences and Signalling We now make these quite general operations more concrete. For example, the 'regular' way of computation within a process is to perform a step of inference. For that purpose, we presume a common inference framework $\vdash$ that is defined upon *Formula*: It takes a conjunctive program and a conjunctive goal as its input and produces a set of optional program/goal pairs as its output.

$\vdash$: $\operatorname{ran} conj \times \operatorname{ran} conj \leftrightarrow \mathbb{P}(\operatorname{ran} conj \times \operatorname{ran} conj)$ [step of inference]

COOP processes implement particular subsets of this inference framework by means of the $\rightarrowtail$ relation. For each *ProcessType*, $\rightarrowtail$ describes a transition upon one selected inference option of the process. $\rightarrowtail$ inputs a pair of $\mathbb{P}$ *Conjunct* consisting of a selected portion of the options' program and a selected portion of the option's goal. $\rightarrowtail$ manipulates the top of the option's exception stack and consumes a characteristic amount of resources specified by a *ResourceAllocation*. In turn, it produces some performance value and a set of program/goal/exceptions/signals tuples from which the successor options and outgoing signals are generated. The important constraint to this construction is that the transitions given by $\rightarrowtail$ are indeed according to $\vdash$ and, moreover, that produced signals transmit a part of the information of their associated option. In Section 5, we will describe in detail how $\vdash$ and $\rightarrowtail$ are to be further refined. In the following, $conj^{\sim}$ denotes the inverse relation to *conj*.

$\rightarrowtail$: *ProcessType* $\rightarrow$ [step of process is subset of inference+interface]
$(\mathbb{P}\, Conjunct \times \mathbb{P}\, Conjunct \times \operatorname{seq} InferenceState \times ResourceAllocation) \leftrightarrow$
$(UtilityScale \times \mathbb{P}(\operatorname{ran} conj \times \operatorname{ran} conj \times \operatorname{seq} InferenceState \times \mathbb{P}\, Signal))$

$\forall pt : ProcessType;\ cs_1, cs_2 : \mathbb{P}\, Conjunct;\ ra : ResourceAllocation;$
$ex, ex' : \operatorname{seq} InferenceState;\ f_1, f_2 : \operatorname{ran} conj;\ ss : \mathbb{P}\, Signal \bullet$
$(f_1, f_2, ex', ss) \in \pi_2^2(\rightarrowtail(pt)(cs_1, cs_2, ex, ra)) \Rightarrow$
$(((f_1, f_2) \in \vdash (translateConjunct(cs_1), translateConjunct(cs_2))) \wedge$ [infer]
$(\forall s : ss \bullet conj^{\sim}(s.program) \subseteq conj^{\sim}(f_1) \wedge conj^{\sim}(s.goal) \subseteq conj^{\sim}(f_2)) \wedge$
$((\exists e : InferenceState \bullet ex' = \langle e \rangle \frown ex) \vee ex' = \langle\rangle))$ [push or pop]

$\rightarrowtail$ is embedded into the $\Delta ProcessInf$ schema which is an instance of $\Delta ProcessOperation$. First, an option inside the process state is selected. Within that option, we identify selected parts of the program and the goal. The amount of resources that is required to perform an inference step is allocated and the head of the exception stack is separated (by the auxiliary *exhead* and *extail*). Then, the successor options registered in *optionMap* are constructed from running $\rightarrowtail(type)$: Each element of the answer set is combined with the non-selected conjuncts of the selected predecessor option. Signals of the selected conjuncts are merged and cloned. Outgoing signals are added to the annotation of the respective conjuncts. And the produced performance is averaged with the previous profile such that the influence of past computations diminishes with time.

$\Delta ProcessInf$

$\Delta ProcessOperation$; $slctOpt : Option$; $slctPrem, slctGoal : \mathbb{P}\, Conjunct$

$derivedOptions : \mathbb{P}\, Option$ [new options inferred from selection]

$derivedSignals, mrgSigs : \mathbb{P}\, Signal$ [relevant (new) signals]

$derived : \mathbb{P}(\operatorname{ran} conj \times \operatorname{ran} conj \times \operatorname{seq} InferenceState \times \mathbb{P}\, Signal)$ [raw result]

$derivedPerformance : UtilityScale$ [produce some performance]

$consume : ResourceAllocation$ [consume some resources]

$optionMap = \{b : state \bullet b \mapsto \{b\}\} \oplus \{slctOpt \mapsto derivedOptions\}$ [transform]

$slctPrem \subseteq slctOpt.program \land slctGoal \subseteq slctOpt.goal$ [select]

$resourceValue' = allocate(resourceValue)(\{consume\})$ [allocate]

$mrgSignals = \bigcup\{c : slctPrem \cup slctGoal \bullet c.annotation\}$

$(derivedPerformance, derived) = \vdash(type)(slctPrem,$ [infer]
$slctGoal, exhead(slctOpt), consume)$

$derivedOptions = \{b : Option;\ fs_1, fs_2 : \mathbb{P}\, Formula;\ sex : \operatorname{seq} InferenceState;$
$ss_1, ss_2 : \mathbb{P}\, Signal \mid (conj(fs_1), conj(fs_2), sex, ss_1) \in derived \land$

$b.exceptions = sex \frown extail(slctOpt) \land$ [build excpt. stack]

$ss_2 = \{s : mrgSignals \bullet (s.clone)(b)\} \land$ [clone orig sigs]

$(b.program = \{c : Conjunct \mid c.state \in fs_1 \land c.annotation =$ [build]
$(ss_2 \cup \{s : ss_1 \mid c.state \in conj^{\sim}(s.program) \bullet s\}) \bullet c\}$ [program]

$\cup\{c, c' : Conjunct \mid c \in slctOpt.program \setminus slctPrem \land$
$c'.state = c.state \land c'.annotation = clone(\!|\, c.annotation \,|\!) \cap mrgSignals \cup$
$c.annotation \setminus clone^{\sim}(mrgSignals) \bullet c'\}) \land$

$(b.goal = \{c : Conjunct \mid c.state \in fs_2 \land c.annotation =$ [build goal]
$(ss_2 \cup \{s : ss_1 \mid c.state \in conj^{\sim}(s.goal) \bullet s\}) \bullet c\}$

$\cup\{c, c' : Conjunct \mid c \in slctOpt.goal \setminus slctGoal \land$
$c'.state = c.state \land c'.annotation = clone(\!|\, c.annotation \,|\!) \cap mrgSignals \cup$
$c.annotation \setminus clone^{\sim}(mrgSignals) \bullet c'\}) \bullet b\}$

$signals\_in' = signals\_in \land derivedSignals = \bigcup\{ffe : derived \bullet \pi_1^4(ffe)\}$

$signals\_out' = signals\_out \cup derivedSignals$

$performance' = ((performance + derivedPerformance)/2.0)$ [average perf.]

The selection of options, conjuncts, and inferences is not further specified here, but crucial to the decent behaviour of processes. This issue is closely linked to the embedded logic and the applied inference principle and will be discussed at length in Section 5. We go on by describing the behaviour of the *active* flag that is set if and only if there exists the possibility to compute, i.e., the respective process is able to perform some $\Delta ProcessInf$ under the currently available local resources.

$\forall p : Process \bullet p.active = 1 \Leftrightarrow (\exists pi : \Delta ProcessInf \bullet p = pi.process)$ [set flag]

Further process operations can be distinguished. A $\Delta ProcessExtension$ is any process operation which does neither change the profile nor consumes or produces any resources. An instance thereof is the $\Delta ProcessStable$ schema which, in addition, leaves the inference options untouched.

$\Delta ProcessExtension \mathrel{\widehat=} [\Delta ProcessOperation \mid performance' = performance \land$
$resourceValue' = resourceValue]$

$\Delta ProcessStable \mathrel{\widehat=} [\Delta ProcessExtension \mid optionMap = \{b : state \bullet b \mapsto \{b\}\}]$

Incorporating an incoming signal into the state of a recipient process (see Figure 16) is now defined as follows: The first signal in the input queue is read and the sending process is looked up from the *context*. The content of the signal, i.e., that part of the sender's current state which is annotated with a clone of the signal, is incorporated into every option of the recipient. For this purpose, the respective logic goal and a subset of the logic program from some of the relevant options is taken. During incorporation, no outgoing signals are generated. The auxiliary *annotation* relates options to the annotations in any of their conjuncts.

$$
\begin{array}{|l}
\Delta ProcessSignal \\
\hline
\Delta ProcessExtension \qquad \text{[an extension]} \\
firstSignal : Signal;\ sender : Process \qquad \text{[the processed signal/the sender]} \\
\hline
firstSignal = head(signals\_in) \land signals\_in' = tail(signals\_in) \qquad \text{[read first sig]} \\
sender \in \mathrm{dom}\, context \land (sender.layer, sender.id) = firstSignal.sender \qquad \text{[lookup]} \\
optionMap = \{b : state;\ bs : \mathbb{P}\, Option \mid \forall\, b' : bs \bullet \qquad \text{[incorporate]} \\
\quad \exists\, b'' : sender.state;\ s : annotation(\!|\ \{b''\}\ |\!) \cap clone \star (\!|\ \{firstSignal\}\ |\!); \\
\quad ps : \mathbb{P}\{c : b''.program \mid s \in c.annotation \bullet c\} \bullet \\
\qquad b'.exceptions = b.exceptions \land b'.program = b.program \cup ps \land \\
\qquad b'.goal = b.goal \cup \{c : b''.goal \mid s \in c.annotation \bullet c\}) \\
\quad \bullet\ b \mapsto bs\} \land signals\_out' = signals\_out \\
\hline
\end{array}
$$

We collect $\Delta ProcessInf$ and $\Delta ProcessSignal$ in the general schema $\Delta ProcessCompute$.

$$\Delta ProcessComputation \mathrel{\widehat=} \Delta ProcessInf \lor \Delta ProcessSignal$$

Shared Memory and Exception Handling Signal incorporation is just one side of process communication in COOP. Actually, any process, which has a shared reference with another process that performs a $\Delta ProcessComputation$, could itself perform a $\Delta ProcessExtension$ (see Figure 17). In more detail, as a by-product of a single process' computation, the correspondingly annotated options in every other process are extended. In the following $\Delta OptionExtension$, we describe this effect of shared memory onto a single option which follows similar considerations as $\Delta ProcessSignal$. It is used to establish the shared memory model afterwards.

$\Delta OptionExtension$

$\Delta Option;\ option, option' : Option;\ allSigs : \mathbb{P}\, Signal$ [an option transition]

$po : \Delta ProcessOperation$ [caused by this operation]

$propagateProgram, propagateGoal : \mathbb{P}\, Conjunct$ [propagated info]

$option = \theta Option$

$option' = \theta Option'$

$allSigs = annotation(\!|\ \{option\}\ |\!)$

$(annotation(\!|\ po.process.state\ |\!) \cap allSigs = \varnothing \land$

$\quad annotation(\!|\ po.process'.state\ |\!) \cap clone(\!|\ allSigs\ |\!) = \varnothing \Rightarrow option' = option)$

$(\exists\, b : po.process.state;\ b' : po.optionMap(b) \bullet$

$\quad (annotation(\!|\ \{b\}\ |\!) \cap allSigs \neq \varnothing \lor annotation(\!|\ \{b'\}\ |\!) \cap clone(\!|\ allSigs\ |\!) \neq \varnothing)$

$\quad \Rightarrow exceptions' = exceptions \land$ [exceptions are stable]

$\quad\quad (propagateProgram \subseteq \{c : b'.program \mid$

$\quad\quad\quad c.annotation \cap clone(\!|\ allSigs\ |\!) \neq \varnothing \bullet c\}) \land$

$\quad\quad (propagateGoal = \{c : b'.goal \mid$

$\quad\quad\quad c.annotation \cap clone(\!|\ allSigs\ |\!) \neq \varnothing \bullet c\}) \land$

$\quad\quad (program' = \{c : program;\ c' : Conjunct \mid c'.state = c.state \land$

$\quad\quad\quad c'.annotation = c.annotation \setminus clone^{\sim}(\!|\ annotation(\!|\ \{b'\}\ |\!)\ |\!) \cup$

$\quad\quad\quad\quad clone(\!|\ c.annotation\ |\!) \cap annotation(\!|\ \{b'\}\ |\!) \bullet c'\} \cup$

$\quad\quad propagateProgram) \land$

$\quad\quad (goal' = \{c : goal \mid clone(\!|\ c.annotation\ |\!) \cap$

$\quad\quad\quad annotation(\!|\ \{b'\}\ |\!) = \varnothing\} \cup$

$\quad\quad propagateGoal))$

The last mechanism that is left to specify on the process level is exception handling. Exception handling 'shields' other process operations, such as a computation and a shared memory operation. Thus, the $\Delta ExceptionHandling$ schema refers to a shielded operation po. Exception handling does only become active upon operations in which some option fails, i.e., it is mapped to an empty set of successors. Failures are restored by retrieving the topmost alternative option from the exception stack. If the exception stack is empty, a recovery is not possible and the process possibly dies.

$\Delta ExceptionHandling$

$\Delta ProcessExtension$ [a process extension]

$po : \Delta ProcessExtension \cup \Delta ProcessComputation$ [embed]

$po.process = process$

$signals\_in' = po.signals\_in'$ [signals stable]

$signals\_out' = po.signals\_out'$

$\forall b : state \bullet$

$((po.optionMap(b) \neq \varnothing \land optionMap(b) = po.optionMap(b)) \lor$ [no excp.]

$(po.optionMap(b) = \varnothing \land b.exceptions = \langle\rangle \land optionMap(b) = \varnothing) \lor$ [failed]

$(po.optionMap(b) = \varnothing \land (\exists e : InferenceState;\ b' : Option \bullet$ [raised excp.]

$optionMap(b) = \{b'\} \land$ [restore option]

$head(b.exceptions) = e \land$ [pop exception]

$b'.program = e.program \land$ [build option]

$b'.goal = e.goal \land$

$b'.exceptions = extail(b))))$

Ordinary processes and control processes compute asynchronously. Therefore, major parts of *Layer*, such as global resources, global profile, allocations, and priorities, do not change during the operation of an internal process. This is expressed in the following $\Delta LayerExtension$ schema.

$\Delta LayerExtension$

$\Delta LayerOperation$ [a layer operation]

$resourceValue' = resourceValue$ [much]

$allocations' = allocations$ [stays]

$processEmphasis' = processEmphasis$ [stable]

$processUtility' = processUtility$

$triggerStore' = triggerStore$

$processDiscount' = processDiscount$

$discountFactor' = discountFactor$

$newProcesses = \varnothing$

$\forall p : processes \bullet ((context(p)).signals\_in) = (p.signals\_in)$

InteRRaP-R performs a computation $\Delta InteRRaP - RCompute$ as a single of its processes performs a computation *po*. As already anticipated, any other process is allowed to simultaneously perform a (possibly non-empty) $\Delta OptionExtension$ to any of its options according to the shared memory model. Both *po* and shared memory extensions are shielded by exception handling.

$\Delta InteRRaP - RCompute$

$\Delta InteRRaP - ROperation;\ po : \Delta ExceptionHandling;\ process, process' : Process$

$\exists blc, llc, slc : \Delta LayerExtension \bullet blc.layer = bbl \land blc.layer' = bbl' \land$
$llc.layer = lpl \land llc.layer' = lpl' \land slc.layer = spl \land slc.layer' = spl'$
$po.po \in \Delta ProcessComputation \land process \in processes \land po.process = process$
$po.process' = process' \land context(process) = process'$
$\forall p, p' : Process \bullet (p' = context(p) \Leftrightarrow$
$(\exists po' : \Delta ExceptionHandling \bullet (po'.po \in \Delta ProcessExtension) \land$
$po'.process = p \land po'.process' = p' \land p'.signals\_out = p.signals\_out \land$
[each process performs an extension shielded by exception handling]
$(\forall b : p.state;\ bs : \mathbb{P}\, Option \bullet$
$bs = \{be : \Delta OptionExtension \mid be.po = po'.po \bullet be.option'\} \land$
$po'.po.optionMap(b) \subseteq bs \land$
$(po'.po.optionMap(b) = \varnothing \Leftrightarrow bs = \varnothing))))$ [if possible]

Besides the *InteRRaP – RCompute* case in which a single, regular process is the actual source of computation, we also have the following $\Delta InteRRaP - RControl$ case in which a single layer or control process performs a control operation. The referred $\Delta LayerControl$ schema will be specified in the following paragraph. Since control processes are not only interleaved with respect to their internal processes, but also with respect to one another, $\Delta LayerControl$ requires all other layers to remain virtually untouched ($\Delta LayerStable$).

$\Delta LayerStable \mathrel{\widehat=} [\Delta LayerExtension \mid \forall p : processes \bullet$ [processes stable]
$\exists op : \Delta ProcessStable \bullet op.process = p \land op.process' = context(p)]$

$\Delta InteRRaP - RControl$

$\Delta InteRRaP - ROperation$

$(\exists lc : \Delta LayerControl;\ transition : \{(bbl, bbl'), (spl, spl'), (lpl, lpl')\} \bullet$
$lc.layer = \pi_1^2 transition \land$
$lc.layer' = \pi_2^2 transition \land$
$lc.context = context \land$
$(\forall transition_2 : \{(bbl, bbl'), (spl, spl'), (lpl, lpl')\} \setminus \{transition\} \bullet$
$(\exists ls : \Delta LayerStable \bullet$
$ls.layer = \pi_1^2 transition_2 \land$
$ls.layer' = \pi_2^2 transition_2 \land$
$ls.context = context)))$

The overall COOP model is collected in the following *COOP* schema. At this point, we informally state that *COOP* should be fair, i.e., that it eventually performs pending control, signalling, and ordinary inferences. A formal treatment of fairness would amount to a lengthy notation in the presented framework; we refer to Section 5 where we present a possibility for establishing fairness at the inference level in mathematical terms.

$COOP \mathrel{\widehat=} \Delta InteRRaP - RCompute \lor \Delta InteRRaP - RControl$

Control The final part of our specification describes the already referenced control operations of InteRRaP-R layers, i.e., control processes. Two functions build the interface of a control process to its associated knowledge base process. *readKnowledge* is used to obtain the current resource parameters, such as allocations, emphasis, and discount factors from state of the knowledge base. *writeKnowledge* is used to store global resources and global profile in the state of the knowledge base for further processing.

$$readKnowledge : \mathbb{P}\, Option \rightarrow ((Id \rightarrow ResourceAllocation) \times (Id \rightarrow UtilityScale) \times (Id \rightarrow \mathbb{R})) \quad \text{[read] [parameters from kb]}$$

$$writeKnowledge : (\mathbb{P}\, Option \times ResourceValue \times (Id \rightarrow UtilityScale)) \rightarrow \mathbb{P}\, Option \quad \text{[write] [parameters into kb]}$$

Next, we give the equation to compute the process *utility* under a given *ResourceValue*, an applied *ResourceAllocation*, a cost specification *ResourceCost*, the process performance, the process emphasis, and the process discount.

$$utility : (Process \times (Id \rightarrow UtilityScale) \times (Id \rightarrow UtilityScale) \times ResourceAllocation \times ResourceValue \times ResourceCost) \rightarrow UtilityScale$$

$$\forall\, p : Process;\ emph, disc : Id \rightarrow UtilityScale;\ ra : ResourceAllocation;\ rv : ResourceValue;\ rc : ResourceCost \bullet \quad \text{[utility is]}$$

$$utility(p, emph, disc, ra, rv, rc) = \quad \text{[performance-resource cost]}$$

$$min\{max\{(disc(p.id) * (emph(p.id) * p.performance - \sum(\{r : Resource \bullet rc(r)(ra(r))(rv(r))\}))), 0.0\}, 1.0\}$$

The recursive *activate* function realises the already mentioned filter that decides which process is to be resumed from a set of candidate processes. Until the candidate set is empty, *activate* chooses the process with the highest utility. If its utility is greater than zero, the process is accepted for activation and is allocated the applied resources. If its utility is equal to zero, the process is filtered out. In each case, the gathered utility values are registered within the global profile *pu*.

$$
\begin{array}{l}
activate : (\mathbb{P}\, Process \times (Id \rightarrow UtilityScale) \times (Id \rightarrow UtilityScale) \times \\
\quad (Id \rightarrow ResourceAllocation) \times ResourceValue \times ResourceCost) \leftrightarrow \\
\quad (\mathbb{P}\, Process \times ResourceValue \times (Id \rightarrow UtilityScale)) \\
\hline
\forall\, ps_1, ps_2 : \mathbb{P}\, Process;\ emph, disc, pu : (Id \rightarrow UtilityScale);\ rc : ResourceCost; \\
\quad pa : (Id \rightarrow ResourceAllocation);\ rv_1, rv_2 : ResourceValue \bullet \\
\quad (ps_2, rv_2, pu) = activate(ps_1, emph, disc, pa, rv_1, rc) \Leftrightarrow \qquad \text{[activation filter]} \\
\qquad (ps_1 = \varnothing \land ps_2 = \varnothing \land rv_2 = rv_1 \land pu = emph) \lor \qquad \text{[nothing to do]} \\
\qquad (\exists\, p : ps_1 \bullet \forall\, p_2 : ps_1 \bullet \qquad \text{[choose best utility]} \\
\qquad\quad (utility(p, emph, disc, pa(p.id), rv_1, rc) \geq \\
\qquad\qquad utility(p_2, emph, disc, pa(p_2.id), rv_1, rc)) \land \\
\qquad\quad (utility(p, emph, disc, pa(p.id), rv_1, rc) = 0.0 \Rightarrow \qquad \text{[zero - no activation]} \\
\qquad\qquad (\exists\, pp : (Id \rightarrow UtilityScale) \bullet \\
\qquad\qquad\quad (activate(ps_1 \setminus \{p\}, emph, disc, pa, rv_1, rc) = (ps_2, rv_2, pp) \land \\
\qquad\qquad\quad pu = pp \oplus \{p.id \mapsto utility(p, emph, pa(p.id), rv_1, rc)\}))) \land \\
\qquad\quad (utility(p, emph, disc, pa(p.id), rv_1, rc) > 0.0 \Rightarrow \\
\qquad\qquad\qquad\qquad \text{[positive - activate \& allocate]} \\
\qquad\qquad (\exists\, pp : (Id \rightarrow UtilityScale) \bullet \\
\qquad\qquad\quad (activate(ps_1 \setminus \{p\}, emph, disc, pa, allocate(rv_1)(\{pa(p.id)\}), rc) = \\
\qquad\qquad\qquad (ps_2 \setminus \{p\}, rv_2, pp) \land \\
\qquad\qquad\quad pu = pp \oplus \{p.id \mapsto utility(p, emph, disc, pa(p.id), rv_1, rc)\})))))
\end{array}
$$

activate is used in the context of the final Δ*LayerControl* schema. Herein, a *Layer* clears and inspects its signal buffer by looking for trigger signals and the associated candidate processes to be possibly resumed: These are suspended processes whose *active* flag is down and new processes to be created.

After updating the resource parameters by accessing the knowledge base using *readKnowledge* and deallocating non-consumed resources from the suspended processes, the *activate* filter is called to decide about process activation, to compute the new global profile, and to determine the new global resources.

All processes which are granted to be resumed are equipped with their trigger signals and are allocated their applied resources, where *anySequence* is an auxiliary function which turns a set (of signals) into an arbitrary ordered sequence (of signals). The discount of resumed processes is increased, i.e., the scaling of their performance in the *utility* function diminishes.

All processes which could have been activated, but are suppressed because of a bad utility, just receive ordinary signals. Their discount is reset and their trigger signals are thrown away, except the activation signals of new processes. These activation signals and the trigger signals dedicated to already active processes are backed up in the *triggerStore*, where the $\lhd$ notation means the domain anti-restriction of a relation.

Finally, resource data and global profile are stored in the knowledge base via *writeKnowledge*; while all other processes stay stable, the knowledge base hence performs a Δ*ProcessExtension*.

$\Delta LayerControl$

$\Delta LayerOperation$ [a layer operation]

$NormalSignals, TriggerSignals : Process \to \mathbb{P}\, Signal$ [signal buffers]

$PossiblyActive, WillBeActive : \mathbb{P}\, Process$ [process sets]

$collectValue : ResourceValue$ [collect resource rests]

$\forall p : processes \cup newProcesses \setminus \{kb\_process\} \bullet$
$\quad \exists op : \Delta ProcessStable \bullet op.process = p \land op.context = context$

$\exists pe : \Delta ProcessExtension \bullet pe.process = kb\_process \land$
$\quad pe.process' = kb\_process'$

$(allocations', processEmphasis', discountFactor') =$ [read parameters]
$\quad readKnowledge(kb\_process.state)$

$NormalSignals = \{p : \mathrm{dom}\, context \bullet p \mapsto \{s : signalBuffer \setminus TriggerSignal \mid$
$\quad s.recipient = (id, p.id) \bullet s\}\}$ [normal signals]

$TriggerSignals = \{p : \mathrm{dom}\, context \bullet p \mapsto \{s : signalBuffer \cap TriggerSignal \mid$
$\quad s.recipient = (id, p.id) \bullet s\}\}$ [trigs]

$newProcesses = \mathrm{dom}\, TriggerSignals \setminus processes$ [new procs]

$PossiblyActive = \{p : \mathrm{dom}\, TriggerSignals \mid p.active = 0 \bullet p\}$ [susp]

$collectValue =$
$\quad deAllocate(resourceValue)(\{p : PossiblyActive \bullet p.resourceValue\})$

$(WillBeActive, resourceValue', processUtility') =$
$\quad activate(PossiblyActive, processEmphasis, allocations, collectValue, costs)$

$\forall p : (processes \cup newProcesses) \cap WillBeActive \bullet$
$\quad ((context(p)).signals\_in) =$
$\quad\quad anySequence(TriggerSignals(p))^\frown$
$\quad\quad anySequence(NormalSignals(p))^\frown p.signals\_in \land$
$\quad ((context(p)).resourceValue) = allocations(p.id) \land$
$\quad processDiscount'(p.id) =$
$\quad\quad min\{1.0, (processDiscount(p.id)/discountFactor(p.id))\}$

$\forall p : (processes \cup newProcesses) \setminus WillBeActive \bullet$
$\quad ((context(p)).signals\_in) = anySequence(NormalSignals(p))^\frown p.signals\_in \land$
$\quad (p \in PossiblyActive \Rightarrow$
$\quad\quad ((context(p)).resourceValue) \in Resource \to (Item \nrightarrow \{0\}) \land$
$\quad\quad processDiscount'(p.id) = 1.0) \land$
$\quad (p \notin PossiblyActive \Rightarrow ((context(p)).resourceValue) = p.resourceValue \land$
$\quad\quad processDiscount'(p.id) = processDiscount(p.id))$

$signalBuffer' = triggerStore'$ [empty the signal buffer]

$triggerStore' = \bigcup(\mathrm{ran}((PossiblyActive \setminus$
$\quad (newProcesses \setminus WillBeActive)) \lhd TriggerSignals))$

$kb\_process'.state = writeKnowledge(kb\_process.state, resourceValue',$
$\quad\quad processUtility')$

3.5 Bottom Line

The crucial role of formal specifications in software engineering and the importance of computational models for the design of intelligent agents in particular has been

$[Variable, Constant]$
$Term ::= Variable \mid Constant(Term^*)$
$Wff ::= \bot \mid Constant(Term^*) \mid \neg Wff \mid Wff \vee Wff \mid \exists\, Variable.Wff$

Figure 19: Syntax of First-Order Logic

discussed: Computational models based on a mathematical notation are a necessary supplement to architectures, theories, and inference systems within a coherent agent design methodology. Computational models are the key to bring operational agent features into a verifiable and unambiguous setting. For the purpose of presentation and analysis, the specification language Z is particularly well-suited.
The COOP model pins down the InteRRaP-R architecture in a type-checked and consistent Z specification. It makes use of programming constructs known from reactive system design [AZ87], modern programming languages [Smo95], fault-tolerant systems [BW96], and intelligent computation scheduling [RW91, BD94, Hor86]. COOP describes InteRRaP-R as a network of concurrent processes interacting via signals, shared memory, and exceptions. COOP realises abstract resources as quantitative representations of process interdependencies. Based on these representations, COOP applies an asynchronous control algorithm to optimise computation via optimising communication.
Our model differs from standard resource management such as [BD94, Hor86] in that it handles a set of simultaneous and independent process dependencies in real-time; to this end, internal profiling has been shown as a surprisingly simple and adequate tool. Our model differs from alternative agent specifications such as [dKLW98, dL98] in that it regards processes not as arbitrary forms of computation, but as continuous sub-inferences in a given logic. The data structures that are exchanged between processes are thus logic data structures, i.e., formulae. In contrast to the architecture-independent DESIRE [DKT94] in which components realise various types of logics, COOP provides a concrete and layered framework for expressing inferential services in a common logic of time and action. This shared logic theory will be developed in the following.

4 Theory: *HEC*

The automation of *symbolic logic* (or *computational logics*) [Fit90] is to develop formal languages for expressing *theories* in an intuitive (declarative) manner using a mathematical semantics and to provide the computational (operational) means to efficiently execute them on a computer. The computational means are *proof procedures*, which check the validity of statements in the logic language, and *inference procedures*, which provide more specialised services, such as the analysis of the consequences of a given theory.

4.1 First-Order Logic and Logic Programming

In the rich history of computational logic researched by philosophers, computer scientists, linguists, and researchers in AI, many formal languages and associated proof procedures have been developed and are still an active topic. The semantics of the most prominent *first-order logic* (Figure 19) is determined by particular mathematical structures which comprise the *models* of a formula (Definition 3). Syntactic sugar, such as the truth value $\top$ and the connectives $\forall$, $\wedge$, $\equiv$, $\subset$, and $\supset$, are expressible in terms of the existing constructs.

Definition 1 (First-Order Two-Valued Structure) *A first-order two-valued structure* $M = (U, IF, IP)$ *consists of a universe* U, *a functional interpretation*
$IF : Constant \times \mathbb{N} \to (U^* \to U)$, *and a relational interpretation*
$IP : Constant \times \mathbb{N} \to (\mathbb{P}\, U^* \to \{0,1\})$ *such that for all* $C : Constant$ *and* $n : \mathbb{N}$ *it holds* $IF(C,n) \in (U^n \to U)$ *and* $IP(C,n) \in (\mathbb{P}\, U^n \to \{0,1\})$

Definition 2 (Assignment and Mapping) *Let* $M = (U, IF, IP)$ *be a first-order two-valued structure. An assignment* $V : Variable \to U$ *is a total function from variables to the universe* U. *The mapping of terms under an assignment*
$\tilde{V} : Term \times (Variable \to U) \to U$ *is defined for all* $C : Constant$; $Va : Variable$; $n : \mathbb{N}$; $Te_1, \ldots, Te_n : Term$; *and* $V : Variable \to U$ *as follows:*

$$\tilde{V}(Va, V) = V(Va) \quad \text{and} \quad \tilde{V}(C(Te_1, \ldots, Te_n)) = IF(C,n)(\tilde{V}(Te_1, V), \ldots, \tilde{V}(Te_n, V))$$

Definition 3 (First-Order Two-Valued Model) *Let* $M = (U, IF, IP)$ *be a first-order two-valued structure. For a formula* $F : Wff$, M *is a first-order two-valued model,* $M \models_2 F$, *iff* F *is valid with respect to* M, *i.e., under all assignments* $V : Variable \to U$, *we have* $M, V \models_2^1 F$. *We define the two-valued valuation* $\models_2^{\{0,1\}}$ *for all*
$F, F_2, F_3 : Wff$; $C : Constant$; $n : \mathbb{N}$; $Va : Variable$; $Te_1, \ldots, Te_n : Term$; *and* $Bo, Bo_2, Bo_3 : \{0,1\}$ *as follows:*

$M, V \models_2^0 \bot$

$M, V \models_2^{Bo} C(Te_1, \ldots, Te_n)$ *iff* $Bo = IP(C,n)(\tilde{V}(Te_1, V), \ldots, \tilde{V}(Te_n, V))$

$M, V \models_2^{1-Bo} \neg F$ *iff* $M, V \models_2^{Bo} F$

$M, V \models_2^{max\{Bo_2, Bo_3\}} F_2 \vee F_3$ *iff* $M, V \models_2^{Bo_2} F_2$ *and* $M, V \models_2^{Bo_3} F_3$

$M, V \models_2^{Bo} \exists\, Va.F$ *iff* $B = max\{D : U;\ Bo_2 : \{0,1\} \mid M, V \oplus \{Va \mapsto D\} \models_2^{Bo_2} F \bullet B_2\}$

The attraction of 'logic programming' (LP) [Kow79] as an engineering method is not so much like telling the computer what to do when, but rather like telling what is true and asking to try and draw conclusions. To quote:

> "the advantages of logic programming should be that computer programs are easier to read. They should not be cluttered up with details about *how* things are to be done — they will be more like specifications of *what* a solution will look like. Moreover, if a program is rather like a specification of what it is supposed to achieve, it should be relatively easy, just by looking

$Literal ::= Constant(Term^*) \mid \neg Constant(Term^*)$
$Goal ::= \top \mid Literal \wedge Goal$
$Clause ::= \tilde{\forall} Constant(Term^*) \subset \tilde{\exists}\, Goal$
$Program ::= \top \mid Clause \wedge Program$

Figure 20: First-Order Normal Form

> at it (or, perhaps, by some automatic means), to check that it really does do what is required. In summary, the advantages of a logic programming language would result from programs having a *declarative* semantics as well as a *procedural* one."[CM87]

Hence, LP offers a means for *rapid prototyping*, i.e., to move directly from high-level specifications, say an axiomatisation of the the delivery problems in the Automated Loading Dock, to corresponding implementations. Particular LP languages, of which Prolog [Col85] is perhaps the best known, correspond to well-structured recursive programs (Figure 20). The usual service that is provided in such languages is to check for entailment of $P : Program;\ G : Goal$, i.e., whether $P \supset \tilde{\exists} G$ is valid in all first-order structures (short: $P \models \tilde{\exists} G$).
However, negation is usually not treated in the way that first-order logic postulates [Kun87]. This is because one wants to draw conclusions in the absence of particular pieces of information. For example in being ignorant about a particular box standing on the shelf, we would normally not be able to infer that there is some space left on the shelf. To this end, the underlying semantics is changed to a *minimal model semantics* applying a *closed-world assumption*: as long as we are not explicitly told that the shelf is occupied by a box, we assume its being free. Syntactically, this can be expressed by modifying P in such as way that everything about which we are nothing told in P, e.g., the occupancy of the box, must be false. This can be captured by the following transformation, the Clark completion [Cla78], which constructs a replacement theory for P. For simplification purposes, we can regard programs as sets of conjuncted clauses.

Definition 4 (Completion) *Let $Comp : Program \rightarrow Wff$ be the completion of logic programs. For all $P : Program$ it holds:*

$$Comp(P) ::= \bigwedge_{\mathcal{I}} \tilde{\forall} C(Va_1, \ldots, Va_n) \equiv \bigvee_{\mathcal{J}(C,n)} \tilde{\exists} Va_1 \doteq Te_1 \wedge \ldots \wedge Va_n \doteq Te_n \wedge G\}$$

where

$$\mathcal{I} ::= \{C : Constant \setminus \{\doteq\};\ n : \mathbb{N};\ Va_1, \ldots, Va_n : Variable \mid Va_i \text{ distinct and not in} P\}$$

and

$$\mathcal{J}(C, n) ::= \{Te_1, \ldots, Te_n : Term;\ G : Goal \mid \tilde{\forall} C(Te_1, \ldots, Te_n) \subset \tilde{\exists} G \in P\}$$

The completion maps each program to a formula which, by using equivalence $\equiv$ definitions, still contains all the implications of the original program; the equivalences

additionally state that the predicates cannot be derived in any other way. Every predicate not defined in P is assigned $\bot$ by the completion. The construction makes use of the binary relation $\doteq$ which should be interpreted as the equality of elements in the universe. A widely used equality theory in logic programming is the *Clark Equality Theory* (*CET*) which has been introduced in combination with the completion and is presented in Definition 5. We can now define the typical LP inference service as checking the validity of $Comp(P) \wedge CET \models \tilde{\exists} Goal$, i.e., whether the goal is valid in all 'minimal' models associated with P. Proposition 1 states the straightforward observation that each minimal model of P is also a model of P.

Definition 5 (CET) *Let $\mathcal{V} : Term \to \mathbb{P}\, Variable$ be the mapping of terms to contained variables. We define $CET ::= CET1 \wedge CET2 \wedge CET3 \wedge CET4$*

$$(CET1) \quad \tilde{\forall} X \doteq X \quad \wedge \quad \tilde{\forall} X \doteq Y \supset Y \doteq X \quad \wedge \quad \tilde{\forall} X \doteq Y \wedge Y \doteq Z \supset X \doteq Z$$

$$(CET2) \quad \{C : Constant;\ n : \mathbb{N};\ let\ X_1, \ldots, X_n, Y_1, \ldots, Y_n : Variable \bullet \tilde{\forall} C_1(X_1, \ldots, X_n) \doteq C_1(Y_1, \ldots, Y_n) \supset \textstyle\bigwedge_{i=1}^{n} X_i \doteq X_i\}$$

$$(CET3) \quad \{C_1, C_2 : Constant;\ n, m : \mathbb{N};\ let\ X_1, \ldots, X_n, Y_1, \ldots, Y_m : Variable\ and\ C_1 \neq C_2\ or\ n \neq m \bullet \tilde{\forall} C_1(X_1, \ldots, X_n) \doteq C_2(Y_1, \ldots, Y_m) \supset \bot\}$$

$$(CET4) \quad \{Te : Term;\ let\ X : Variable\ and\ X \in \mathcal{V}(Te) \bullet \tilde{\forall} X \doteq Te \supset \bot\}$$

Proposition 1 (Minimal Model) *Let P be a normal logic program. Then any model M for $Comp(P) \wedge CET$ is also a model of P.*

The well-definedness of *CET* has been conjectured in [Kun87] and formally shown in [Mah88]. Since the 1980's, the LP framework has found successful applications in many areas of software engineering and AI such as relational databases, mathematical logic, natural language understanding, design automation, symbolic solving of equations, diagnosis, configuration, and biochemical structure analysis. From this background, many state-of-the-art programming environments, such as CHIP [DHS+88], ECLIPSE [MS92], and Oz [Smo95], have emerged. Can we transfer this appealing paradigm to agent design, in particular to the construction of hybrid agents?

4.2 Cognitive Robotics

A first-order variant (Figure 21) has been introduced by [FK97] upon completed clausal programs, i.e., by relying on equivalence definitions and disjunctive goals. During completion, clauses that represent *facts* melt into existentially quantified positive literals including equality statements. Furthermore, *integrity constraints* are introduced which are universally quantified implications. The computational service that is modelled with this logic is to verify for given P : *Program*; Δ : *Facts*; G : *Goal*; and IC : *Constraints* the entailment of the goal by the completion of program and facts $Comp(P \wedge \Delta) \wedge CET \models \tilde{\exists} G$[10] and in addition the *theoremhood of integrity*: $Comp(P \wedge \Delta) \wedge CET \models IC$[11]. Throughout this thesis, we abbreviate these conditions

[10] *Comp* can be straightforwardly extended to equivalence definitions and facts.

[11] Another possibility to treat integrity constraints would be to check their *consistency*. Theoremhood of constraints turns out to be semi-decidable, while consistency is not [FK97].

$SubGoal ::= \top \mid Literal \wedge SubGoal$
$Facts ::= \top \mid Constant(Term^*) \wedge Facts$
$Goal ::= \bot \mid \tilde{\exists} SubGoal \vee Goal$
$Definition ::= \tilde{\forall} Constant(Variable^*) \equiv Goal$
$Program ::= \top \mid Definition \wedge Program$
$Constraint ::= \tilde{\forall} Facts \supset Goal$
$Constraints ::= \top \mid Constraint \wedge Constraints$

Figure 21: First-Order Equivalence Definitions with Constraints

to $P, IC, \Delta \models G$.
Kowalski & Sadri [KS96b] propose this logic as an appropriate foundation to transfer the features of logic programming to the design of agents. They pin down this claim to the question of an agent's having both to act deliberatively as the result of tracing evidences back to a background theory of the world (by the logic program) as well as to act reactively as the result of trying to maintain its mandatory integrity (by the constraints stating, e.g., 'never bump into a wall'). Although a plausible argument, this is too general for situated agents: Just as we would not describe an agent as a Turing machine, but as a specific program running upon it, it is necessary to have a closer look at a logical agent's background theory, i.e., its logic program, in order to determine its model of the world more precisely (Figure 22).

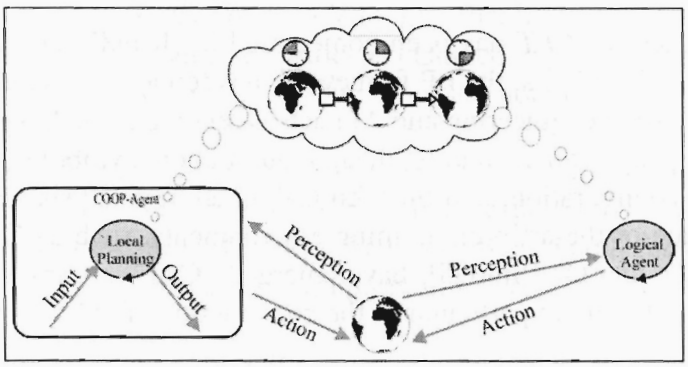

Figure 22: Cognitive Robotics in Hybrid and Unified Agents

There is a substantial amount of research devoted to finding such declarative foundations of situated representation and reasoning. Although the umbrella title *Cognitive Robotics* [LLL+94, Bow87] has been coined in the early 90's, these works can be traced back to the very fundamental ideas of McCarthy & Hayes in the late 50's [McC58, McC63, MH69]. Back then, McCarthy & Hayes proposed to introduce a notion of *time* into logic in order to describe how the state of the world (in the form of information particles, called *fluents*) evolves as being *caused* by *actions*. By axiomatising these concepts, an agent is able to logically trace or explain the frequent observations about the world that it is perceiving and to logically anticipate or plan the future state of affairs including its own actions.

It was not until the publication of [Rei91] that the fundamental *frame problem* posed back in [MH69] has found a first satisfying solution in terms of a first-order theory (Subsection 4.2.1). This has raised the interest in Cognitive Robotics and led to a number of alternative formalisms for realising the McCarthy & Hayes postulate. Today, researchers tend to generalise their ideas about state constraints, side-effects, continuous trajectories, natural actions, etc. across those core calculi. Current implementations show a considerable expressiveness and performance when reasoning about partially observable blocks worlds [Rei99] or robot navigation [Sha97a].
So why should we bother about Agent Engineering at all if there already exists a prototyping methodology for agents? The answer is: because Cognitive Robotics is yet incomplete. As conceptually clean as the separation of declarative theory and computational inference is, it is not enough to derive concrete and implemented agent systems running, e.g., in the Automated Loading Dock and the RoboCup simulation. The more practical agent design is based on the pragmatic and operational guidelines of hybrid architectures and computational models.
So why should we care about Cognitive Robotics at all if we concentrate on traditionally engineered agents? The answer is: because it provides the most convenient supplement when it comes to specify the inner part of primitive modules, i.e., to specify the reasoning processes inside an operational agent framework (Figure 22). For both a unified agent, say a decision maker situated in a dynamic environment, as well as a particular module, say the local planning process operating inside the hybrid InteRRaP-R agent, the same aspects of situated representation and reasoning are highlighted: For both the agent and a process, the world (including other agents and processes) is a partially observable and dynamic environment that is to be rationally explained and controlled. This shows the intimate relation between Cognitive Robotics and Agent Engineering and motivates a common logic framework.
The road-map of the present and the following section is to define such a theory and its underlying inferences respectively which can be applied in a unified decision making agent as well as in a decision making module of a hybrid agent. Concentrating on planning, for the moment, this does not mean that other InteRRaP-R functionalities cannot be handled by this logic-based approach. We will elaborate that, in the end, every process within our hybrid agent model can be expressed as a well-defined subtheory and an associated sub-inference of the complete logic framework.

4.2.1 The State-Based Situation Calculus

The first approach to use a predicate logic formalisation for representing action and change in a situated agent (called *reasoning program* or *advice taker*, back then) has been given by McCarthy & Hayes [McC58, McC63, MH69]. Fluent predicates are annotated with additional situation arguments in order to trace their validity over time $at(box_1, truck, s_0)$[12]. An implicit temporal relation between situations is given by a function *do* which maps situations to successor situations that have been caused by the execution of actions $do(pickup(rob_1, box_1), s_0)$. Change is expressed as an implication of the applicability of actions, i.e., the validity of their preconditions:

[12] We use lower-case letters to distinguish individual constants box_1 from individual variables Box_1.

$$(SIT)\quad \tilde{\forall} holding(Box, Rob, do(pickup(Rob, Box), S)) \subset \begin{array}{l} \tilde{\exists} ahead(Rob, Box, S) \\ \quad \wedge at(Box, Area, S) \\ \quad \wedge handempty(Rob, S) \end{array}$$

Given a description of an initial situation I : *Facts*, such as

$$I ::= ahead(rob_1, box_1, s_0) \wedge at(box_1, truck, s_0) \wedge handempty(rob_1, s_0)$$

and a goal G : *Goal*, such as $G ::= holding(box_1, rob_1, S)$, this *Situation Calculus* is able to reason about the connection of a situation with a goal, i.e., it can both analyse and synthesise plans: $I \wedge SIT \models \tilde{\exists} G$. This early formalisation has been implemented by Green [Gre69] and, as shown by [GLR91], its expressiveness exceeds the one of typical planning algorithms of today [FN71, BF95, KS96a]. For example, it is possible to reason about partial initial situations as well as conditional and universal effects of actions. At the same time, a very principle problem behind the Situation Calculus has been revealed [MH69] that is the problem of determining persistent facts (non-effects) which do not change when applying an action, for example, the location of other boxes, the category of their content, etc. In the Situation Calculus, these frame fluents have to be treated just as the changing fluents by additional axioms (*FRA*):

$$(FRA)\quad \begin{array}{l} \tilde{\forall} at(Box_1, Area, do(pickup(Rob, Box_2), S)) \subset \tilde{\exists} at(Box_1, Area, S) \\ \qquad\qquad \wedge \neg Box_1 \doteq Box_2 \\ \tilde{\forall} at(Box_1, Area, do(label(Rob, Box_2), S)) \subset \tilde{\exists} at(Box_1, Area, S) \\ \tilde{\forall} category(Box_1, Cat_1, do(label(Rob, Box_2, Cat_2), S)) \subset \\ \qquad \tilde{\exists} category(Box_1, Cat_1, S) \wedge \neg Box_1 \doteq Box_2 \\ \tilde{\forall} category(Box_1, Cat_1, do(pickup(Rob, Box_2), S)) \subset \tilde{\exists} category(Box_1, Cat_1, S) \end{array}$$

This *frame problem* is nowadays well-recognised as one of the classical problems of AI. Besides its philosophical aspect, it has great engineering repercussions: For specifying a problem domain such as the Automated Loading Dock in the Situation Calculus, it is a tedious task to specify all the non-effects in the form of separate axioms, i.e., one for each fluent and each action. Small changes in fluent and action representation amount to great changes in the axiomatisation. In order to make the logic-based approach to decision making practical, a different formalisation has to be found in which the concise specification of effects implicitly also determines the non-effects. Such a solution to the frame problem should possibly not restrict expressiveness as done in traditional planning algorithms.

Equivalences, such as used in the Clark completion, play an important role in the calculus proposed by Reiter [Rei91]. By combining all the explicit evidences for a fluent being changed into a single definition, we derive Successor-State-Axioms (*SSA*). From these equivalences, the independence of fluents and actions can then be logically derived, such as the persistence of *category* over any *pickup* action:

$$(SSA)\quad \tilde{\forall} category(Box, Cat, do(A, S)) \equiv \begin{array}{l} \tilde{\exists} A \doteq label(Rob, Box, Cat) \\ \vee \tilde{\exists} category(Box, Cat, S) \\ \quad \wedge \neg \tilde{\exists} A \doteq label(Rob, Box, Cat_2) \end{array}$$

The formulation of Reiter is the basis of the GOLOG language [LLL+94]. However, it has the drawback of not separating its basic reasoning principle from the domain representations of fluents and actions. In *SSA*, both aspects are intermingled. To enable the extraction of such a domain-independent logic program, we have to switch from a fluent representation by means of predicates to a fluent representation by means of manipulatable objects, hence terms of our theory. The appropriate technical notion is called *reification* and has been applied by Kowalski [KS94] to obtain a variant of the Situation Calculus (*SITK*) which looks like a domain-independent version of *SSA*:

$$
\begin{array}{ll}
 & \tilde{\forall} holds(F, do(A, S)) \equiv \quad \tilde{\exists} initiates(A, F, S) \\
(SITK) & \qquad\qquad\qquad\quad \vee \tilde{\exists} holds(F, S) \\
 & \qquad\qquad\qquad\qquad\quad \wedge \neg terminates(A, F, S)
\end{array}
$$

In *SITK*, the *holds* predicate is introduced to describe whether a given fluent, now as an element of the universe, is true in a particular situation. We can now separate an initial situation description

$$
\begin{array}{c}
I ::= holds(ahead(rob_1, box_1), s_0) \wedge holds(at(box_1, truck), s_0) \wedge \\
holds(handempty(rob_1), s_0) \wedge holds(category(box_1, toys), s_0)
\end{array}
$$

and a goal $G ::= holds(holding(rob_1, box_1, do(pickup(rob_1, box_1, s_0))))$ from the background theory *SITK* and a domain description *DOM*. *DOM* determines the positive and negative effects of domain actions by means of the predicates *initiates* and *terminates*.

$$
\begin{array}{ll}
 & \tilde{\forall} initiates(A, F, S) \equiv \quad \tilde{\exists} F \doteq holding(Rob, Box) \\
 & \qquad\qquad\qquad\qquad\quad \wedge A \doteq pickup(Rob, Box) \\
 & \qquad\qquad\qquad\qquad\quad \wedge holds(at(Box, Area), S) \\
 & \qquad\qquad\qquad\qquad\quad \wedge holds(ahead(Rob, Box), S) \\
 & \qquad\qquad\qquad\qquad\quad \wedge holds(handempty(Rob), S) \\
 & \qquad\qquad\qquad\quad \vee \tilde{\exists} F \doteq category(Box, Cat) \\
 & \qquad\qquad\qquad\qquad\quad \wedge A \doteq label(Rob, Box, Cat) \\
 & \qquad\qquad\qquad\quad \vee \ldots \\
(DOM) & \tilde{\forall} terminates(A, F, S) \equiv \quad \tilde{\exists} F \doteq at(Box, Area) \\
 & \qquad\qquad\qquad\qquad\quad \wedge A \doteq pickup(Rob, Box) \\
 & \qquad\qquad\qquad\qquad\quad \wedge holds(at(Box, Area), S) \\
 & \qquad\qquad\qquad\qquad\quad \wedge holds(ahead(Rob, Box), S) \\
 & \qquad\qquad\qquad\qquad\quad \wedge holds(handempty(Rob), S) \\
 & \qquad\qquad\qquad\quad \vee \tilde{\exists} F \doteq category(Box, Cat_1) \\
 & \qquad\qquad\qquad\qquad\quad \wedge A \doteq label(Rob, Box, Cat_2) \\
 & \qquad\qquad\qquad\quad \vee \ldots
\end{array}
$$

This gives us the desired framework $SITK \wedge DOM \wedge I \models \tilde{\exists} G$ (short notation: $DOM \wedge I \models_{SITK} \tilde{\exists} G$). In recent years, the reification technique has been extended to also cover possible combinations of fluents in partial situation descriptions [HS90, Thi99]. Their *Fluent Calculus* uses a particular equational theory (the multi-set theory *AC*1 [GHS+92]) to axiomatise changes in partial situations using State-Update-Axioms (*SUA*). It can be shown that this treatment allows for computational advances when inferring the frame.

4.2.2 The Narrative-Based Event Calculus

State-based approaches to Cognitive Robotics, such as the Situation Calculus, are characterised by their implicit notion of time and their explicit focus on global states. It has been argued, especially in the context of natural language systems and narrative understanding [All84, KS86], that this is a major reason for the frame problem to appear — states enforce the distinction of what is relevant for the reasoning from what is not. Furthermore, they are a cognitively non-plausible representation for a human hearer or reader: In narratives and discourse, seldom an overall description of the initial world state is given and hardly a complete sequence of actions is told. Rather, the important parts of the story are introduced piecewise and presented with incomplete temporal annotations ("first, there was a box labelled with 'toys' standing on the truck ... one robot picked it up ... guess what its category was after the other robot labelled it with 'guns'."). The hearer or reader of the story then has to reason under the assumption that all the relevant information has been given to him. Sometimes, he has to withdraw wrong conclusions when getting more information (non-monotonic reasoning).
The role of an interactive hearer or reader is a natural picture for a situated agent within a multi-agent system, too. The agent cannot perceive every detail of the world, but is frequently gathering bits of information whose temporal ordering could be unclear. From these bits, the agent must derive preliminary conclusions and decisions. Hence, the agent is not able to project a complete state representation of the world into past and future, but only the relevant parts of its vague estimation thereof. That is why narrative-based logics of action, which were originally developed in the discourse understanding and temporal database background to avoid global states, turn out as useful formalisms for Cognitive Robotics, too.
One of these formalisms is the *Event Calculus* which has been introduced in [KS86] and brought into a form quite similar to *SITK* by [Sha89, KS94]
($ECK ::= ECK1 \wedge ECK2, ECK3 \wedge ECK4 \wedge ECK5$).

$$(ECK1)\quad \tilde{\forall} holds(F,T_1) \equiv \tilde{\exists} happens(E,A,T_2) \wedge initiates(A,F,T_2) \wedge T_2 \dot{<} T_1 \wedge \neg clipped(F,T_2,T_1)$$

$$(ECK2)\quad \tilde{\forall} clipped(F,T_1,T_2) \equiv \tilde{\exists} happens(E,A,T_3) \wedge terminates(A,F,T_3) \wedge T_1 \dot{\leq} T_3 \dot{\leq} T_2$$

$$(ECK3)\quad \tilde{\forall} T_1 \dot{<} T_2 \wedge T_2 \dot{<} T_3 \supset \tilde{\exists} T_1 \dot{<} T_3$$

$$(ECK4)\quad \tilde{\forall} T_1 \dot{<} T_1 \supset \tilde{\exists} \perp$$

$$(ECK5)\quad \tilde{\forall} happens(E,A_1,T_1) \wedge happens(E,A_2,T_2) \supset \tilde{\exists} A_1 \doteq A_2 \wedge T_1 \doteq T_2$$

The ontological entities in *ECK* are fluents, *events* (as unique tokens of a certain type of action), and time points. Similar to Kowalski's approach in *SITK*, there exists a *holds* predicate which denotes that a certain fluent is valid at a particular point in time. The axiom of change *ECK*1 realises a restricted version of the law of strict inertia: A fluent

holds at a particular point in time if and only if there exists an event that happened (the *happens* predicate) at earlier in time (the before relation $\dot{<}$ in infix notation; $T_1 \dot{\leq} T_2$ is an abbreviation for $T_1 \dot{<} T_2 \vee T_1 \doteq T_2$) and that has successfully initiated the fluent — unless the fluent ceased to persist in the meantime (it is *clipped*: *ECK*2) as the result of being terminated by some other event. *ECK*3 and *ECK*4 are the background constraints to obtain temporal order as a transitive and anti-symmetric relation. *ECK*5 describes events to be unique and instantaneous action appearances over time.
Given an initial situation[13] and a narrative

$$I ::= holds(on(box_1, truck), t_0) \wedge holds(category(box_1, toys), t_0)$$

$$\Delta ::= happens(e_1, label(rob_2, box_1, guns), t_1) \wedge t_0 \dot{<} t_1 \wedge$$
$$happens(e_2, pickup(rob_1, box_1), t_2) \wedge t_0 \dot{<} t_2 \wedge t_1 \dot{<} t_3 \wedge t_2 \dot{<} t_3$$

and the identical domain description *DOM* as in the *SITK* case[14], we could now infer

$$ECK1 \wedge ECK2 \wedge DOM \wedge I \wedge \Delta, ECK3 \wedge ECK4 \wedge ECK5 \models \tilde{\exists} holds(category(box_1, guns), t_3)$$

(short notation: $DOM \wedge I \wedge \Delta \models_{ECK} \tilde{\exists} G$).
This result is due to the completion of the situation I and the narrative Δ that was not mandatory in the Situation Calculus. In *SITK*, a closed-world assumption can connect partial states to global situations, but is not able to deal with incomplete temporal information. Quite as the Situation Calculus is regarded as the theory behind *state-space planners* [McC85], the completion of $\dot{<}$ into a partial order thus closely relates the Event Calculus to algorithmic *partial-order planning* [Esh88, Mis91, Sha97a].
However, the notion of a correct plan in *ECK* is different from the common intuition. Typically, a solution plan is one which satisfies the goal in all of its linearisations, i.e., its extensions to totally ordered plans. By minimising the $\dot{<}$ relation, *ECK* already demonstrates the validity of effects under the existence of a single successful linearisation; Suppose we add another action and some initial facts

$$\Delta ::= happens(e_3, pickup(rob_2, box_1), t_4) \wedge t_0 \dot{<} t_4 \wedge t_4 \dot{<} t_3$$

$$I ::= holds(handempty(rob_1), t_0) \wedge holds(handempty(rob_2), t_0) \wedge$$
$$holds(ahead(rob_1, box_1), t_0) \wedge holds(ahead(rob_2, box_1), t_1)$$

we then infer $G ::= \tilde{\exists} holds(holding(rob_1, box_1), t_3) \wedge holds(holding(rob_2, box_1), t_3)$ which is not intuitive.
For partial-order planning with the Event Calculus, an alternative formalisation $ECS ::= ECK1 \wedge ECS1 \wedge ECS2, ECK3 \wedge ECK4 \wedge ECK5$ has been proposed by [Sha89, Mis91, Sha95]. Instead of qualifying persistence-destroying events inside the persistence interval, they are now required not to happen outside the interval bounds.

$$\tilde{\forall} clipped(F, T_1, T_2) \equiv \tilde{\exists} happens(E, A, T_3)$$
$$(ECS1) \qquad \wedge terminates(A, F, T_3)$$
$$\wedge \neg out(T_3, T_1, T_2)$$

[13] According to *ECK*1, each *holds* expression must have an associated initiator. In the Event Calculus, the initial situation is thus normally described by a particular 'dummy' event in Δ and *DOM* which introduces all the initial fluents. We have omitted this for the purpose of simplification.
[14] The correspondence of situations and time points has been used to compare *ECK* and *SITK* [KS94].

$$(ECS2) \quad \tilde{\forall} out(T_3, T_1, T_2) \equiv \begin{array}{l} \tilde{\exists} T_3 \dot{<} T_1 \\ \vee \tilde{\exists} T_2 \dot{<} T_3 \end{array}$$

ECS restricts its models in such a way that a solutionplan has to be correct in all of its linearisations. In the absence of relevant temporal information, such as in our previous example, *ECS* would not predict that the box is kept by any of the robots — this is what we expect from any hearer that requires more information in order to resolve a story. But this restriction comes at the high price of a computationally intractable semantics! For our example, we can construct two minimal *ECS* models which differ in the validity of statements: one in which rob_1 is successfully picking box_1, thus rendering the action of rob_2 non-effective, and one in which the opposite is the case. To give this a semantical basis, the notion of three-valued models (Definition 7) has been introduced by [Kun87]. The minimal three-valued *ECS* model of our example hence leaves the truth values for preconditions and effects of both *pickup* actions 'undefined' (0.5).

Definition 6 (First-Order Three-Valued Structure) *A first-order three-valued structure* $M = (U, IF, IP)$ *consists of a universe* U*, a functional interpretation* $IF : Constant \times \mathbb{N} \rightarrow (U^* \rightarrow U)$ *and a relational interpretation* $IP : Constant \times \mathbb{N} \rightarrow (\mathbb{P}\, U^* \rightarrow \{0, 0.5, 1\})$*, such that for all* $C : Constant$*;* $n : \mathbb{N}$*,* $IF(C, n) \in (U^n \rightarrow U)$ *and* $IP(C, n) \in (\mathbb{P}\, U^n \rightarrow \{0, 0.5, 1\})$*.*

Definition 7 (First-Order Three-Valued Model) *We define three-valued assignments, three-valued valuations* $\models_3^{\{0,1\}}$*, and three-valued models* $\models_3$ *identical to two-valued assignments,* $\models_2$*, and* $\models_2^{\{0,1\}}$ *in Definitions 2 and 3.*

Proposition 2 (Two-Valued and Three-Valued Models) *Every first-order two-valued model of a formula is also a first-order three-valued model of that formula.*

Proposition 3 (Two-Valued Models and Undefinedness)
Let $M_1 = (U, IF, IP_1)$ *and* $M_2 = (U, IF, IP_2)$ *be two two-valued models for a formula* $F : Wff$*. Then there exists a three valued model* $M_3 = (U, IF, IP_3)$ *of* F*, such that*

$$IP(C, n)(U_1, \ldots, U_n) = \begin{cases} 0.5, & IP_1(C, n)(U_1, \ldots, U_n) \neq \\ & IP_2(C, n)(U_1, \ldots, U_n) \\ IP_1(C, n)(U_1, \ldots, U_n), & else \end{cases}$$

Undefinedness gives a natural interpretation to the *ECS* behaviour. Nevertheless, it is computationally intractable: In a minimal three-valued model, the conditions for $\models^1$ and $\models^0$ turn out to be computable, but $\models^{0.5}$ describes the case in which an inference procedure cannot decide and does not halt. In other words, inference procedures will steadily loop ('flounder') between the minimal two-valued models. Especially during planning within ignorant agents, such as the forklifts in the loading dock, such cases could appear rather often and 'paralyse' the agents forever.
Different semantics, such as stable models [GL88] and well-founded models [GRS88], have been proposed to allow for more useful inferences, at the same time keeping the expressiveness of the LP framework. We could adopt such semantics for *ECS* as well. Since this would require special inference procedures, this would restrict the range of implementability with respect to standard platforms for logic programming, such as constraint-based languages.

Instead, we have taken in [JFB96a] the pragmatic approach to refine the calculus in order to allow for unique two-valued models, again. At first sight, this is in conflict with a purist view on Cognitive Robotics. It is however justified as long as the calculus keeps its intuitive form, i.e., the extensions have a declarative reading. This argument will be used again when talking about representational extensions in the following subsection.

The crucial observation of [JFB96a] was that, instead of running into mutual dependencies, some sort of pessimistic worst-case analysis has to be performed by the calculus, i.e., to apply the most conservative notion of a partially specified planning solution that is available. A first approach is thus to omit the precondition checks invoked by the *clipped* axioms. This way, any action, even if its preconditions are not valid, could threat the persistence of some fluent and requires efforts from the agent in order to re-establish the wanted effect. Such efforts could be to strengthen temporal constraints or to insert repair actions. This scheme does however not allow to incorporate immediate counter-measures to 'neutralise' adversary actions in advance. These are highly necessary means in non-cooperative multi-agent systems [EM91].

For this purpose, the calculus

$$ECJ ::= ECJ1 \wedge \ldots \wedge ECJ4 \wedge ECS3, ECK3 \wedge \ldots \wedge ECK5$$

has been presented which just focuses its worst-case analysis on mutual dependencies, thus is able to reason about the preconditions of persistence destroyers as well. *ECJ* predicates, such as *holds*, are extended to keep book about the visited events in a causal chain C (using a list representation, e.g., $C \doteq cons(e_1, cons(e_2, \ldots))$), the current worst case B, and the event E that is currently under consideration:

$$\begin{aligned}\tilde{\forall} holds(F, T_1, C, B, E) \equiv\ & \tilde{\exists} member(E, C) \wedge B \doteq \dot{1} \\ & \vee \tilde{\exists} \neg member(E, C) \\ & \quad \wedge happens(E_2, A, T_2) \\ & \quad \wedge initiates(A, F, T_2, cons(E, C), B, E_2) \\ & \quad \wedge T_2 \dot{<} T_1 \\ & \quad \wedge flip(B, B_2) \\ & \quad \wedge \neg clipped(F, T_2, T_1, C, B_2, E)\end{aligned} \tag{ECJ1}$$

$$\begin{aligned}\tilde{\forall} clipped(F, T_1, T_2, C, B, E) \equiv\ & \tilde{\exists} happens(E_2, A, T_3) \\ & \quad \wedge terminates(A, F, T_3, cons(E, C), B, E_2) \\ & \quad \wedge \neg out(T_3, T_1, T_2)\end{aligned} \tag{ECJ2}$$

$$\begin{aligned}\tilde{\forall} flip(B_1, B_2) \equiv\ & \tilde{\exists} B_1 \doteq \dot{1} \wedge B_2 \doteq \dot{0} \\ & \vee \tilde{\exists} B_1 \doteq \dot{0} \wedge B_2 \doteq \dot{1}\end{aligned} \tag{ECJ3}$$

$$\begin{aligned}\tilde{\forall} member(E, C) \equiv\ & \tilde{\exists} C \doteq cons(E, C_2) \\ & \vee \tilde{\exists} C \doteq cons(E_2, C_2) \\ & \quad \wedge member(E, C_2)\end{aligned} \tag{ECJ4}$$

ECJ reasons similar to *ECS* unless trying to prove the precondition of some event E in *ECJ*1 which has already been entered into that list (the *member* predicate in *ECJ*4). Then, we have detected some causal cycle and a worst case assumption must

be applied: The worst case is indicated by the additional parameter B which can take any of the values $\dot{0}$ and $\dot{1}$. For example, the worst case for wanting to demonstrate the validity of an effect is that the precondition of its initiator does not hold ($\dot{0}$). For example, the worst case for demonstrating the persistence of some fluent is of course that a possible destroyer is successfully terminating that fluent ($\dot{1}$). Hence, B is flipped (*ECJ*3) each time it crosses a negation in the calculus. The ultimate goals for *ECJ* start with an empty causal chain *nil* and refer to a 'dummy' consumer E, e.g., $holds(category(box_1, guns), t_3, nil, \dot{0}, E)$. Also the domain description *DOM* is correspondingly extended:

(*DOM*)

$$\begin{aligned}
\tilde{\forall} initiates(A, F, T, C, B, E) \equiv \quad & \tilde{\exists} F \doteq holding(Rob, Box) \\
& \wedge A \doteq pickup(Rob, Box) \\
& \wedge holds(at(Box, Area), T, C, B, E) \wedge \ldots \\
\tilde{\forall} terminates(A, F, T, C, B, E) \equiv \quad & \tilde{\exists} F \doteq at(Box, Area) \\
& \wedge A \doteq pickup(Rob, Box) \\
& \wedge holds(at(Box, Area), T, C, B, E) \wedge \ldots
\end{aligned}$$

We shall now prove the well-definedness of *ECJ* and use the following argumentation: Any undefined value in a minimal three-valued *ECJ* model can only affect the defined predicates *holds*, *clipped*, *initiates*, and *terminates* and results in an infinite sequence of undefined *holds* values that incrementally build up a causal chain (Proposition 4). For any given narrative, this chain then must have a cycle. From the definition of the calculus, it follows that the appropriate *holds* value must be either 1 or 0 (Proposition 5). Since this prohibits any proper minimal three-valued model for *ECJ*, we can derive the uniqueness of an appropriate minimal two-valued model (Theorem 1). Using a similar argumentation, Theorem 2 shows that any statement valid in the worst case ($\dot{0}$) is also valid in the optimistic case ($\dot{1}$). Some of the proofs can be found in Appendix B.

Proposition 4 (Existence of Infinite Sequences in *ECJ*) *Let M be a minimal three-valued (Herbrand) model for ECJ under I, Δ, DOM:*

$$M \models_3 Comp(ECJ1 \wedge \ldots \wedge ECJ4 \wedge ECS3 \wedge \Delta \wedge I \wedge DOM) \wedge CET \wedge ECK3 \wedge \ldots \wedge ECK5$$

For all C : Constant; $n : \mathbb{N}$, it holds $0.5 \in ran\, IP(C, n)$ if and only if there exists an infinite sequence $i : \mathbb{N}$; $U_{i,1}, \ldots, U_{i,6} : U$ such that $IP(holds, 5)(U_{i,1}, \ldots, U_{i,5}) = 0.5$, $IP(happens, 3)(U_{i,5}, U_{i,6}, U_{i,2}) = 1$, and $U_{i+1,3} = IF(cons, 2)(U_{i,5}, U_{i,3})$.

Proof. see Appendix B. □

Proposition 5 (Three-Valued Minimal Models of *ECJ*) *Any minimal three-valued (Herbrand) model M of ECJ is already a two valued model of ECJ.*

Proof. see Appendix B. □

Theorem 1 (Unique Minimal Two-Valued Model of *ECJ*) *ECJ has a unique minimal two-valued (Herbrand) model*

$$M \models_2 Comp(ECJ1 \wedge \ldots \wedge ECJ4 \wedge ECS3 \wedge \Delta \wedge I \wedge DOM) \wedge CET \wedge ECK3 \wedge ECK4 \wedge ECK5$$

Proof. Suppose that there are two-valued (Herbrand) models M_1, M_2 which differ in the truth value of some $IP(C,n)(U_1,\ldots,U_n)$. Then by Proposition 3, we could construct a three-valued (Herbrand) model M_3 with $IP(C,n)(U_1,\ldots,U_n) = 0.5$. This contradicts Proposition 5. □

Theorem 2 (Treatment of Worst Case in *ECJ*) *Let M be the minimal (Herbrand) model of ECJ:*

$$M \models \tilde{\forall} holds(F,T,C,\dot{0},E) \supset holds(F,T,C,\dot{1},E)$$

Proof. see Appendix B. □

Due to its expressiveness and computational properties, *ECJ* has been successfully applied in the context of the original InteRRaP architecture and the Automated Loading Dock by standard LP techniques [JFB96a]. Especially the ability to treat partially-ordered multi-agent plans has been a key requirement to encode the delivery tasks and the necessary coordination between forklifts. [Sha97a] has shown that abductive inferences with the Event Calculus closely mirror the behaviour of partial-order planning algorithms, giving a declarative meaning to concepts such as protected links, threats, clobberers, and the promotion and demotion of clobberers. In recent years, several alternative narrative-based formalisms have been developed for dealing with the frame problem. The *Temporal Action Language (TAL)* [DGKK98, San94], for example, grew out of an evaluation framework for action logics. It is currently applied in the off-line verification of an unmanned airborne vehicle. But, it has not yet been integrated into on-line decision making.

4.3 Abstraction In The Event Calculus

With the core formalisms of *SSA*, *SUA*, *ECJ*, and *TAL*, the practical impact of Cognitive Robotics has been sufficiently demonstrated and the frame problem seems to be solved today. Research now focuses on other aspects, such as indeterminate effects [Sha97b, BT97, Lin95], simultaneous actions [Sha97c, BT94, LS95], the modelling of continuous actions [Sha90, HT96], and the incorporation of state-constraints and side-effects of actions (the *ramification* problem [Thi97, KM97a]).
These extensions develop increasingly sophisticated, thus increasingly expensive reasoning machines without worrying about the foremost requirement of situatedness both for agents as well as for particular reasoning modules inside agents: the need for doing early and approximate decisions. Our experiments in [JFB96a], for example, have demonstrated that an *ECJ*-based planner is able to navigate a forklift's BBL, but only if the timing requirements are not too tight. Otherwise, the planner takes too long at computing future details, such as complete navigation paths, for influencing the fast BBL reactions in time.

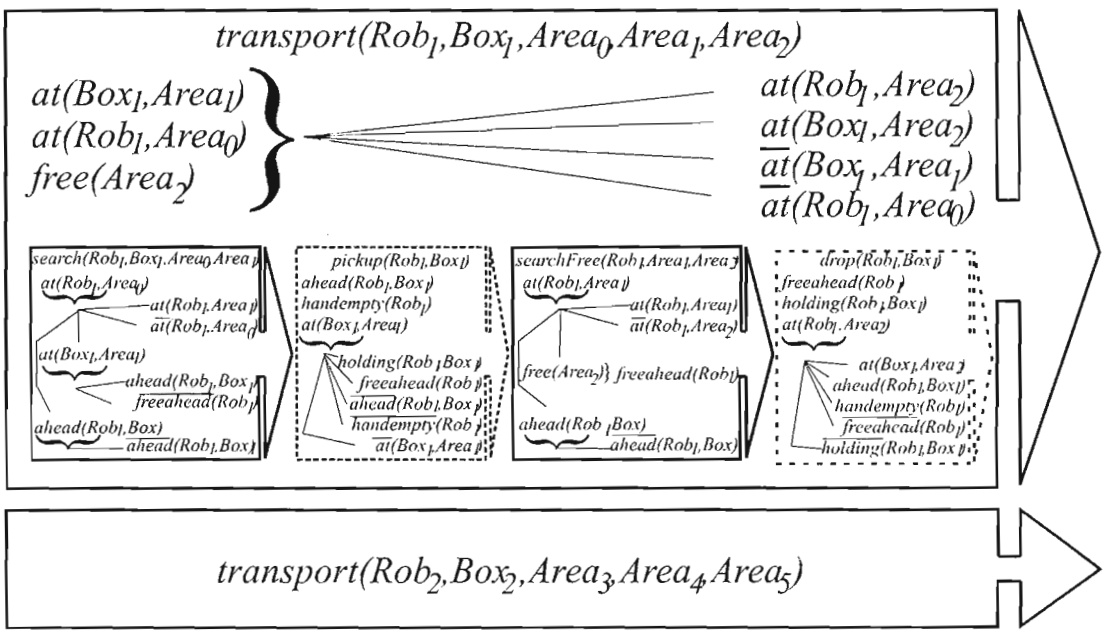

Figure 23: An Abstraction Hierarchy

Instead of being occupied with details, a reasoner should be able to first treat the important issues of the problem, hence to solve its problem approximately. Later, this solution sketch is to be refined into a detailed result. This is the idea of anytime algorithms [BD94]. In a logic-based setting, abstract representations are an intuitive means to indicate which features of the original problem specification are most important and to hide other information for later incorporation [tTvH98]. For planning systems, this has been most reasonably argued by [Sac74]. Because abstract representations are organised in a decomposition hierarchy, we often speak of hierarchical planning. In hierarchical planning, abstraction can be applied to fluents (*situation abstraction*) and narratives (*action abstraction*). The latter subsumes the former if regarding initial situations and goals as 'dummy' actions with no preconditions or no effects, respectively. Abstraction planning is useful for interleaving planning and action in real-time architectures [WHR96]. Figure 23 shows an extract of a forklift's representation hierarchy in which two *transport* actions are performed concurrently. From an abstract viewpoint, *transport* is defined as an opaque action with preconditions and effects that describe the movement of robot and delivered box. Hereby, *transport* looses information about particular fluents which have to do with the robot's positioning (areas of the loading dock, their reachability, etc.) and with the robot's ability to pick and drop a box. The *transport* macro also looses information about its complex temporal substructure that consists of two sequential sub-actions which are macros by themselves (*search the box and deliver it to a free destination*, *searchFree*).

transport allows to quickly connect a delivery goal with the current situation. Using this decision as a kind of 'promise' for being able to solve the goal, a planner could already commit to certain actions, e.g., by influencing a forklift's BBL to move the robot to the initial area of the box, while still refining *transport* on the next level of representation, e.g., to insert *pickup* and *drop* actions and to develop a complete navigation

path to the destination.
Thus, both for a planning module inside InteRRaP-R and for a single decision-making agent, abstract representations are a useful tool. Put in a more general context, any rational agent and any situated reasoning process must always be aware of its representations naturally being abstractions of the real world. To cater for this aspect, a declarative foundation for abstraction hierarchies in Cognitive Robotics has to be found. One prerequisite, the composition of primitive actions into macro actions, has already been discussed for most of the core calculi [LRL97, San94, Dav96, EHT96]. In these extensions, macros are not allowed to have effects by themselves. Causal reasoning is still performed at the most concrete level of representation. Shanahan [Sha97a] goes further by introducing effect axioms also for macro events. Still, his macros are not to be called abstract, since their effects must be logically equivalent to the lower level axiomatisation. By this design, macros do not really loose information which gives no advantage for enabling approximate reasoning.
It is the loss of information that makes real abstractions a non-trivial concept for logic-based treatment. In the example of Figure 23, the conclusion $at(Box_1, Area_2)$ is provable at a high level of abstraction. But this conclusion cannot be necessarily made at the next lower level of abstraction, since, e.g., the delivering robot could already be occupied with some other box ($\neg holds(handempty(Rob_1), \ldots)$ alternatively written as the *dual* fluent $holds(\overline{handempty}(Rob_1), \ldots)$) which we have not taken into account before and which requires additional efforts, e.g., to drop the carried box, in order to install the wanted result. Hence, treating macros by purely semantical or inferential means, such as above approaches promote to preserve their minimal ontology, would require some sort of non-monotonic reasoning principle.
Now we return to our previous argument in the context of *ECJ*. Why not extend the calculus in order to explicitly deal with abstractions if this allows for an intuitive construct (accessible by an agent programmer) and at the same time for a broader implementability because of standard interpretations?
The contribution of this section is to introduce causally-effective macros at separate levels of abstraction into a logic theory of action and time which we call the *Hierarchical Event Calculus* (*HEC*). We chose the Event Calculus, in particular *ECJ*, as the basis because of positive experience with its narrative-based reasoning in multi-agent settings. It will turn out that, similar to the relation between Event Calculus and partial-order planning, we can install a one-to-one correspondence between *HEC* inferences and hierarchical partial-order planning.

4.3.1 Prerequisites

Representing macros and levels of abstraction in the Event Calculus needs a few prerequisites which we would like to discuss before giving their formalisation. From these, it will be apparent that *HEC* keeps an intuitive reading, moreover unveils and addresses a deeper problem in reasoning about causality that is also inherent to single-level approaches, such as *ECS*.

Duration In *ECK*, *ECS*, and *ECJ*, events are ideally regarded as instantaneous. When switching to macro actions, such as *transport*, this idealisation does not hold anymore.

Since macros are complex compositions of temporal substructures, e.g., they are possibly long-lasting configurations of underlying reactive processes, they must have a positive duration. Therefore, events must be assigned a time interval consisting of a start time point and an end time point. The end time is greater or equal to the start time.

Preconditions In *ECK*, *ECS*, and *ECJ*, preconditions are valid iff they are provably present at the start time of the respective instantaneous event which is equal to its end time. In *HEC*, events represent opaque substructures with duration. Thus, it is not possible to prove preconditions just at the start of some action, such as to check $free(Area_2)$ just at the beginning of a *transport*.
Worst-case assumptions are the right tool to deal with the absence of further information at this level of reasoning: In the worst case, preconditions are needed by some sub-event of the macro which is located quite at the end (*searchFree* in *transport*). Hence, it is safe to speak of a valid precondition iff it has been demonstrably initiated before the start of the macro and is not clipped until its very end. For example, we need to ensure that until the end of *transport*, no concurrent activity is able to put a different box to the last free space of the envisaged shelf, hence does not terminate $free(Area_2)$.

Effects When do effects become visible? Similar to the consumption of a precondition, the effect of a macro ($at(Rob_1, Area_2)$ in *transport*) could be produced by some sub-event relatively late with respect to the overall duration (here: *searchFree*). Thus, we can take effects not for granted until the very end of some action.
On the other hand, effects could as well be caused by some sub-event rather early in the course of the macro, such as $at(Rob_1, Area_0)$ being terminated by *search*. Therefore, the persistence of preconditions is violated right from the beginning of some initiating action. Vice versa, terminating effects violate the persistence of preconditions right from the beginning of a destroyer event. Using these conservative rules, we take as much as possible care of the further refinement of an abstract plan.

Causality Interestingly, a special version of the above worst-case assumptions has already been present in the *ECK* and *ECS* calculi. *ECK* and *ECS* state that initiators have to happen before ($\dot{<}$) the consumption of their effects while destroyers already influence simultaneous settings ($\dot{\leq}$).
The inherent possibility of running into causal cycles with that design leads to the computational intractability of *ECS* by partially undefined minimal models. A possible fix is the requirement that no two actions can happen simultaneously [KS94]. For *HEC*, this is too heavy a restriction, since actions have durations and could happen interleaved (Figure 24).
Hence, we take over the solution of *ECJ* not as a purely practical issue to reinstall well-definedness, but also as a deeper question with respect to the applied causality principle: What *ECJ* already anticipated and what is taken over to *HEC* is, in a nutshell, the naive physical stance that excludes any effect from altering its own cause, any event to influence the validity of its own preconditions.

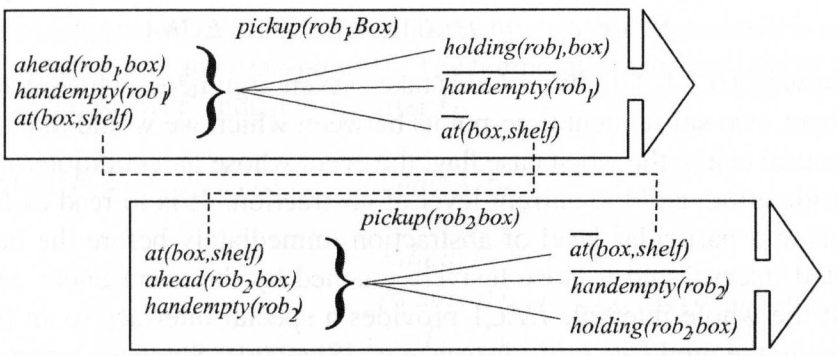

Figure 24: Causal Cycles Lead to Partially Undefined Models

Dual Fluents Relying on the completion of partial information and worst-case analyses, our calculus distinguishes between not being able to demonstrate the validity of a fluent ($\neg holds(at(Rob_1, Area_0), t, \ldots)$) and being able to demonstrate its not being valid (the dual fluent $holds(\overline{at}(Rob_1, Area_0), t, \ldots)$). It is sometimes convenient to talk about the latter case in preconditions, thus we extend the calculus to reify duals and to treat them symmetrically. For example,

$$\neg holds(at(Rob_1, Area_0), t, \ldots) \wedge \neg holds(\overline{at}(Rob_1, Area_0), t, \ldots)$$

should be satisfiable as a matter of ignorance, while

$$holds(at(Rob_1, Area_0), t, \ldots) \wedge holds(\overline{at}(Rob_1, Area_0), t, \ldots)$$

should not.

Level of Abstraction So far, we concentrated on durable and information-loosing macros. Once obtained, the representation of levels of abstraction nearly comes for free: Predicates are simply annotated with abstraction-level terms, for example one term referring to *transport* and corresponding fluents, one level referring to *search*, *searchFree*, *pickup*, and *drop* and their respective fluents, etc. In this way, we can express fluents which are valid at a particular level of abstraction ($holds(at(Box_2, Area_2), t, \ldots, l_2)$ where this does not necessarily imply their being valid at a different level ($holds(at(Box_2, Area_2), t, \ldots, l_3)$. The reasoning at different levels is however not completely separated: An operation which performs the (de-)composition of representations is added and installs the connection between abstract macros and primitive sub-events, between high-level fluents and more concrete state descriptions. This (de-)composition performs bidirectionally, hence serves as a declarative foundation for decomposing approximate plans into refined decisions and for reconstructing high-level intentions from piecewise observations.

4.3.2 The Hierarchical Event Calculus

We now incrementally formalise the Hierarchical Event Calculus

$$HEC ::= HEC1 \wedge \ldots \wedge HEC5 \wedge ECJ3 \wedge ECJ4,$$

$$HEC6 \wedge \ldots \wedge HEC12 \wedge ECK3 \wedge ECK4$$

In the following *HEC*1 definition, *holds* takes seven arguments which denote the envisaged fluent, two subsequent time-points between which we would like the fluent to persist, a causal chain, the worst case flag, the event whose preconditions are currently under consideration, and the current level of abstraction. It is to read as follows: the fluent holds at a particular level of abstraction immediately before the beginning of the indicated interval and it is exclusively touched by the event under consideration throughout the whole interval. *HEC*1 provides a special interface to an initial situation I by using the predicate *initially* (see, e.g., [Sha97a]). Since we assume the initial situation to happen at the very earliest time-point in the narrative, a special version of persistence (*iclipped*) is used. The effects of actions are introduced by a single predicate (*causes*) that is defined in the domain axiomatisation *DOM*. Both *initially* and *causes* operate on fluents and dual fluents.

$$\begin{aligned}
\tilde{\forall} holds(F, T_1, T_2, C, B, E, L) \equiv\ & \tilde{\exists} T_1 \dot{\leq} T_2 \wedge member(E, C) \wedge B \doteq \dot{1} \\
& \vee \tilde{\exists} T_1 \dot{\leq} T_2 \wedge \neg member(E, C) \\
& \quad \wedge initially(F, L) \\
& \quad \wedge flip(B, B_2) \\
& \quad \wedge \neg iclipped(F, T_2, C, B_2, E, L) \\
(HEC1) \qquad & \vee \tilde{\exists} T_1 \dot{\leq} T_2 \wedge \neg member(E, C) \wedge \\
& \quad \wedge happens(E_i, A_i, T_3, T_4, L) \\
& \quad \wedge T_4 \dot{<} T_1 \\
& \quad \wedge causes(A_i, F, T_3, T_4, cons(E, C), B, E_i, L) \\
& \quad \wedge flip(B, B_2) \\
& \quad \wedge \neg clipped(F, T_3, T_2, C, B_2, E, L)
\end{aligned}$$

The *clipped* predicate in *HEC*2 is also extended by the current level of abstraction L. Since it is defined over fluents and dual fluents, a destroyer is now identified by its causing the dual fluent — the *dual* predicate defined in *HEC*3 uses the function symbol *not* to switch between the two fluent versions — and by its not being disjoint (the *disjoint* predicate defined in *HEC*4) with the proper persistence interval. Since any event should not be able to alter its own preconditions, a destroyer furthermore must be different from E. This coincides with our above remark about *holds* in which only E, if any event, is able to touch the fluent throughout the persistence interval.

$$\begin{aligned}
\tilde{\forall} clipped(F, T_1, T_2, C, B, E, L) \equiv\ & \tilde{\exists} happens(E_t, A_t, T_3, T_4, L) \\
& \quad \wedge \neg E_t \doteq E \\
(HEC2) \qquad & \quad \wedge dual(F, F_-) \\
& \quad \wedge causes(A_t, F_-, T_3, T_4, cons(E, C), B, E_t, L) \\
& \quad \wedge \neg disjoint(T_1, T_2, T_3, T_4)
\end{aligned}$$

$$\begin{aligned}
(HEC3) \qquad \tilde{\forall} dual(F, F_-) \equiv\ & \tilde{\exists} F \doteq not(F_-) \\
& \vee \tilde{\exists} F_- \doteq not(F)
\end{aligned}$$

$$\begin{aligned}
(HEC4) \qquad \tilde{\forall} disjoint(T_1, T_2, T_3, T_4) \equiv\ & \tilde{\exists} T_2 \dot{<} T_3 \\
& \vee \tilde{\exists} T_4 \dot{<} T_1
\end{aligned}$$

*HEC*5 defines *iclipped* which checks the defect of persistence between the very beginning of the narrative and any time-point T_1. The temporal constraints in *HEC*2 simplify in this case to a destroyer's not starting after T_1.

$$\begin{aligned}\tilde{\forall} iclipped(F, T_1, C, B, E, L) \equiv \quad & \tilde{\exists} happens(E_t, A_t, T_3, T_4, L) \\ & \wedge \neg E_t \doteq E \\ & \wedge dual(F, F_-) \\ & \wedge causes(A_t, F_-, T_3, T_4, cons(E, C), B, E_t, L) \\ & \wedge \neg T_1 \dot{<} T_3\end{aligned} \tag{HEC5}$$

The following axioms (*HEC*6 – *HEC*9) relate neighbour levels of abstraction. We assume that there exist (de-)composition operations *decomposeMacro* and *decomposeHolds* which are defined in *DOM* and which describe the correspondence of higher-level representations (macro actions and abstract fluents) with more primitive occurrences (sub-events and more concrete fluents). Intuitively, there should be an equivalence between *happens* and *decomposeMacro* which we do not immediately express as a definition. Rather, we use two separate constraints (*HEC*6 and *HEC*8) for 'maintaining the integrity' of the given abstraction hierarchy. This will turn out to be useful in the next section. Abstract *holds* statements are subject to information loss. Hence, lower-level fluents will imply the occurrence of higher-level ones (*HEC*9), but not vice versa. For the opposite direction, we determine a weaker relation (*HEC*7) which just focuses on the initial situation I and requires a *decomposeInitially* definition in *DOM* quite analogous to *decomposeHolds*.

$$\tilde{\forall} happens(E, A, T_1, T_2, L) \supset \quad \tilde{\exists} decomposeMacro(E, A, T_1, T_2, L) \tag{HEC6}$$

$$\tilde{\forall} initially(F, L) \supset \quad \tilde{\exists} decomposeInitially(F, L) \tag{HEC7}$$

$$\tilde{\forall} decomposeMacro(E, A, T_1, T_2, L) \supset \quad \tilde{\exists} happens(E, A, T_1, T_2, L) \tag{HEC8}$$

$$\begin{aligned}&\tilde{\forall} decomposeHolds(F, T_1, T_2, nil, \dot{0}, E_1, L) \supset \\ &\quad \tilde{\exists} holds(F, T_1, T_2, nil, \dot{0}, E_2, L)\end{aligned} \tag{HEC9}$$

Finally, we add three constraints *HEC*10, *HEC*11, and *HEC*12 which state that each event has a positive duration and is unique with respect to its action type, its duration, and its level of abstraction, and that the initial situation must be consistent with respect to dual fluents. The background theory of $\dot{<}$ is lend from *ECK* (*ECK*3 and *ECK*4).

$$\tilde{\forall} happens(E, A, T_1, T_2, L) \supset \quad \tilde{\exists} T_1 \dot{\leq} T_2 \tag{HEC10}$$

$$\begin{aligned}\tilde{\forall} happens(E, A_1, T_1, T_2, L_1) \wedge happens(E, A_2, T_3, T_4, L_2) \supset \quad & \tilde{\exists} A_1 \doteq A_2 \\ & \wedge T_1 \doteq T_3 \\ & \wedge T_2 \doteq T_4 \\ & \wedge L_1 \doteq L_2\end{aligned} \tag{HEC11}$$

$$\tilde{\forall} initially(F, L) \wedge initially(F_-, L) \wedge dual(F, F_-) \supset \quad \tilde{\exists} \bot \tag{HEC12}$$

4.3.3 Domain Representation and (De-)Composition

A narrative in *HEC* (see our example in Figure 23) is a set of facts of the form

$$\begin{aligned}\Delta ::= & \mathit{happens}(e_1, \mathit{transport}(\mathit{rob}_1, \mathit{box}_1, \mathit{parking}, \mathit{truck}, \mathit{shelf}_3), t_1, t_2, l_2) \wedge \\ & \mathit{happens}(e_2, \mathit{transport}(\mathit{rob}_2, \mathit{box}_2, \mathit{parking}, \mathit{shelf}_1, \mathit{truck}), t_3, t_4, l_2) \wedge \\ & \mathit{happens}(e_3, \mathit{search}(\mathit{rob}_1, \mathit{box}_1, \mathit{parking}, \mathit{truck}), t_5, t_6, l_3) \wedge \ldots \wedge \\ & t_1 \dot{\leq} t_5 \dot{\leq} t_6 \dot{\leq} t_2 \dot{<} t_7 \dot{\leq} t_8 \wedge t_3 \dot{\leq} t_4 \dot{<} t_7 \wedge \ldots\end{aligned}$$

Where we had to encode the initial situation I within the narrative and the domain description *DOM* before, this is now much easier to realise using the *initially* predicate:

$$\begin{aligned}I ::= & \mathit{initially}(\mathit{at}(\mathit{box}_1, \mathit{truck}), l_2) \wedge \mathit{initially}(\mathit{at}(\mathit{rob}_1, \mathit{park}), l_2) \wedge \\ & \mathit{initially}(\mathit{free}(\mathit{shelf}_3), l_2) \wedge \mathit{initially}(\mathit{at}(\mathit{box}_1, \mathit{truck}), l_3) \wedge \\ & \mathit{initially}(\mathit{handempty}(\mathit{rob}_1), l_3) \wedge \ldots\end{aligned}$$

Domain-dependent situation abstraction is encoded by means of the following definition of *decomposeHolds* (and an analogue definition of *decomposeInitially*) in *DOMSAB*. It relates particular fluents at higher levels of abstraction to fluents within a more primitive or even the same level of abstraction. For example, the occupancy of areas within the loading dock can be inferred from more concrete positioning data with respect to landmarks (*atPos*). For example, *ahead* can be derived from positioning and orientation. The decomposition of most primitive fluents is simply $\top$.

(DOMSAB)

$$\begin{aligned}& \tilde{\forall}\mathit{decomposeHolds}(F, T_1, T_2, C, B, E, L) \equiv \\ & \quad \tilde{\exists} L \doteq l_3 \wedge F \doteq \mathit{at}(\mathit{Object}, \mathit{truck}) \\ & \qquad \wedge \mathit{holds}(\mathit{atPos}(\mathit{Object}, \dot{1}, \dot{1}), T_1, T_2, C, B, E, l_4) \\ & \quad \vee \tilde{\exists} L \doteq l_4 \wedge F \doteq \mathit{ahead}(\mathit{Rob}, \mathit{Box}) \wedge X_1 \doteq X_2 \wedge Y_2 \doteq \dot{+}(Y, 1) \\ & \qquad \wedge \mathit{holds}(\mathit{atPos}(\mathit{Rob}, X_1, Y_1), T_1, T_2, C, B, E, L) \\ & \qquad \wedge \mathit{holds}(\mathit{orient}(\mathit{Rob}, \mathit{north}), T_1, T_2, C, B, E, L) \\ & \qquad \wedge \mathit{holds}(\mathit{atPos}(\mathit{Box}, X_2, Y_2), T_1, T_2, C, B, E, L) \\ & \quad \ldots\end{aligned}$$

As for *HEC*'s ancestor calculi, *DOM* contains the causal effects of actions *DOMCAU*. These are defined through a single *causes* predicate which assigns both fluents ($\mathit{at}(\mathit{Box}, \mathit{Area}_1)$) and dual fluents ($\mathit{not}(\mathit{at}(\mathit{Rob}, \mathit{Area}_0))$) as the result of executing an action under particular (positive or negative) preconditions. *causes* distinguishes actions according to different levels of abstraction, i.e., the same action type, such as *pickup*, could have more abstract preconditions and effects at a higher level than at a lower level of abstraction, such as the *atPos* fluent which does not become apparent until level l_4.

$$
\begin{aligned}
&\tilde{\forall} causes(A, F, T_1, T_2, C, B, E, L) \equiv \\
&\quad \tilde{\exists} L \doteq l_2 \\
&\qquad \wedge A \doteq transport(Rob, Box, Area_0, Area_1, Area_2) \\
&\qquad \wedge F \doteq at(Box, Area_2) \\
&\qquad \wedge holds(at(Rob, Area_0), T_1, T_2, C, B, E, L) \\
&\qquad \wedge holds(at(Box, Area_1), T_1, T_2, C, B, E, L) \\
&\qquad \wedge holds(free(Area_2), T_1, T_2, C, B, E, L) \\
&\quad \vee \tilde{\exists} L \doteq l_2 \\
&\qquad \wedge A \doteq transport(Rob, Box, Area_0, Area_1, Area_2) \\
&\qquad \wedge \neg Area_0 \doteq Area_1 \\
&\qquad \wedge F \doteq not(at(Rob, Area_0)) \\
&\qquad \wedge \ldots \\
&\quad \vee \tilde{\exists} L \doteq l_3 \\
&\qquad \wedge A \doteq pickup(Rob, Box) \\
&\qquad \wedge F \doteq holding(Rob, Box) \\
&\qquad \wedge holds(at(Box, Area), T_1, T_2, C, B, E, L) \\
&\qquad \wedge holds(handempty(Rob), T_1, T_2, C, B, E, L) \\
&\qquad \wedge holds(ahead(Rob, Box), T_1, T_2, C, B, E, L) \\
&\quad \vee \tilde{\exists} L \doteq l_4 \\
&\qquad \wedge A \doteq pickup(Rob, Box) \\
&\qquad \wedge F \doteq not(atPos(Box, X_1, Y_1)) \\
&\qquad \wedge holds(atPos(Box, X_1, Y_1), T_1, T_2, C, B, E, L) \\
&\qquad \wedge \ldots
\end{aligned}
\tag{DOMCAU}
$$

The final task of *DOM* is to encode abstraction within the temporal narrative by defining the *decomposeMacro* predicate (*DOMAAB*). A successful decomposition of a macro is most straightforwardly described as the occurrence (*happens*) of corresponding sub-events at the next level of abstraction and the validity of temporal constraints between their duration. In our example, the *transport* macro decomposes into a sequence, i.e., a completely ordered set, of sub-actions. It is possible that actions just decompose into more refined versions of themselves, such as it is the case for *pickup* and *drop* from level l_3 to l_4.

$$
\begin{aligned}
&\tilde{\forall} decomposeMacro(E, A, T_1, T_2, L) \equiv \\
&\quad \tilde{\exists} L \doteq l_2 \wedge A \doteq transport(Rob, Box, Area_0, Area_1, Area_2) \\
&\qquad \wedge happens(E_1, search(Rob, Box, Area_0, Area_1), T_1, T_3, l_3) \\
&\qquad \wedge happens(E_3, searchFree(Rob, Area_1, Area_2), T_4, T_2, l_3) \\
&\qquad \wedge T_1 \dot{\leq} T_3 \dot{\leq} T_4 \dot{\leq} T_2 \\
&\quad \tilde{\exists} L \doteq l_3 \wedge A \doteq pickup(Rob, Box) \\
&\qquad \wedge happens(E_1, pickup(Rob, Box), T_1, T_2, l_4) \\
&\quad \tilde{\exists} L \doteq l_3 \wedge A \doteq drop(Rob, Box) \\
&\qquad \wedge happens(E_1, drop(Rob, Box), T_1, T_2, l_4) \\
&\quad \ldots
\end{aligned}
\tag{DOMAAB}
$$

Using the expressiveness of first-order logic, the (de-)composition predicates can be converted into a powerful description tool. For example, arbitrarily interleaved activities, such as the two delivery macros in Figure 23, can be described in *DOMAAB* by

loose temporal relations

$$\tilde{\exists} happens(E_1, transport(\ldots), T_3, T_4, l_2) \wedge happens(E_2, transport(\ldots), T_5, T_6, l_2) \wedge \\ T_1 \dot{\leq} T_3 \dot{\leq} T_4 \dot{\leq} T_2 \wedge T_1 \dot{\leq} T_5 \dot{\leq} T_6 \dot{\leq} T_2$$

As [Dav96] has shown, *DOMAAB* implements the fundamental concepts of a procedural programming language including concurrent statements, sequential statements, recursion, and even conditionals. For example, the *search* macro can be procedurally refined as

$$\tilde{\exists} holds(at(Rob, Area_1), T_1, T_2, l_3) \wedge \\ holds(ahead(Rob, Box), T_1, T_2, l_3) \\ \vee \tilde{\exists} happens(E_1, moveArea(Rob, Area_1), T_1, T_3, l_4) \wedge \\ happens(E_2, look(Rob, Box, Area_0, Area_1), T_4, T_2, l_3) \wedge \\ T_1 \dot{\leq} T_3 \dot{\leq} T_4$$

where *moveArea* and *look* are lower-level navigation 'routines'.
This property of *DOMAAB*, namely the treatment of plans or narratives as procedures, is the key to specify the complex intentions of agents. This is of course not too surprising, since the definitions just lift the expressiveness of underlying logic programming. One may argue that the use of *HEC* is therefore a trivialisation to the general application of logic programming to agent design. As already argued in [McC63], the difference is that a logic 'procedure' and a logical 'application of the procedure' are now represented as reified terms of our theory of time and action and hence subject to ongoing reasoning about explicit causal and temporal relationships. This holds for the prediction of abstract situations from given observations such as needed to build a knowledge base module. This holds for the task of a planning module to synthesise an intention from designer-given pieces of behaviour. And this holds for the plan's on-line interpretation in interaction with the environment within intention execution modules.
The final part of a *HEC* specification are the overall goals to be achieved which are defined as a set of conservative *holds* expressions (sceptical mode $\dot{0}$, causal chain *nil*, 'dummy' consumer E_1, E_2) referring to different levels of abstraction

$$G ::= holds(at(box_1, shelf_3), t_7, t_8, nil, \dot{0}, E_1, l_2) \wedge \\ holds(at(box_1, shelf_3), t_7, t_8, nil, \dot{0}, E_2, l_3)$$

We then derive the framework $DOM \wedge I \wedge \Delta \models_{HEC} \tilde{\exists} G$.

4.3.4 Well-Definedness and Other Properties

As we have demonstrated in the case of *ECJ*, it is important to establish the computational tractability of *HEC* with respect to partially undefined models. Since the technique of dealing with mutual dependencies has been carried over to *HEC*, the arguments and proofs that partially moved into Appendix B are similar, if not identical. First, we show that undefined predicates result in an infinite sequence of undefined *holds* values incrementally building up a causal chain. The events referred in that sequence are introduced via *happens* facts in the narrative Δ (Proposition 6). Then, the appearance of a causal cycle and the definedness of some intermediate *holds* value, hence the collapse of the infinite sequence can be shown.

This construction also carries over to the worst-case behaviour of *HEC* (Theorem 4) from which we finally derive in Theorem 5 that dual fluents are treated as intuitively expected: It is not possible to demonstrate the persistence of both a fluent and its dual within the same pessimistic context.

Proposition 6 (Existence of Infinite Sequences in *HEC*) *Let M be a minimal three-valued (Herbrand) model for HEC under I, Δ, DOM:*

$$M \models_3 Comp(HEC1 \wedge \ldots \wedge HEC5 \wedge ECJ3 \wedge ECJ4 \wedge \Delta \wedge I \wedge DOM) \wedge \\ CET \wedge ECK3 \wedge ECK4 \wedge HEC6 \wedge \ldots \wedge HEC12$$

For all C : Constant; $n : \mathbb{N}$, it holds $0.5 \in ran\, IP(C, n)$ if and only if there exists an infinite sequence $i : \mathbb{N}$; $U_{i,1}, \ldots, U_{i,8} : U$ such that $IP(holds, 7)(U_{i,1}, \ldots, U_{i,7}) = 0.5$, $U_{i+1,4} = IF(cons, 2)(U_{i,6}, U_{i,4})$, and $IP(happens, 5)(U_{i,6}, U_{i,8}, U_{i,2}, U_{i,3}, U_{i,7}) = 1.0$.

Proof. see Appendix B. □

Proposition 7 (Three-Valued Minimal Models of *HEC*) *Any minimal three-valued (Hebrand) model M of HEC is already a two valued model.*

Proof. see Appendix B. □

Theorem 3 (Unique Minimal Two-Valued Model of *HEC*) *HEC′ has a unique minimal two-valued (Hebrand) model*

$$M \models_2 Comp(HEC1 \wedge \ldots \wedge HEC5 \wedge ECJ3 \wedge ECJ4 \wedge \Delta \wedge I \wedge DOM) \wedge \\ CET \wedge ECK3 \wedge ECK4 \wedge HEC6 \wedge \ldots \wedge HEC12$$

Theorem 4 (Treatment of Worst Case in *HEC*) *Let M be the minimal (Herbrand) model of HEC:*

$$M \models \tilde{\forall} holds(F, T_1, T_2, C, \dot{0}, E, L) \supset holds(F, T_1, T_2, C, \dot{1}, E, L)$$

Theorem 5 (Treatment of Dual Fluents) *Let M be the minimal (Herbrand) model of HEC:*

$$M \models \tilde{\forall} holds(F, T_1, T_2, C, \dot{0}, E, L) \wedge Dual(F, F_-) \supset \neg holds(F_-, T_1, T_2, C, \dot{0}, E, L)$$

Proof. see Appendix B. □

4.4 Bottom Line

A common declarative framework for describing the reasoning of unified agents as well as of particular processes inside a hybrid agent has to care about the representation of fluents, time, actions, and their inherent causal relationships possibly in an 'executable' first-order logic. This section developed the Hierarchical

Event Calculus as an expressive theory that is derived from the narrative-based formalisms of [KS86, Sha97a, Dav96]. Like the calculi of [LRL97, Sha97a, Dav96], *HEC* reifies a procedural sub-language which is able to synthesise and analyse the complex intentions of agents, such as behaviours, plans, and protocols. Unlike [LRL97, Sha97a, Dav96], *HEC* explicitely deals with multiple levels of abstraction that incorporate macro events with own duration, own effects, and own preconditions. This is to address the foremost requirement of situatedness which is the making of approximate inferences and decisions.
For this purpose, *HEC* relies on standard logic programming for broad implementability and exhibits useful properties, such as well-definedness, in reasoning about incomplete information. Just as *SC* is regarded as the theory behind state-space planning, just as the Event Calculus has been shown to declaratively mirror the computation of partial-order planning à la UCPOP [Wel94], *HEC* provides a formal basis for expressing the abstraction planning of, e.g., Hierarchical Transition Networks [EHN94], and, in general, for expressing all the InteRRaP-R processes, such as mental model, reflex, and protocol execution in an inferential setting. A suitable inference framework for *HEC* shall be developed in in the following.

5 Inference: ALP

Model theory is the prime tool for designing and analysing logic specifications such as the Hierarchical Event Calculus. Computational logics also rely on their proof-theoretic, *inferential* semantics that describes how to trace consequences and explanations of a given theory in a systematic fashion, e.g., how to 'execute' *HEC* for realising various reasoning processes. Proof theory has even been propagated as a partial substitute for model theory [Kow95]. Usually, however, proof and inference procedures are evaluated with respect to *soundness* and *completeness* for a given model theory; sound procedures only accept statements which are valid in the model-theoretic sense. Complete procedures are able to eventually derive all valid statements.

5.1 Logic Programming and SLDNF

The problem of determining the satisfiability of a first-order formula, i.e., to derive whether for a given F : *Wff* there exists a model M and an assignment V such that $M, V \models F$, is semi-decidable: We can only give procedures which terminate if a formula is unsatisfiable or valid, but do not necessarily halt if the formula is satisfiable and not valid. Early proof procedures by Davis & Putnam [DP60] and Robinson [Rob65] were building on the work of Herbrand [Her67] who showed remarkable connections between syntactical manipulations on first-order formulae and their semantics.
It was especially due to the *resolution* principle of Robinson that the automation of first-order logic became an attractive and fruitful field of research. Resolution can be coded as a relation $\vdash$: *Wff* $\leftrightarrow$ *Wff* that describes a set of possible computation steps starting from an initial formula F : *Wff* and successfully ending if $F \vdash^{\star} \bot \wedge F_1$ where the transitive hull $\vdash^{\star}$ enumerates all derivations that are according to the resolution principle. Resolution incorporates the following particular inference steps AND + FAL + TRU + RES + EXI + ALL which partially *substitute* the quantified variables

(Definition 8). For reasons of simplicity, we have omitted to establish the associativity and commutativity of $\vee$ and $\wedge$.

Definition 8 (Substitution) *A substitution* σ : *Variable* $\rightarrow$ *Term is a mapping from variables to terms. The application of a substitution* σ *to a term Te,* $Te\sigma$*, is inductively defined for all Va* : *Variable*; C : *Constant*; $Te_1, \ldots, Te_n$: *Term as*

$$Va\sigma = \sigma(Va) \qquad \text{and} \qquad C(Te_1, \ldots, Te_n)\sigma = C(Te_1\sigma, \ldots, Te_n\sigma)$$

$$\frac{(\neg(F_1 \vee F_2) \vee F_3) \wedge F_4}{(\neg F_1 \vee F_3) \wedge (\neg F_2 \vee F_3) \wedge F_4}\ \text{AND} \qquad \frac{(\exists\, Va.F_1 \vee F_2) \wedge F_3 \quad C \text{ not in } F_1, F_2, F_3}{(F_1(Va \mapsto C) \vee F_2) \wedge F_3}\ \text{EXI}$$

$$\frac{(\neg \exists\, Va.F_1 \vee F_2) \wedge F_3 \quad \mathcal{V}(Te) = \varnothing}{(\neg \exists\, Va.F_1 \vee F_2) \wedge (\neg F_1(Va \mapsto Te) \vee F_2) \wedge F_3}\ \text{ALL}$$

$$\frac{(\bot \vee F_1) \wedge F_2}{F_1 \wedge F_2}\ \text{FAL} \qquad\qquad \frac{\neg\bot \wedge F_1}{F_1}\ \text{TRU}$$

$$\frac{(\bigvee_i C_i(Te_{i,1}, \ldots, Te_{i,n_i}) \vee F_1) \wedge (\bigvee_i \neg C_i(Te_{i,1}, \ldots, Te_{i,n_i}) \vee F_2) \wedge F_3}{(\bigvee_i C_i(Te_{i,1}, \ldots, Te_{i,n_i}) \vee F_1) \wedge (\bigvee_i \neg C_i(Te_{i,1}, \ldots, Te_{i,n_i}) \vee F_2) \wedge (F_1 \vee F_2) \wedge F_3}\ \text{RES}$$

Resolution is a sound *refutation* procedure. If $\bot$ appears in the top-level conjunct of a derived formula, we know that it is unsatisfiable, hence this must hold for all intermediately computed formulae and thus the initial formula, too. To demonstrate that a particular statement is valid, we demonstrate the unsatisfiability of its negation by using the resolution principle. Resolution is a complete refutation procedure by eventually deriving $\bot$ from any unsatisfiable formula [Fit90].

But these are just in-principle statements because $\vdash$ does not immediately lead to a deterministic algorithm. One measure for improvement is to switch to a unification-based approach [Rob65, Sie89, LMM88] which delays the non-deterministic choice of the right variable replacement in ALL until it is really needed to perform RES. An *unification* algorithm is deployed to derive the *most general unifier* that renders two literals dual and enables the application of the RES step. First-order unification is decidable; early algorithms were specified recursively with exponential complexity. Recent developments have shown that it can be done in quasi-linear time [MM82].

Another measure is to choose more convenient canonical representations, such as normal logic programs, and more appropriate inference services, such as the checking of entailment $P \models \tilde{\exists} G$. For example, if we restrict to positive literals in the body of clauses and the goal (*definite logic programs*), all rewritings can fully concentrate on the goal expression $\vdash$: *Program* $\rightarrow$ *Goal* $\leftrightarrow$ *Goal* and can be captured within a single inference step. In the following, we abbreviate $\vdash (P)$ to $\vdash_P$, let σ : *Variable* $\rightarrow$ *Term* be a most general unifier, and suppose $\pi(P)$ to be a variant of the program P that is an equivalent formula, but has all variables 'renamed apart'.

$$\frac{C(Te_1, \ldots, Te_n) \wedge G_1 \quad \pi(P) \ni \tilde{\forall} C(Te'_1, \ldots, Te'_n) \subset \tilde{\exists} G_2 \quad Te_j\sigma = Te'_j\sigma}{G_2\sigma \wedge G_1\sigma}\ \text{SLD}$$

This principle is called SLD (Linear Resolution with Selection Function on Definite Clauses) and is due to [KK71]. It is a sound specialisation of a unification-based RES step trying to refute $P \wedge \neg\tilde{\exists}G$. When deriving $G \vdash_P^\star \top$, we know that $P \wedge \neg\tilde{\exists}G$ is not satisfiable, hence $P \models \tilde{\exists}G$.
By embanking the possible derivations, not every SLD-step eventually leads to the $\top$ goal, anymore. A complete SLD algorithm therefore has to simultaneously trace several optional rewritings $G_1 \vee \ldots \vee G_n$ that are scheduled by a *search rule* and build a *search tree* quite similar to our construction of inference processes from Section 3.
SLD has been shown complete if it either terminates or its search rule eventually rewrites all the available options. Then, it is called *semi-fair*. This is independent of the *computation rule* that chooses the sub-goal to rewrite within the selected search option. We speak of *fairness* if the derivation terminates or the computation rule eventually rewrites all the literals. For these concepts, see, e.g., [Llo87].
The practical aspect of SLD comes at the price of expressiveness, here: the abandonment of negated literals. SLD has been extended to cover normal logic programs by treating negated literals in the goal as finite failures of embedded resolution proofs. 'Negation-as-Failure' (NF) that is safe for *ground* literals, i.e., those containing no variables, has been theoretically investigated by Clark [Cla78] in connection with the already presented completion semantics. In the following, $\vdash_P^n$ captures all derivations of length $n : \mathbb{N}$.

$$\frac{\neg C(Te_1, \ldots, Te_n) \wedge G_1 \quad C(Te_1, \ldots, Te_n) \vdash_P^n \{\bot \wedge \ldots\} \quad \mathcal{V}(Te_j) = \varnothing}{G_1}\ \text{NF1}$$

$$\frac{\neg C(Te_1, \ldots, Te_n) \wedge G_1 \quad C(Te_1, \ldots, Te_n) \vdash_P^n \top \quad \mathcal{V}(Te_j) = \varnothing}{\bot}\ \text{NF2}$$

SLDNF::=SLD+NF1+NF2 has been shown sound with respect to inferring entailment from $Comp(P) \wedge CET$ [Cla78]. Given fairness, [CL89] also demonstrated its completeness under *strict* queries for which, alternatively, a three-valued fix-point semantics has been given by [Kun87]. In contrast to SLD, semi-fairness is no longer enough for building a complete algorithm: We could imagine computation rules which hinder the construction of finite failures, hence the existence of a finite depth of the nested proof.
The expressiveness of normal logic programs under SLDNF matches the one of Turing machines. This has rendered SLDNF the prime inference principle for logic programming. It has been practically applied to a range of applications, especially in prototyping problem solving for, e.g., databases, linguistics, scheduling, and configuration applications that are beyond the scope of traditional imperative or object-oriented programming. It has initiated a highly active and influential field of research that investigates the trade-off between first-order expressiveness and effectiveness.

5.2 Abductive Logic Programming and IFF

In several problem solving areas, such as diagnostics, two particular drawbacks of NF are its closed-world assumption and its rigid safety condition. For example, we do not always want to minimise lacking information within the system, such as observable symptoms and their medical reasons. Rather we would like to derive all diagnoses

that lead (inter alia) to the observed symptoms. Some of these are minimal and trivial, but some of these could be complicated and serious. Moreover, we do not want the inference procedure to get stuck because of being ignorant about the colour of the patient's rash. Instead we would like to gain a proposed instantiation of the colour depending on the current hypothesis, such as "Influenza is one possible diagnosis, if the rash is not red."

An inference service that extends the completion entailment of normal logic programs is the 'Abductive Logic Programming' (ALP) framework of [KKT93]. So far, we dealt with *deductive* inferences demonstrating that some formula G logically follows from a background theory: $Comp(P) \wedge CET \models \tilde{\exists} G$. *Abduction* additionally looks for possible factual extensions Δ of the background theory which explain the desired or observed goal: $Comp(P \wedge \Delta) \wedge CET \models \tilde{\exists} G$. It can be shown that abductive assumptions about a limited set of predicates, so-called *abducibles*, are just a special version of a closed-world assumption [TK95] and can be realised using ordinary SLDNF. This form of ALP is closely related to the minimisation policies of *default logics* [Rei80].

In order to have a direct access to hypotheses and to cater for non-ground negative literals, special ALP procedures have been developed [CDT91, DS92, JFB96a]. The perhaps most comprehensive account that is based on the clausal, factual, and constraint representation of Figure 21 is given by the IFF proof procedure [FK97]:

$$\frac{\neg\tilde{\exists} G \ldots}{(\tilde{\forall} G \supset \tilde{\exists}\bot)\ldots}\ \text{SMP1} \qquad \frac{\tilde{\forall}(F_1 \wedge \neg\tilde{\exists} G) \supset F_2 \ldots}{\tilde{\forall} F_1 \supset (F_2 \vee \tilde{\exists} G)\ldots}\ \text{SMP2} \qquad \frac{(\tilde{\forall}\bot \supset F_1) \wedge F_2 \ldots}{F_2 \ldots}\ \text{SMP3}$$

$$\frac{C(Te_1,\ldots,Te_n) \wedge F_1 \ldots \quad \pi(P) \ni \tilde{\forall} C(X_1,\ldots,X_n) \equiv \bigvee_i \tilde{\exists} G_i \quad X_j\sigma = Te_j}{\bigvee_i (G_i\sigma \wedge F_1)\ldots}\ \text{UNF1}$$

$$\frac{\tilde{\forall}(C(Te_1,\ldots,Te_n) \wedge G) \supset F_2 \ldots \quad \begin{array}{c}\pi(P) \ni \\ \tilde{\forall} C(X_1,\ldots,X_n) \equiv \bigvee_i \tilde{\exists} G_i\end{array} \quad X_j\sigma = Te_j}{\bigwedge_i (\tilde{\forall}(G_i\sigma \wedge G) \supset F_2)\ldots}\ \text{UNF2}$$

$$\frac{(F_1 \wedge F_2) \vee F_3 \quad F_1 = C(Te_1,\ldots,Te_n) \wedge (\tilde{\forall}(C(S_1,\ldots,S_n) \wedge G) \supset F_4)}{((\tilde{\forall}(\bigwedge_{j\in\{1,\ldots,n\}} S_j \doteq Te_j \wedge G) \supset F_4) \wedge F_1 \wedge F_2) \vee F_3)}\ \text{PRP}$$

$$\frac{(\tilde{\forall} G \supset (\tilde{\exists} C(Te_1,\ldots,Te_n) \vee F_1) \wedge F_2) \vee F_3 \quad \mathcal{V}(Te_j) \cap \mathcal{V}(G) = \varnothing}{(C(Te_1,\ldots,Te_n) \wedge F_2) \vee ((\tilde{\forall} G \supset F_1) \wedge F_2) \vee F_3}\ \text{FIR}$$

$$\frac{(C(Te_1,\ldots,Te_n) \wedge F_1) \vee F_2 \quad F_1 = C(S_1,\ldots,S_n) \wedge F_3}{(C(Te_1,\ldots,Te_n) \wedge F_1 \wedge \tilde{\forall} \bigwedge_{j\in\{1,\ldots,n\}} S_j \doteq Te_j \supset \tilde{\exists}\bot) \vee (F_1 \bigwedge_{j\in\{1,\ldots,n\}} S_j \doteq Te_j) \vee F_2}\ \text{FCT}$$

Starting from an initial formula $G \wedge IC$, IFF applies a number of equivalence rewritings $\bigvee_i G_i \wedge \Delta_i \wedge IC_i \vdash_P \bigvee_j G'_j \wedge \Delta'_j \wedge IC'_j$. These include simplification (SMP), the unfolding (resolving) of goals and constraint bodies (UNF1 and UNF2), the 'propagation' of information into constraints (PRP), the splitting ('firing') of constraints (FIR), and the condension of facts through factoring (FCT). Related to the 'Negation-as-Inconsistency' principle of [GS86], SMP1 transforms negated goals $\neg\tilde{\exists} G$ into a particular type of constraints $\tilde{\forall} G \supset \tilde{\exists}\bot$ that are elaborated interleaved to the other formulae.

IFF also employs a constraint-sensitive version (EQU) of the unification mechanism of [MM82] of which we leave out some of the details. IFF additionally introduces a case analysis (CAS) of the equalities appearing in constraint bodies:

$$\frac{Te \doteq Te \wedge F_1 \ldots}{F_1 \ldots}\ \text{EQU1} \qquad \frac{C(Te_1,\ldots,Te_n) \doteq C(S_1,\ldots,S_n) \ldots}{\bigwedge_{j\in\{1,\ldots,n\}} Te_j \doteq S_j \ldots}\ \text{EQU2}$$

$$\frac{(X \doteq Te \wedge F_1) \vee F_2}{(X \doteq Te \wedge F_1(X \mapsto Te)) \vee F_2}\ \text{EQU3} \qquad \ldots$$

$$\frac{((\tilde{\forall}(G \wedge X \doteq Te) \supset F_1) \wedge F_2) \vee F_3 \quad X \notin \mathcal{V}(Te)}{((\tilde{\forall} G \supset F_1) \wedge X \doteq Te \wedge F_2) \vee (\neg\tilde{\exists} X \doteq Te \wedge F_2) \vee F_3}\ \text{CAS}$$

A successful derivation $G \wedge IC \vdash_P^* (\bigwedge_j X_j \doteq Te_j \wedge \Delta' \wedge IC') \vee F$ produces a disjunct in which, besides a set of equalities and constraints, only positive abducibles Δ' : *Fact* (the so-called *residue*) appear. From such a disjunct, an answer hypothesis σ; Δ can be constructed as follows: Let σ be a substitution such that $\bigwedge_j X_j\sigma = Te_j$ hold, such that the inequalities in IC' (constraints of the form $\tilde{\forall} X \doteq Te \supset \tilde{\exists}\bot$) are implied and such that σ assigns ground terms ($\mathcal{V}(Te) = \varnothing$) to all variables in Δ'. We then let $\Delta = \Delta'\sigma$ and write $\vdash_{P,IC}^{\sigma,\Delta} G$. This construction can be generalised to any intermediate state of a proof:

Definition 9 (Intermediate Substitutions and Hypotheses) *Let*

$$F ::= \bigwedge_j S_j \doteq Te_j \wedge IC \wedge \Delta \wedge G$$

be an intermediate disjunct derived by IFF. We define corresponding sets of intermediate substitutions $\mathcal{S}(F)$ and intermediate hypotheses $\mathcal{H}(F)$ as

$$\mathcal{S}(F) ::= \{\sigma \text{ a ground substitution } \mid \sigma \wedge CET \models IC \bigwedge_j S_j \doteq Te_j\} \quad \mathcal{H}(F) ::= \bigvee_{\sigma\in\mathcal{S}(F)} \Delta\sigma$$

Proposition 8 (Monotonicity of Intermediate Hypotheses) *For all $t : \mathbb{N}$ such that $G \wedge IC \vdash_P^t (G' \wedge \Delta' \wedge IC') \vee F_1$ and $(G' \wedge \Delta' \wedge IC') \vdash_P (G'' \wedge \Delta'' \wedge IC'') \vee F_2$, it holds that $\mathcal{S}(G'' \wedge \Delta'' \wedge IC'') \subseteq \mathcal{S}(G' \wedge \Delta' \wedge IC')$ and $\mathcal{H}(G'' \wedge \Delta'' \wedge IC'') \models \mathcal{H}(G' \wedge \Delta' \wedge IC')$.*

Proof. This can be seen from the way that IFF is treating abducibles. New abducibles can be added according to FIR and UNF, hereby extending the predecessor hypothesis. Two abducibles can only be merged (FCT) by establishing new equalities. □

IFF is sound in that it does not change the satisfiability of a formula according to three-valued completion semantics and the theoremhood of integrity. Moreover, IFF rewrites the formulae equivalently under the given background program and hence improves the completeness results of earlier proof procedures [CDT91, DS92, JFB96a]. Completeness for abductive inferences is reasonably restricted to the generation of 'plausible' or 'minimal' hypotheses. This is because we do not want to generate every possible diagnosis to a given observation: there are infinitely many of them which distinguish in fully problem-irrelevant details. Instead, for every possible diagnosis, we like to find at least one representative explanation that is a valid hypothesis by itself and, at the same time, comprises a subset of the original assumptions. This coincides with the philosophical stance of *Okham's Razor* and is technically realised in the following theorem (where substitutions and residues are sets of literals):

Theorem 6 (Soundness and Completeness of IFF) *The IFF procedure is sound and complete with respect to the three-valued completion semantics, i.e., for all P : Program and IC : Constraints; G : Goal, it derives* $\vdash_{P,IC}^{\sigma,\Delta} G$ *only if* $P, IC, \Delta \models G\sigma$. *For all* σ, Δ' *such that* $P, IC, \Delta' \models G\sigma$, *there exists a* $\Delta \subseteq \Delta'$ *such that* $\vdash_{P,IC}^{\sigma,\Delta} G$.

Proof. see [Fun96]. □

We recognise that IFF is as close to a traditional proof procedure as to a logic programming service. This is due to its rich language, the expressive power of 'Negation-as-Inconsistency', and the reintegration of all inference options into a single equivalence transformation. As such, some of the IFF steps can result in explosive operations, such as unfolding. IFF also keeps some of the problems related to negation and unfairness. For particular programs and constraints, appropriate selection rules can be given which, e.g., postpone explosive steps as long as possible, while still retaining fairness, hence completeness. Then, IFF can be practically realised upon standard LP. IFF has already been used for semantic query optimisation, for deductive database view updates, for constraint programming, and even for planning.
Especially in the latter context, ALP and IFF exhibit some outstanding feature for turning an inference service into a situated reasoning process. As a by-product of abductive completeness, IFF allows the incremental specification of problems, that is, at any time during a proof and for any valid hypothesis, we can find an intermediate result that is extendible to the hypothesis (Proposition 9). It follows that, even if we extend the problem specification by establishing new constraints, new goals, new instantiations, and new abducibles, completeness is not affected (Theorem 7). We will elaborate in the rest of this section that this property renders ALP and IFF a flexible basis for 'executing' a theory of time and action, such as *HEC*, within hybrid InteRRaP-R.

Proposition 9 (Existence of Intermediate Hypotheses) *If* $P \wedge \Delta, IC \models G\sigma$, *then for each* $t : \mathbb{N}$ *there exists* $G \wedge IC \vdash_P^t (G' \wedge \Delta' \wedge IC') \vee F$ *such that* $\sigma \in \mathcal{S}(G' \wedge \Delta' \wedge IC')$ *and* $\Delta \models \mathcal{H}(G' \wedge \Delta' \wedge IC')$.

Proof. Suppose a $t : \mathbb{N}$ such that $G \wedge IC \vdash^t \bigvee_i F_i$ and $\Delta \not\models \mathcal{H}(F_i)$ or $\sigma \notin \mathcal{S}(F_i)$. Then, by Proposition 8, this also holds for all $t' > t$. Because of Theorem 6, however, there must be some $t'' : \mathbb{N}$ and F such that $\mathcal{H}(F) \subseteq \Delta$, $\mathcal{S}(F) = \{\sigma\}$, and for all $t''' \geq t''$ there exists a F''' such that $G \wedge IC \vdash^{t'''} F \vee F'''$. This derives a contradiction. □

Theorem 7 (Incrementality of IFF) *For any* $t : \mathbb{N}$ *and* $P \wedge \Delta \wedge \Delta', IC \wedge IC' \models (G \wedge G')\sigma$, *there is* $\Delta \wedge G \wedge IC \vdash_P^t (\Delta'' \wedge G'' \wedge IC'') \vee F$ *such that* $\sigma \in \mathcal{S}(\Delta'' \wedge G'' \wedge IC'')$ *and* $\Delta \wedge \Delta' \models \mathcal{H}(\Delta'' \wedge G'' \wedge C'')$

Proof. First, we note that for any Δ, the procedure $\Delta \wedge G \wedge IC \vdash_P^{\sigma,\Delta\wedge\Delta'}$ is sound and complete with respect to inferring σ, Δ' such that $P \wedge \Delta \wedge \Delta', IC \models G\sigma$. This is because we could reduce its inference steps to those of the procedure $G \wedge IC \vdash_P^{\sigma,\Delta\wedge\Delta'}$ for which, by Proposition 9, intermediate hypotheses compatible with Δ exist (1).
Within a minimal model, the monotonicity of logic consequence still holds. Hence we have that $P \wedge \Delta \wedge \Delta', IC \wedge IC' \models (G \wedge G')\sigma \models G\sigma$ and furthermore $P \wedge \Delta \wedge \Delta', IC \wedge IC' \models G\sigma$ implies $P \wedge \Delta \wedge \Delta', IC \models G\sigma$. Thus, $\Delta \wedge \Delta'$ is a

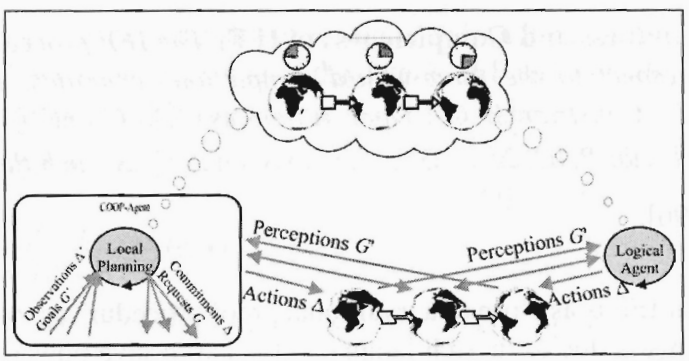

Figure 25: Abductive Inferences in Hybrid and Unified Agents

solution to the restricted problem for which, by (1), compatible intermediate results in $\Delta \wedge G \wedge IC \vdash_P^I$ exist. □

5.3 Local Planning: On-line Decision Making by ALP &*HEC*

To realise agent systems with the presented inferential LP machinery, Cognitive Robotics provides us with particular logic programs to be 'executed' (the core calculi) and particular data structures to be handled (e.g., fluents, events, and time). For example, using the Situation Calculus *SITK* as the logic program for deductive SLDNF — $\vdash^{\sigma}_{I \wedge DOM \wedge SITK}$ G — gives us a provably sound and complete algorithm for both analysing and synthesising linear plans. In this framework, fluents, situations, and plans are manipulated terms of the first-order language. The type of service, i.e., prediction versus planning, smoothly varies with the degree of instantiating G through the answer substitution σ. The resulting algorithm which we notate as $I \wedge DOM \vdash^{\sigma}_{SITK} G$ can easily imitate the behaviour of traditional state-space planners, such as STRIPS [McC85].
Using the Event Calculus *ECJ* under SLDNF — $I \wedge DOM \wedge \Delta \vdash^{\sigma}_{ECJ} G$ — establishes a sound and complete algorithm for analysing non-linear narratives according to the restrictive interpretation of partially-ordered solutions. For both calculi, we recognise that LP introduces a programmable, verifiable, and flexible logic surface (including, e.g., conditional and universal effects, partial specifications, deduced fluents, etc.) while bridging the gap to algorithmic issues.
Because plans in the Event Calculus are formulae, not terms, a decision making agent, such as the one in the right part of Figure 25, cannot be purely based on deductive inferences. Rather, its planning requires to make hypotheses Δ about the future course of the world. Similarly, perceptual observations G' do not immediately tell the agent what its environment is like, rather represent symptoms from which the agent must predict, how the world has been in the past and what it looks like presently. This is a quite diagnostic setting and, subsequently, the ALP framework has been proposed in [Esh88, Sha89, JFB96a] to execute Event-Calculus-style of calculi, such as $I \wedge DOM \vdash^{\sigma,\Delta}_{ECJ} G'$, for obtaining sound and complete partial-order planning procedures *à la* UCPOP [Wel94]. This considerably extends the range of the above deductive and situation-based approach [KS94].
Figure 25 illustrates that there is even a more fundamental motivation for using ALP-

based agent inferences: Other agents and environmental processes perform actions and inferences by themselves which leads to a stream of new observations and goals. These have to be incorporated in order to continuously derive rational and consistent commitments. Therefore we are also in a quite interactive setting and consequently, [KS96b] proposed the incremental IFF to handle the commitments and observations of a situated ALP-agent as on-line abducibles.

Equipped with a particular variant of the Event Calculus that is able to represent macro events [Dav96], Kowalski & Sadri's ALP-agent has been made programmable at the level of plans. Similar to the GOLOG approach [LRL97], the employed background theory however abolishes any proper axiom of causality to trace means and ends which is why the resulting agents primarily execute macros and do not really plan them. The rationale behind this simplification is to avoid the otherwise too complex logic reasoning for simultaneously predicting, planning, and executing within a dynamic environment.

The purpose of this section is to reinstall the full potential of Cognitive Robotics even for the case of broad agents. This is possible by our *HEC* theory supporting approximate reasoning about causality on several levels of abstraction. This is moreover possible by describing the overall agent not as a single proof procedure, but as a set of independent, but interacting inference processes. Each of those processes focuses on a particular prediction, planning, or execution task within a vertically modularised layer. Each of those layers focuses on a particular temporal and representational spectrum within the horizontally modularised agent.

The local planning process developed in this subsection serves as a prototype to obtain all other agent processes in Subsection 5.4, such as knowledge base processes and intention execution processes. In Subsection 5.5, we will then discuss the mainly representational issues which allow to distinguish BBL, LPL, and SPL of InteRRaP-R.

5.3.1 Hierarchical Partial-Order Planning

The encoding of planning problems into the Event Calculus comprises an initial situation I, a domain axiomatisation DOM, a goal expression G, and a plan representation Δ that includes *happens* and $\dot{<}$ facts and should be the residue of an abductive proof procedure.

In the Hierarchical Event Calculus, all of these formulae now contain expressions that refer to various levels of abstraction. To define what constitutes an Abstraction Planning Problem (APlP) in our formal language along [Sac74], we use the notation $F(\{l_0, \ldots, l_n\})$ to extract those definitions and facts out of a formula F which refer to a particular set of level of abstraction: We are then interested in a sequence of incrementally refined plans down to the most concrete level of representation, but not necessarily starting with the most abstract level, all of which solve intermediate planning problems, i.e., the related goals logically follow from an appropriate portion of the initial situation, the domain description, and *HEC*. To this end, we chose $PlHEC ::= HEC \setminus \{HEC8, HEC9\}$ as the sub-theory of *HEC* that focuses on refinement, not the construction of abstractions and otherwise keeps all the properties shown in Section 4.

Definition 10 (APIP) *Let $\mathcal{L} ::= \{l_0, l_1, \ldots, l_n\}$ be a set of incremental levels of abstraction. Let I, DOM, G be an initial situation, a domain axiomatisation, and a goal. The Abstraction Planning Problem $APIP(I, DOM, G, \mathcal{L})$ is the problem of finding an $i : \mathbb{N}$ with $i \leq n$ and a series of incrementally refined plans and substitutions $\{(\Delta_j, \sigma_j) \mid i \leq j \leq n\}$. For all $i \leq j < n$, it must hold that σ_j is identical to σ_{j+1} restricted to variables in $G(\{l_0, \ldots, l_j\})$ and $\Delta_{j+1}(\{l_0, \ldots, l_j\}) = \Delta_j(\{l_0, \ldots, l_j\})$. Moreover, it holds*

$$I(\{l_0, \ldots, l_j\}) \wedge DOMCAU \wedge \Delta_j \wedge \Phi(\{l_j\}) \models_{PIHEC} G(\{l_i, \ldots, l_j\})\sigma_j$$

where Φ is the set of all 'decomposeInitially' and 'decomposeMacro' literals such that $I \wedge DOMSAB \wedge DOMAAB \wedge \Delta_n \models \Phi$.
The APIP is said to be solved completely if for all $\{(\Delta'_j, \sigma'_j) \mid i \leq j \leq n\}$ which solve the APIP, we can find a solution $\{(\Delta_j, \sigma_j)\}$ such that $\Delta_j(\{l_j\}) \subseteq \Delta'_j(\{l_j\})$ and $\sigma_j = \sigma'_j$.

In order for a particular level of abstraction to be consistent with *HEC*6 and *HEC*7, the conditions of the corresponding *decomposeMacro* and *decomposeInitially* operations on the next level of abstraction have to be valid, too. Hence, Definition 10 regards an abstract plan as correct if we can show the entailment of goals given all those literals in $\Phi(\{l_j\})$.
Using the IFF proof procedure as $\vdash_{PIHEC}$ immediately gives us a sound Abstraction Planning Algorithm (APIA). This and the following results hold if either employing the three-valued or the two-valued completion semantics in Definition 10. This is because both coincide in the *HEC* case according to Theorem 3.

Theorem 8 (Sound APIA) *$I \wedge DOM \vdash^{\sigma,\Delta}_{PIHEC} G$ is a sound Abstraction Planning Algorithm (APIA) for solving an APIP $(I, DOM, G, \mathcal{L})$.*

Proof. In Definition 10, let $i = 0$. From the soundness in Theorem 6, we know that $I \wedge DOM \wedge \Delta \models_{PIHEC} G\sigma$ (1), and hence $I \wedge DOM \wedge \Delta$ entails all the constraints in *PIHEC* (2).
For all $j \leq n$, we let $\Delta_j ::= \Delta(\{l_0, \ldots, l_j\})$ and σ_j be σ restricted to variables in $G(\{l_0, \ldots, l_j\})$. Then we have $\Delta_{j+1}(\{l_0, \ldots, l_j\}) = \Delta_j = \Delta_j(\{l_0, \ldots, l_j\})$ and σ_{j+1} is identical to σ on variables from $G(\{l_0, \ldots, l_j\})$ which is in turn identical to σ_j on variables from $G(\{l_0, \ldots, l_j\})$.
In order for *decomposeInitially* and *decomposeMacro* literals to be true, they have to maximally refer to valid abducibles and facts on the next level of abstraction. By (2), we know that $I(\{l_0, \ldots, l_j\}) \wedge DOMCAU \wedge \Delta_j \wedge \Phi(\{l_j\})$ entails all constraints in *PIHEC*, in particular *HEC*6 and *HEC*7 (2).
From the construction of *HEC*, we know that a *holds* goal on a particular level of abstraction can be derived purely by *happens*, $\dot{<}$, *initially* statements on the same level, hence by (1) and (3):

$$I(\{l_0, \ldots, l_n\}) \wedge DOMCAU \wedge \Delta_j \wedge \Phi(\{l_j\}) \models_{PIHEC} G(\{l_i, \ldots, l_j\})\sigma_j$$

□

Completeness does not hold, because there may be solutions starting with $i > 0$, but none with $i = 0$. In this case, some upper level of abstraction is not solvable at all,

while more concrete ones are. Hence, the inferential service of Theorem 8 would not produce any result. More general, although we could be able to generate all the solutions starting with $i = 0$, this does not necessarily enumerate all the solutions starting at level l_1, l_2, etc., at the same time. The question is whether we could characterise settings in which this implication holds, hence when our inference service is complete. In [Yan90], two APlP properties are defined that relate the solutions on neighbour levels of abstraction. The *Downward Solution Property* (DSP) states that each abstract solution is always refineable to a more concrete solution. The *Upward Solution Property* (USP) states that each concrete solution always has a more abstract version that also solves the more abstract problem. Their formalisation can now be restated within our logic.

Definition 11 (Downward Solution Property) *The Downward Solution Property (DSP) holds for an APlP$(I, DOM, G, \mathcal{L})$ if and only if for all $l, l' : \mathcal{L}$; Δ; σ with l and l' subsequent levels of abstraction and $I \wedge DOM \wedge \Delta \models_{PlHEC} G(\{l\})\sigma$, there are Δ'; σ' such that $I \wedge DOM \wedge \Delta' \models_{PlHEC} G(\{l, l'\})\sigma'$, $\Delta'(\{l_0, \ldots, l\}) = \Delta(\{l_0, \ldots, l\})$, $\Delta'(\{l', \ldots, l_n\}) \supseteq \Delta(\{l', \ldots, l_n\})$, and σ' restricted to variables in $G(\{l_0, \ldots, l\})$ is identical to σ.*

Definition 12 (Upward Solution Property) *The Upward Solution Property (USP) holds for an APlP$(I, DOM, G, \mathcal{L})$ if and only if for all $l, l' : \mathcal{L}$; Δ; σ with l and l' subsequent levels of abstraction and $I \wedge DOM \wedge \Delta \models_{PlHEC} G(\{l'\})\sigma$, there exist Δ'; σ' such that $I \wedge DOM \wedge \Delta' \models_{PlHEC} G(\{l, l'\})\sigma'$, $\Delta'(\{l', \ldots, l_n\}) = \Delta(\{l', \ldots, l_n\})$, $\Delta'(\{l_0, \ldots, l\}) \supseteq \Delta(\{l_0, \ldots, l\})$, and σ' restricted to variables from $G(\{l', \ldots, l_n\})$ is identical to σ.*

The Downward Solution Property is useful for improving the selection of available options. If we are just interested in finding any solution, then we can fully commit to the assumptions , i.e., plans, that are created at an upper level of abstraction, because these are guaranteed to be refineable to a concrete solution (by induction over the levels of abstraction and using Definition 11). We will come back to the issue of selection in a minute.
For obtaining completeness, the Upward Solution Property is much more important since ensuring that any solution plan to the APlP is extendible into one that starts from $i = 0$. IFF then generates a minimal hypothesis that is provably contained within the full plan and hence satisfies the conditions of Definition 10:

Theorem 9 (Complete APlA) *If the APlP$(I, DOM, G, \mathcal{L})$ exhibits the USP, then $I \wedge DOM \vdash^{\sigma,\Delta}_{PlHEC} G$ is a complete APlA.*

Proof. Suppose, we have a solution $\{(\Delta_j, \sigma_j) \mid i \leq j \leq n\}$ to the APlP. Then, by Definition 10, $\Delta_n(\{l_0, \ldots, l_j\}) = \Delta_j(\{l_0, \ldots, l_j\})$ and σ_n restricted to variables from $G(\{l_0, \ldots, l_j\})$ is identical to σ_j. Furthermore, we know that $I \wedge DOMCAU \wedge \Delta_n \models \Phi$ and $\Phi \supseteq \Phi(\{l_n\})$, hence $I \wedge DOM \wedge \Delta_n \models_{PlHEC} G(\{l_i, \ldots, l_n\})\sigma_n \models G(\{l_i\})\sigma_n$. We can now repeatedly apply Definition 12 (with $l' = l_i, l_{i-1}, \ldots, l_0$) and arrive at some Δ', σ' such that $I \wedge DOM \wedge \Delta' \models_{PlHEC} G\sigma'$, $\Delta'(\{l_i, \ldots, l_j\}) = \Delta_n(\{l_i, \ldots, l_j\}) = \Delta_j(\{l_i, \ldots, l_j\})$, and σ' restricted to variables from $G(\{l_i, \ldots, l_j\})$ identical to σ_n, thus σ_j.

By the completeness of IFF that we have stated in Theorem 6, we are then able to derive $I \wedge DOM \vdash^{\Delta'',\sigma''}_{PIHEC} G$ such that $\Delta'' \subseteq \Delta'$ and $\sigma'' = \sigma'$. According to Theorem 8, this constructs a solution $\{(\Delta''_j, \sigma''_j) \mid i \leq j \leq n\}$ to the APIP$(I, DOM, G, \mathcal{L})$, where $\Delta''_j ::= \Delta''(\{l_0, \ldots, l_j\})$ and σ''_j is identical to σ'' restricted to variables in $G(\{l_0, \ldots, l_j\})$. Furthermore, it holds that $\Delta''_j(\{l_j\}) = \Delta''(\{l_j\}) \subseteq \Delta'(\{l_j\}) = \Delta_n(\{l_j\} = \Delta_j(\{l_j\})$ and σ''_j restricted to variables in $G(\{l_j\})$ coincides with $\sigma'' = \sigma'$. Since σ' coincides with σ_n upon variables from $G(\{l_j\})$ and since σ_n coincides with σ_j, we have shown completeness according to Definition 10. □

Human-inspired abstraction hierarchies, which are useful to approximate decisions, necessarily loose information. This makes DSP and USP hard to guarantee. While other macro formalisations, such as the one of [Sha97a], stick with the expressiveness (and even less) of Theorem 8, it is one of the achievements of *HEC* and Definition 10 that we do not rely on these properties[15].
For establishing a complete planning process in the absence of USP, the crucial observation is that failures on some high level of abstraction hinder the service of Theorem 8 to find a proper concrete solution. Such as in algorithmic approaches to hierarchical planning, the idea is to interleave 'planning from scratch' on a lower level of abstraction with the decomposition of results from the upper level. This is straightforwardly expressible in *HEC* and leads to a provably sound and complete APIA independently of the domain.

Theorem 10 (Sound And Complete APIA) $I \wedge DOM \vdash^{\sigma,\Delta}_{PIHEC} \bigvee_{l \in \mathcal{L}} G(\{l, \ldots, l_n\})$ *is a sound and complete APIA for solving an APIP* $(I, DOM, G, \mathcal{L})$.

Proof. Soundness follows the proof of Theorem 8 by showing that any result $\Delta;\ \sigma$ solves at least one of the disjuncts, hence $\Delta_j = \Delta(\{l_0, \ldots, l_j\})$ and σ_j identical to σ restricted to variables from $G(\{l_0, \ldots, l_j\})$ comprises a solution to the APIP starting at some level l.
Completeness follows from the fact that any solution $\Delta_j;\ \sigma_j$ starting with level l satisfies $I \wedge DOM \wedge \Delta_n \models_{PIHEC} G(\{l, \ldots, l_n\})\sigma_n \models \bigvee_{l \in \mathcal{L}} G(\{l, \ldots, l_n\})\sigma_n$ for which, by the completeness of IFF in Theorem 6, we can derive some $\Delta';\ \sigma'$ with $\Delta'(\{l_j\}) \subseteq \Delta_n(\{l_j\}) = \Delta_j(\{l_j\})$ and σ' restricted to variables from $G(\{l_j\})$ is identical to σ_n which in turn coincides with σ_j upon those variables. □

We have now derived a sound and complete APIA from IFF and *HEC* which provides for declaratively 'programmable' representations I and DOM. Algorithmic partial-order planning concepts such as causal links, threats, promotion, and demotion come as a by-product of the inferential semantics of the Event Calculus [Sha97a].
HEC extends this coincidence to techniques known from HTN planning [EHN94], such as *sharing*: Early sub-optimal APIAs decomposed distinct macros into disjoint sub-events. For example, if a forklift performs two sequential deliveries, one from the truck to a shelf and one in the opposite direction, the movement to the shelf should not be planned twice being already part

[15] *HEC*9 is weaker than the USP because allowing for 'phenomenal' higher-level fluents without a proper decomposition. These are needed for, e.g., imposing an absolute coordinate system onto the Automated Loading Dock — see Subsection 5.5 and Section 7.

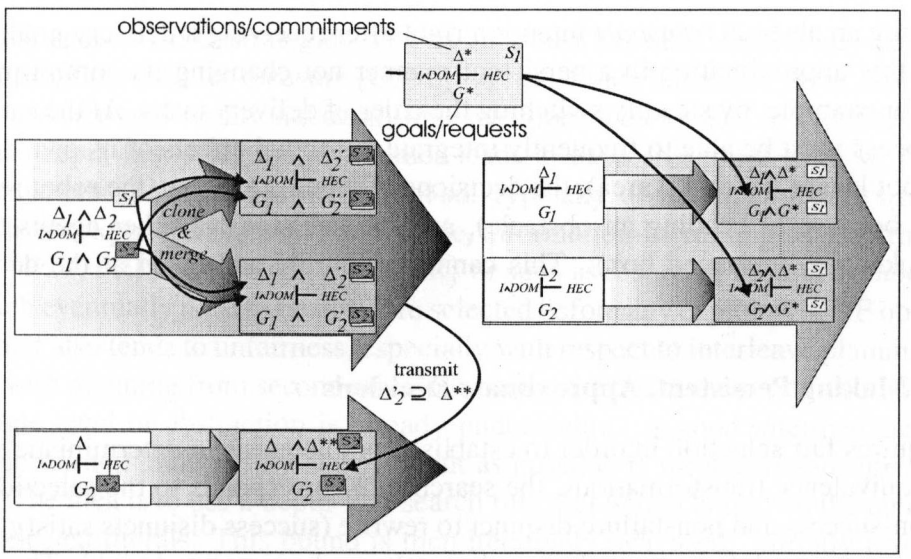

Figure 26: Shared Memory and Signalling between Abductive Inference Processes

of the first delivery. Under IFF, sharing is a most natural consequence of the FCT step unifying abducibles $happens(E_1, moveArea(rob_1, shelf_3), T_1, T_2, l_4) \wedge happens(E_2, moveArea(rob_1, shelf_3), T_3, T_4, l_4)$ from different *transport* macros.

Due to its incrementality (Theorem 7), this abductive planner furthermore nicely coincides with the continuous process model of COOP by identifying $\vdash(local\_planning) ::= \vdash_{PlHEC}$ (Figure 26). The state of the resulting planning process corresponds to a formula derived by IFF under *PlHEC*, *I*, and *DOM*. Different *Option*s of the process state correspond to the disjuncts of the intermediate formula. The *Program* part of each process option consists of the residue, i.e., a narrative of *happens*, and $\dot{<}$ facts, in the corresponding disjunct. Its *Goal* part is comprised by the rest of the disjunct, in particular by *holds* statements and substitutions that describe partial world situations in past, present, and future.

Accordingly, signals and shared memory exchanged between this planning process and other processes during runtime bidirectionally transmit abducibles, situation requests, and substitutions. But note that the (logic) separation into*Program* and *Goal* is orthogonal to the (functional) separation into agent believes and agent goals: Believes can be transferred both by *Program* in terms of made observations and in fixing a particular *Goal* in terms of unavoidable situations or predictions. Agent goals can be specified both by *Goal* in terms of desired situations and by *Program* in the form of committed occurrences of actions. This means that the input streamed into our logic-based planning process covers goal requests and state information, at the same time observed and predicted actions and conditions. Its output covers solution plans and at the same time instantiations and sub-services that the planner requests to be solved externally on its behalf.

Although compatible in form, this does not mean that our logic-based planner already already behaves reasonably as a continuous process that produces decisions for, e.g., forklift robots in the Automated Loading Dock. From any such computation within InteRRaP-R, we expect interactiveness with its environment and other processes: It must be able to issue approximate solutions rather early, such as the planning process

outputting an abstract *transport* intention right after receiving a delivery goal. It must realise this approximation in a persistent manner not changing its commitments too often, for example, by steadily switching the order of delivery tasks. At the same time, any process must be able to frequently integrate the actual effects of its own decisions (the robot leaves the initial area), the decisions of other processes (the robot reactively dodges because of heading an obstacle), and the decisions of other agents (another robot takes the envisaged box). This cannot be purely addressed at the declarative surface.

5.3.2 Making Persistent, Approximate Decisions

IFF requires fair selection in order to establish completeness in deterministic settings. In its equivalence transformations, the search rule corresponds to the selection of the next non-success and non-failure disjunct to rewrite (success disjuncts satisfy the condition of successful derivations, failure disjuncts can be simplified to $\bot$). The computation rule corresponds to the choice of the sub-formula of the selected disjunct that should be the subject of the next step of inference.
Fairness can be formalised by attaching indices to the literals of a formula and by introducing a system clock that measures the length of derivations. After each inference step, the clock is increased and newly introduced literals are annotated by increasing the maximal annotation of their selected predecessors. For example, we call the procedure semi-fair if and only if for all $F : \mathit{Wff}$, either $F \vdash^{\star}$ terminates or

$$\mathit{lim}_{t\to\infty}\, \mathit{min}\{i \mid F' \text{ a non-success disjunct such that } F \vdash^{t} F' \vee F'' \text{ and } i = \mathit{max}\{j \mid j \text{ the index of a literal in } F'\}\} = \infty$$

Fairness is not enough for an abstraction planning algorithm: Our presented inference process is able to issue output (signals carrying abducibles, such as $\mathit{happens}(e_1, \mathit{transport}(\mathit{rob}_1, \mathit{box}_1, \mathit{parking}, \mathit{truck}, \mathit{shelf}_3), t_1, t_2), l_2))$ as soon as it has solved one particular level of abstraction. However, the specification of *PlHEC* does not determine which goals at which levels of abstraction are to be resolved first. Using, e.g., a detailed navigation path to build up a *transport* macro may be a good idea for plan recognition (see Subsection 5.4), but not for quickly arriving at decisions.
Moreover, when looking at the inference service of Theorem 10, we would like to delay planning from scratch as long as possible: Hierarchical approaches (also: 'planning from second principles') are able to pull the planning complexity down to be exponential in the amount of abstraction levels rather than the greater, overall size of the narrative [RN95]. Hence, even if the convenient solution properties of Definitions 11 and 12 are not exhibited by the domain, starting with goals on higher levels of abstraction and incrementally working downwards, is a reasonable heuristics.
Practical agents must have relatively persistent believes, goals, and intentions. Hence, this must hold for their computations, too. In that respect, having search and computation rules that steadily switch options and priorities is certainly not a good idea. For example, a planning process that has proposed some abstract solution, such as first putting box_1 on the shelf and afterwards loading box_2 onto the truck, should stay with that commitment as long as possible in order to not confuse the connected execution process. Moreover, it is known in the planning literature that loosing a clear focus in elaborating conjunctive goals (unloading box_1, loading box_2) immediately blows up

the search space. LIFO-strategies (last-in, first-out) has been empirically confirmed as an appropriate counter-measure [GS96]. A rule which addresses these aspects in a logic-based setting is traditional depth-first selection, such as used in PROLOG.
However, depth-first selection sometimes leads to explosive proof trees while there would exist other selections which do not so. Typically, a *least-commitment* strategy is advisable for exponential problems: Simple, information-increasing steps in the proof, such as {FCT, PRP, EQU, CAS, SMP, FIR} or such as resolving ground *member* literals, which eventually have to be done are selected before any explosive UNF operation.
Depth-first also tends to unfairness, especially with respect to interleave planning from scratch with planning from second principles, since the the existence of a solution plan at a single level of abstraction is already undecidable. A good intermediate choice to uphold fairness, but to stay as persistent as possible is the *iterative-deepening* approach. The idea is to use a depth-first search rule, but with a bound on the annotation of the selected literals. This bound is then iteratively increased. Initially invented as a tool to diminish memory consumption, we use it as a tool to gain persistency while upholding completeness.
How could such bounds be designed in the case of ALP and *HEC*? In Theorem 3, we have shown the well-definedness of *HEC* that avoids 'floundering' of deductive inferences. This should now be investigated more closely within the abductive setting: Given a bound on the intermediate residues and provided a least-commitment strategy, IFF terminates upon *PIHEC* (Proposition 10).
Residue bounds have an intuitive interpretation in terms of the plausibility of a hypothesis. Usually, we are interested in the most plausible hypothesis, here a cost-effective plan. Hence, we can use such measures in a straightforward way to steer the selection of disjuncts, i.e., to realise the search rule. From the termination of IFF & *PIHEC* upon bounds and the existence of intermediate hypotheses (Proposition 9), it follows that any semi-fair least-commitment rule has some interpretation in the form of plausibility and vice versa, using plausibility to steer the selection amounts to a semi-fair and complete procedure (Theorem 11).

Proposition 10 (Termination) *Let $\mathcal{C} : \Delta \to \mathbb{N}^\infty$ be a strictly monotonic cost function on abducibles, i.e., $\Delta_1 \models \Delta_2$ only if $\mathcal{C}(\Delta_1) \geq \mathcal{C}(\Delta_2)$ and $\Delta_1 \models\!\!\models \Delta_2$ only if $\mathcal{C}(\Delta_1) = \mathcal{C}(\Delta_2)$. Then for any I; DOM; G, any search rule that never select disjuncts such that $\mathcal{C}(\mathcal{H}(F)) > k \in \mathbb{N}$ and any least-commitment computation rule that delays UNF until all other steps including the unfolding of member literals, $I \wedge DOM \vdash_{PIHEC} G$ terminates.*

Proof. see Appendix B. □

Theorem 11 (Selection Strategy) *Let $\mathcal{C}$ be any strictly monotonic cost function on abducibles, then any least-commitment APIA of Theorem 10 is semi-fair if and only if the APIA terminates or $lim_{t\to\infty} min\{\mathcal{C}(\mathcal{H}(F)) \mid F$ a non-success disjunct $\in ran \vdash'\} = \infty$ (1). Any semi-fair least-commitment APIA of Theorem 10 is complete (2).*

Proof. (1, only if): Suppose the least commitment APIA being semi-fair and non-terminating. Each non-success disjunct is then eventually selected for an infinite

amount of times. Because of Proposition 10, there cannot exist any upper bound on the cost of any disjunct. Hence the minimum of the residue costs must diverge.
(1, if) On the other hand, we know that any unfair search rule allows for a non-failure and non-success disjunct not being selected for an infinite amount of times which also implies its finite residue costs being constant. This gives an upper bound for the minimal residue costs.
(2, terminating case) By Proposition 9, we know that for any solution to the APIP, there must exist a compatible intermediate hypothesis in a non-failure node at any time during the proof. If the APIA terminates, there are only success and failure nodes left. Hence there must be some success node that comprises a compatible result according to Definition 10.
(2, non-terminating case) In the case of semi-fairness and non-termination, the minimal residue costs diverge over time (by 1). Hence, there is some point in time at which there exist no more non-failure and non-success nodes with a residue cost that is less or equal to the cost of any solution to the APIP. Proposition 9 then implies that there must be at least one success node at that time which is compatible with the solution and constructs a minimal hypothesis according to Definition 10. □

In the InteRRaP-R decision making processes, we chose a least-commitment, otherwise depth-first approach that iteratively deepens a domain-dependent cost measure $\mathcal{C}$ in favour of abstraction, i.e., macros are cheaper than their decomposition. A further strategic issue has to do with unfolding the bodies of constraints, such as the $\neg$*clipped* goals realising the persistence checks in *HEC*, because they could generate numerous options due to splitting (FIR) afterwards. In general, it is advisable to delay unfolding these constraints with respect to unfolding ordinary goals, since the latter centrally introduces new evidence. This evidence can be propagated into various constraints and hence restricts their possibilities. In partial-order planning, this is closely related to the LIFO-strategy trying to keep separate 'threads' of the plan independent.

5.3.3 Making Ego-Centred, Future-Oriented Decisions On-line

Approximate and persistent decisions is just one aspect of a local planning process whose results are concurrently executed by plan execution processes located at the same agent layer and simultaneously to whose processing, other agents and other, e.g., behavioural processes are computing and acting in the environment, thus changing the status of its companion mental model process (see Subsection 5.4). We require such a continuous planner to incrementally cope with new input in the form of COOP signals and shared memory operations. Such input covers observations Δ' and constraints IC' which come through dynamic changes of the world. Such input also covers additional goals and instantiations G' appearing during the course of planning.
ALP is an attractive device for this kind of situated reasoning because it does not rely on a closed world assumption with respect to abducibles Δ'. Theorem 7 has demonstrated that the incrementality of IFF moreover covers additional goals G' and constraints IC'. Hence, we can frequently incorporate signals and shared memory operations by extending the intermediate residues Δ, constraints IC, and goals G just as illustrated in Figure 26. This does not affect completeness, i.e., incompatible options will rewrite to $\bot$ while the extendible ones are kept.

The *HEC* surface does not distinguish between future decisions for the agent's own actions and past explanations for other agents' activity. It does not discriminate between wanted goals and unavoidable states. This is useful for allowing a flexible treatment of the above on-line phenomena. This is painful, on the other hand, because of denying an ego-centred and future-oriented decision making in the first place. We will now discuss how to trade off these aspects by operational considerations.

Integrating On-line Observations Useful observations Δ' incorporated during planning are the (macro) actions that the agent has actually executed itself. For example, a plan execution process of a forklift robot already starts decomposing a planner-committed *transport* macro by moving the robot towards the truck while the local planner is still computing landmarks towards the shelf. It is important that the planner is up-to-date with the commitments that the executer does, hence is reported $\Delta' ::= happens(e_1, moveArea(rob_1, truck), t_1, t_2, l_4)$.
In general, useful observations are all changes in the agent's believes which are relevant to the planning problem. Notice that, up to now, the initial situation I is fixed throughout the whole planning problem. Indeed, the fact that the agent believed in a certain fluent being valid, such as the location of box_1 on the truck, does not become invalid ever. Rather, the agent, or better the mental model process, has gained additional evidence, such as the other robot having already taken the box or being in the course of doing so. Getting signalled $\Delta' ::= happens(e_2, pickup(rob_2, box_1), t_3, t_4, l_3)$ influences the planner's prediction of present and future world states and forces it to adapt its decisions accordingly.

Integrating On-line Goals & Surprises Given new evidence, the mental model could also signal new goals. For example, a forklift's local planning process is able to adopt the command to deliver a particular box to the truck as the additional goal $G' ::= holds(at(box_3, truck), t_5, t_6, nil, \dot{0}, E, l_2)$.
Since goals and abducibles are logically coupled, the transmission of $G' ::= holds(not(at(box_1, truck))), t_7, t_8, nil, \dot{0}, E, l_3)$ is an alternative to sending a proper observation: Such a *surprise* forces IFF & *PlHEC* to find a diagnosis in the past, such as the $\Delta' ::= happens(e_2, pickup(rob_2, box_1), t_3, t_4, l_3)$ from the previous paragraph, which in turn adapts the future projection accordingly.

Integrating Miracles Diagnosis, here plan recognition, is an as expensive problem as planning and should be reasonably restricted to knowledge base processes. Its input to the planner should thus be given as a combination of Δ', G' in order to provide ready-to-propagate goals G', at the same time reduce the effort for hypothesising an explanation Δ'. Since agents are seldom to be equipped with a complete domain description, some evolvements rather appear as *miracles* which they cannot explain, but which they must face.
A pragmatic miracle formalisation is to introduce, in advance, for each relevant fluent an instantaneous dummy action type that unconditionally causes the fluent. Adding changes in the form of temporally annotated miracles Δ', we gather a redirected search without putting too much effort into explanations. Miracles should not be accessible to decision making, hence not be a subject to proper abduction. This is accomplished

by an operational restriction of the FCT step that enforces the unification of newly introduced miracles with already introduced ones.

Integrating Constraints for Future-Orientation In a continuous planner, the initial situation I always refers to the past. Hence, IFF & *PlHEC* allow for unrealistic agents that try to make decisions about the past. To synchronise the reasoned time-points with the external system clock, we assume both to be represented by the same logic vocabulary (for which we give a reasonable implementation in the next section). At time points t_i, we then add the additional constraint $IC' ::= \tilde{\forall} happens(E, A, T_1, T_2, L) \supset t_i \leq T_1$. This constraint must distinguish between past observations and future decisions by a corresponding restriction of PRP.

Ego-Centreing An additional operational asymmetry helps to establish a more ego-centred perspective for the planning process: The current formalisation takes a rather objective stance making decisions for other agents, too. This is needed to infer multi-agent plans which can be communicated, negotiated, and cooperatively executed. On the other hand, we want the agent to take over responsibility for its own capabilities. One effective measure is to let the residue costs$\mathcal{C}$ penalise other agent's actions, such as $pickup(rob_2, box_1)$ being more expensive than $pickup(rob_1, box_1)$. This is reasonable in terms of the implicitly present coordination efforts — see Subsection 5.5.3.

Handling Inconsistent and Unreachable Goals The incremental IFF deals with a major part of intermediate inconsistencies that are unavoidable in situated reasoning. Because of undecidability, it cannot detect inconsistent goals that steadily contribute to the residue and prohibit a proper solution. It does also not identify generally unreachable goals which must be abandoned in order to recover the inference process. With a bit of additional operational machinery, however, the most common cases can be handled at the level of COOP (scetion 3).
If a forklift is asked to cancel a previously commanded delivery, its logic-based planning process can be recovered by using exception handling: Whenever goals are incorporated into the planning process, an additional exception is created whose continuations contain *rolled-up* problem descriptions.
A *rolled-up* problem consists of deleting non-committed abducibles, compiling the residue into a new initial situation I, and deleting several predecessor goals. Cancelling a goal can then be done by adding the constraint $IC' ::= \forall holds(at(box_1, shelf_3), t_1, t_2, nil, \dot{0}, E, l_2) \supset \perp$ which immediately fails all the inconsistent options and invokes exception handling.
Thanks to an explicit representation of time, we reach a similar recovery effect by detecting open goals whose deadline has been past and by rewriting them to $\perp$.

Roll-Up and Hypothesising the Initial State Although on-line signals and shared memory operations enlarge the inference state at the first sight, we have documented that, provided some operational means, new evidence can focus situated search as to be similarly effective as traditional abstraction planning algorithms.
The roll-up mechanism plays an important role, because it does not only compile incremental specifications into computationally more handier descriptions, but allows

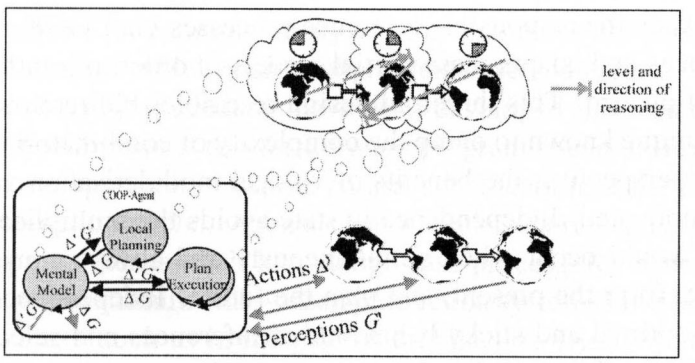

Figure 27: Inference Processes in the Vertically Modularised LPL

the agent to forget about past and future occurrences, about intermediate observations and goals. Without such a frequently applied 'mental garbage collection', agents and processes with life-cycles in the range of minutes, hours, days, and weeks would not be constructible.

Due to the incomplete temporal information that the agent accumulates, the roll-up of initial situations I must be necessarily incomplete, too. In general, we cannot expect to equip agents with a complete description at startup, such as the forklift robots with the complete state of the loading dock.

It is hence convenient to relax the closedness of I by making *initially* abducible, hence a part of Δ. This does not affect the theoretical properties of the APlA and of the upcoming processes. Rather, by the help of integrity constraints *HEC*7, *HEC*9, and *HEC*12, it allows forklift robots to search for a box, which they temporally assume to be located on the shelf, until they notice the contrary. Within a given domain, this activity is restricted to dedicated types of fluents, for example, an agent should not 'invent' arbitrarily many helping partner agents.

5.4 LPL: Vertically Interacting Processes

The preceding subsection developed an on-line decision procedure from ALP & *HEC* that comprises the local planning process of InteRRaP-R. We have presented operational restrictions which allow the otherwise too general logic to run practically and focused on a particular abstraction planning functionality.

With respect to a perception-action cycle, such as InteRRaP-R's LPL performs (Figure 27), planning is just an intermediate computation: Before the agent knows its present situation and what its current planning goals are, there must be first some computation which predicts the state of the world from made perceptions — the mental model process $\vdash(mental\_model)$.

And after the planner has outputted some approximate and abstract decision, there must be some computation which interprets those decisions, i.e., decomposes them into primitive actions — the plan execution processes $\vdash(plan\_execution)$. We will give ALP & *HEC* specifications of those functionalities in the following.

As Figure 27 illustrates, the resulting logic-based LPL hence simultaneously builds up several partial models of the world each by taking a different 'direction' of reasoning (from past to future or vice versa, from abstract to concrete or vice versa). The

interaction between the responsible inference processes via COOP signals and shared memory interfaces and adapts these partial models in order to jointly maintain a possibly consistent picture. This threefold design resembles bidirectional search, a quite traditional technique known to damp the complexity of combinatorial problems.
From the logic perspective, the benefits of vertical modularisation and encapsulation can now be remotivated: Independence in state avoids the multiplication of inference options which would occur when simultaneously and always consistently trying to explain the past, form the present, and plan the future. Independence in computation avoids the sup-optimal and sticky behaviour of inferences and selection rules which dive into one of these functionalities in order to produce a persistent output.

5.4.1 Mental Model: On-line Prediction using ALP & *HEC*

Declaratively, the task of a knowledge base relies on the same theoretical background as decision making, hence can be based on a variant of *HEC*. Inferentially, its function (the following Abstraction Prediction Problem, APrP, builds on traditional plan recognition [KP88]) is complementary: We are no longer trying to generate incrementally refined ego-centred plans Δ_i to enable wanted goals G. Rather, we are speculating all (incrementally more abstract) effects G_i of environmentally-invoked observations Δ. For this purpose, we identify $PrHEC ::= HEC \setminus \{HEC6, HEC7\}$ as the well-defined part of *HEC* that is concerned with building macro representations from primitives.

Definition 13 (APrP) *Let $\mathcal{L} ::= \{l_0, l_1, \ldots, l_n\}$ be a set of incremental levels of abstraction. Let I, DOM, Δ be an initial situation, a domain axiomatisation, and a narrative of observations.*
The Abstraction Prediction Problem $APrP(I, DOM, \Delta, \mathcal{L})$ is the problem of finding an extension of the narrative $\Delta' \supseteq \Delta$ and a series of incrementally more abstract goals and substitutions $\{(G_j, \sigma_j) \mid n \geq j\}$ such that $I \wedge DOM \wedge \Delta' \models_{PrHEC} G_j\sigma_j$, $G_j(\{l_n, \ldots, l_j\})\sigma_j \supseteq G_{j+1}(\{l_n, \ldots, l_j\})\sigma_{j+1}$, and σ_{j+1} is identical to σ_j restricted to variables in G_j. Moreover, it holds

$$\Phi(\{l_n, \ldots, l_j\}) \subseteq G_j\sigma_j$$

where Φ is any conjunction of ground holds literals that refer to Δ' and that have at most one duplicate entry in their causal chain such that $I \wedge DOM \wedge \Delta' \models_{PrHEC} \Phi$.
The APrP is said to be solved completely if for all Δ''; $\{G'_j, \sigma'_j \mid n \geq j \geq 0\}$ which solve the APrP, we can find a solution Δ'; $\{(G_j, \sigma_j)\}$, such that $\Delta' \subseteq \Delta''$ and $\sigma'_j = \sigma_j$.

We could use IFF for that purpose if systematically guessing the goals G_i and applying the proof procedure with initial hypotheses Δ afterwards. But this is computationally intractable, because there are infinitely many choices for G_i which fail after an expensive amount of computation. Following the same argumentation, we cannot simply reverse the derivation relation of IFF, since this requires to invent hypotheses Δ' and constraints in advance.

Forward-Chaining Deduction: DLS & DLSFN The stated functional complementarity actually needs a complementary definition of the inferences which traditionally, such as in the case of SLD, SLDNF, and IFF, elaborate programs in a *backward-chaining* manner (from a definition's head to its body; from goals to preconditions). If

maintaining a database, for example, many users or processes are able to pose queries not known in advance of which many subproblems are redundant. To optimise responses, *forward-chaining* procedures have been developed that compile the conclusions of a database with respect to a background program ahead of time.
This database-driven approach is a quite natural specification of the initial part of an agent's perception-action cycle: Agents cannot behave purely goal-driven, because they first need a prime means, their knowledge base process, for deriving these goals from the environmental stimuli. Rule-based production systems have been successfully applied in this context, such as in the ACT-R architecture [And93] and the MAGSY system [Fis93a].
Complementary to backward-chaining, forward-chaining starts with an empty database ($\top$). It subsequently fills the database with consequences by feeding its intermediate results into the bodies of clausal definitions. An example is the following DLS step which builds the complement to SLD (letting $Li : \mathit{Literal}$):

$$\frac{G_1 \wedge G_2 \quad \pi(P) \ni (\tilde{\forall} Li \subset \tilde{\exists} G) \quad G_1\sigma = G\sigma}{Li\sigma \wedge G_1 \wedge G_2} \text{ DLS}$$

Answers can be extracted out of each intermediate database $\top \vdash_P^t G$ by generating all substitutions σ of free variables. If fairness is established in the selection of clauses and goals, we derive a sound and complete procedure for generating all the conclusions of a definite program. At the first sight, DLS is not close to an implementation, because it does not prevent multiple derivations of the same goals. Moreover, the matching of conjuncted goals within multiple clauses is more complicated than unifying literals. To avoid redundancies, pattern-matching algorithms, such as RETE [For82], have been employed that only propagate the deltas of the the database.
Trying to find a forward-chaining basis for normal logic programs, i.e., for deriving negated literals, is not quite as straightforward. The point in complementing SLDNF is to build up negative consequences interleaved with the positive literals. The following FN step arguments over the completion of the program: A negated literal can be derived if, for all relevant clauses in the program, at least one literal in the body appears negated ($\overline{G}$ denotes the set of dual literals) in the predecessor database. Fair selection presumed, we thus gain a sound and complete DLSFN ::= DLS + FN procedure. DLSFN is closely related to the fix-point semantics of [Kun87].

$$\frac{G \quad \pi(P) = \bigwedge_i \tilde{\forall} Li_i \subset \tilde{\exists} G_i \wedge G_i' \quad Li\sigma \neq Li_i\sigma \text{ or } (\bigwedge_i G_i \wedge \overline{G'}_i)\sigma \subseteq G\sigma}{\neg Li\sigma \wedge G} \text{ FN}$$

Forward-Chaining Abduction: FFI In [BGLM92], a compositional fix-point semantics for definite programs, Open Logic Programming (OLP), is given which has some relations to Abductive Logic Programming. There have been efforts to connect forward-reasoning procedures with assumption-based truth maintenance systems (ATMS) [FDC87, OI92] from the background of default logics [Rei80]. We will now transfer the basic principle behind these approaches to equivalence programs with integrity constraints to obtain the forward-chaining variant of IFF, FFI: Instead of incrementally extending a database of pure literals, FFI maintains a set of range-restricted clauses $K ::= \bigwedge(G_1 \supset G_1 \wedge G_2)$ with $\mathcal{V}(G_2) \subseteq \mathcal{V}(G_1)$. Each range-restricted clause determines (positive and negative) hypotheses G_1 and their (positive and negative) consequences $G_1 \wedge G_2$ with respect to P. Range-restricted clauses

are generalisations of integrity constraints: By 'firing' their consequences into P, we eventually arrive at clauses whose conditions only refer to abducibles and equalities: $\Delta' \wedge \overline{\Delta''} \wedge X_j \doteq Te_j \wedge \neg X_i \doteq Te_i \supset G$. Given any σ that satisfies the equalities and inequalities and that leads to $\overline{\Delta''}\sigma \cap \Delta'\sigma \not\approx \varnothing$, we gain answers Δ', σ, G for which holds $\Delta'\sigma \models_P G\sigma$.

The main step of FFI is 'folding' (FLD1). Hereby, the conclusion of a range-restricted clause is partially matched (modulo abducible expressions Δ and $\overline{\Delta'}$) against a disjunct in the body of a program definition. This derives a new clause that combines the preconditions of its predecessor with the non-matched abducibles in the definition's body that collectively imply the defined literal, the abducibles, and the former consequences. Similarly, we can fold negated literals (FLD2) by finding in each disjunct of a definition at least one literal that appears dual in the range-restricted clause (or that can be abduced as invalid). For reasons of simplicity, we have omitted to make (dis-)unification explicit in FFI.

$$\frac{(G_1 \supset G_2) \wedge K \quad \pi(P) \ni (\tilde{\forall} Li \equiv \tilde{\exists} G_3 \vee G_4) \quad G_3\sigma \subseteq (G_2 \wedge \Delta \wedge \overline{\Delta'})\sigma}{(G_1 \wedge \Delta \wedge \overline{\Delta'} \supset G_2 \wedge \Delta \wedge \overline{\Delta'} \wedge Li)\sigma \wedge (G_1 \supset G_2) \wedge K} \text{ FLD1}$$

$$\frac{(G_1 \supset G_2) \wedge K \quad \begin{array}{c}\pi(P) \ni \\ (\tilde{\forall} Li \equiv \bigvee_i \tilde{\exists} G_i \wedge G_i')\end{array} \quad \bigwedge_i (G_i \wedge \overline{G_i'})\sigma \subseteq (G_2 \wedge \Delta \wedge \overline{\Delta'})\sigma}{(G_1 \wedge \Delta \wedge \overline{\Delta'} \supset G_2 \wedge \Delta \wedge \overline{\Delta'} \wedge \neg Li)\sigma \wedge (G_1 \supset G_2) \wedge K} \text{ FLD2}$$

By requiring FLD1 and FLD2 to match maximal subsets of conditions, FFI builds up minimal hypotheses and recognises dead ends during a proof due to redundant clauses (the following PMS simplification). Redundant clauses contain inconsistent conclusions due to dual literals or failed dis-unification.

$$\frac{(G_1 \supset (Li \wedge \neg Li \wedge G_2)) \wedge K}{K} \text{ PMS1} \qquad \frac{(G_1 \supset (\neg Te \doteq Te \wedge G_2)) \wedge K}{K} \text{ PMS2}$$

Integrity constraints are a means to restrict the 'hypothetical models' generated by FFI. This is realised by the following integrity analysis step (ITA). Since a complete set of consequences is built within the conclusions of the range-restricted clauses, ITA frequently tries to match constraint conditions there. ITA then introduces successor clauses with possibly extended conditions and consequences that reinstall integrity. ITA furthermore contains a version of case analysis by which the integrity of a constraint is also preserved, if the relevant parts of the clause can be dis-unified.

$$\frac{(G_1 \supset G_2) \wedge K \quad \pi(IC) \ni (\tilde{\forall} G_3 \supset \bigvee_i \tilde{\exists} G_i) \quad G_3\sigma \subseteq G_2\sigma \quad \bigvee_j \neg X_j \doteq Te_j = \neg\sigma}{\bigwedge_j((G_1 \wedge \neg X_j \doteq Te_j) \supset (G_2 \wedge \neg X_j \doteq Te_j)) \bigwedge_i((G_1 \wedge G_i) \supset (G_2 \wedge G_i))\sigma \wedge K} \text{ ITA}$$

The requirements triggered by ITA on the left-hand side of range-restricted clauses do not only cover (dis-)equalities and abducibles, but defined literals as well. Up to now, these will not be resolved. In order to re-establish the canonical form of clauses, we introduce unfolding steps (FNU1 and FNU2) which are quite similar to unfolding of constraints in IFF, but also operate on negated literals. Unfolding enables FFI to be incremental with respect to additional goals G'.

$$\frac{(Li_1 \wedge G_1 \supset G_2) \wedge G_3 \quad \pi(P) \ni (\tilde{\forall} Li_2 \equiv \bigvee_i \tilde{\exists} G_i) \quad Li_1\sigma = Li_2\sigma}{\bigwedge_i (G_i \wedge G_1 \supset G_i \wedge G_2)\sigma \wedge G_3} \text{ FNU1}$$

$$\frac{(Li_1 \wedge G_1 \supset G_2) \wedge G_3 \quad \pi(P) \ni (\tilde{\forall}\overline{Li_2} \equiv \bigvee_i \tilde{\exists} G_i) \quad Li_1\sigma = Li_2\sigma}{(\bigwedge_i \neg G_i \wedge G_1 \supset \bigwedge_i \neg G_i \wedge G_2)\sigma \wedge G_3} \text{ FNU2}$$

FFI starts with the initial clause $K ::= \tilde{\forall}\Delta \supset \Delta$ and by verifying that it performs equivalence transformations under P, we gain soundness and completeness for deriving answers Δ'; σ; G. This means that even infinite sets of consequences G are appropriately approximated in the fix-point. We will not give a formal proof of this result, here. Rather, we immediately turn to *PrHEC* where we postulate FFI to install a sound and complete algorithm for solving the APrP.

Theorem 12 (Sound And Complete APrA) *$I \wedge DOM \vdash^{\sigma,G}_{PrHEC} \Delta$ is a sound and complete Abstraction Prediction Algorithm (APrA) for solving an APrP$(I, DOM, \Delta, \mathcal{L})$.*

Proof. Similarly to planning with IFF in Theorem 9, soundness and completeness of FFI carry over to the prediction task. In contrast to Theorem 9, this result does not rely on any solution properties, because of the definition of the prediction task. □

In the following, we will look at operational issues which turn FFI & *PrHEC* into an interactive prediction framework.

Making Persistent and Past-Oriented Predictions On-line FFI and *PrHEC* comprise InteRRaP-R's mental model process, $\wr\!\!-(mental\_model) ::= \vdash_{PrHEC}$, in which each *Option* corresponds to a range-restricted clause, its *Program* containing the left-hand side of the clause and its *Goal* containing the right-hand side. Moreover, FFI and *PrHEC* provide a suitable on-line interface: Intermediate solutions can be extracted which fully predict particular levels of abstraction and can be committed by sharing and signalling its parts, e.g., to the local planning process, to plan execution processes, and to other knowledge base processes (see Subsections 5.3.3, 5.4.2, and 5.5). In addition, FFI upholds the same degree of incrementality as IFF with respect to hypotheses Δ', goals G', and constraints IC' that are reported to the mental model process, i.e., we can extend any intermediate clause $\tilde{\forall}\Delta \wedge G_1 \supset G_2$ to $\tilde{\forall}\Delta \wedge \Delta' \wedge G_1 \wedge G' \supset \Delta' \wedge G_2 \wedge G'$ and constraints IC to $IC \wedge IC'$ in order to redirect the inference.
As for local planning, operational design must be added to make the abductive prediction framework running in practice. The FFI selection strategy solidifies in the selection of range-restricted clauses (search rule) and definition matches (computation rule). Least-commitment in FFI means to always choose new and maximal matches (keep Δ and $\overline{\Delta'}$ in FLD as minimal as possible), to perform ITA, PMS, and FNU before any FLD step, and to only *member* literals whose lists exclusively refer to events from the residue and have at most one duplicate entry. The well-definedness of *HEC* gives also in this forward-chaining case a terminating procedure for generating the finitely many effects of a bounded residue, hence a complete prediction procedure by iterative-deepening. The cost measure $\mathcal{C}$ is chosen to first recognise detailed occurrences and afterwards turn to the more abstract ones, i.e., macros are more expensive than their decomposition.
As a by-product of prediction, the mental model determines future goals G of the agent from to the given environmental stimuli. We realise such intrinsic 'desires' as the mandatory integrity of the mental model, thus as constraints and definitions of the following form:

$$
\begin{aligned}
&\tilde{\forall} unloadDesire(Box, Shelf, T_1, T_2, E) \equiv \\
(DES1) \quad &\tilde{\exists} T_3 \dot{\leq} T_2 \wedge holds(unloadCommand(Box), T_1, T_3, nil, \dot{0}, E, l_2) \\
&\quad \wedge holds(category(Box, Cat), T_1, T_3, nil, \dot{0}, E, l_2) \\
&\quad \wedge holds(category(Shelf, Cat), T_1, T_3, nil, \dot{0}, E, l_2)
\end{aligned}
$$

$$
\begin{aligned}
&\tilde{\forall} unloadDesire(Box, Shelf, T_1, T_2, E) \supset \\
(DES2) \quad &\tilde{\exists} holds(at(Box, Shelf), T_3, T_2, nil, \dot{0}, E, l_2) \\
&\quad \wedge holds(at(Box, Shelf), T_3, T_2, nil, \dot{0}, E, l_3) \\
&\vee \tilde{\exists} holds(at(Box, Shelf), T_3, T_2, nil, \dot{0}, E, l_3)
\end{aligned}
$$

Whenever FFI predicts the situation defined in *DES*1, the constraint *DES*2 is fired by ITA and adds its conclusions into the left-hand side of the selected range-restricted clause. These are abstraction planning goals according to Theorem 10 and will be signalled to (or better, requested from) the planner. In doing so, a particularly persisting situation is not allowed to spawn arbitrarily many goals:

$$
\begin{aligned}
&\tilde{\forall} \neg disjoint(T_1, T_2, T_3, T_4) \\
(DES3) \quad &\wedge unloadDesire(Box, Shelf, T_1, T_2, E_1) \\
&\wedge unloadDesire(Box, Shelf, T_3, T_4, E_2) \supset \tilde{\exists} E_1 \doteq E_2 \wedge T_1 \doteq T_2 \wedge T_3 \doteq T_4
\end{aligned}
$$

FFI has in principle the possibility to elaborate such goals by FNU, i.e., to perform planning. Complementary to the partially suppressed explanation facilities of the local planning process, the mental model just speculates about effects of, e.g., observed intentions, but does not issue decisions by itself. Therefore, FNU is restricted not to elaborate goals whose interval begins after the current time point t_i. Moreover, it is enforced to unify abduced actions of its proper agent ($pickup(rob_1, box_1)$) with commitments that have been reported by planner and executer.

Miracles are useful to provoke and explain ad-hoc updates in the knowledge base, such as partial observations of object movements and unexpected robot displacements, which have none or only expensive diagnoses, otherwise. Miracles are not used in the regular course of prediction; they are associated a quite high cost C that is traded off against the expense of inferences that are needed to explain the observation regularly. When applying miracles to explain some low-level of abstraction, FFI and *PrHEC* nevertheless recognise regular higher-level narratives, e.g., intentions of other agents which cannot be immediately perceived, but could be communicated, from their abstract effects.

When minimising persistence perturbations in the past, the mental model generates partially-instantiated $\neg happens$ abducibles. This is reasonable, since new evidence always leads to back-dating explanations and requires to invalidate some of these hypothesised persistencies. On the other hand, it is not plausible to let these activities range arbitrarily far into the past. We thus strengthen the inferential power of the knowledge base by stating that all furtherly abduced events must not have happened after $t_{i-\epsilon}$

$$(IC') ::= \tilde{\forall} T_1 \dot{\leq} t_{i-\epsilon} \supset \neg \tilde{\exists} happens(E, A, T_1, T_2, L)$$

where ϵ is the layer- and domain-dependent roll-up interval (Subsection 5.5).

Exception handling and roll-ups in the mental model operate on the level of range-restricted clauses to keep a compact and tractable representation and to recover from

incorporating inconsistent information, e.g., wrong hypotheses and unsatisfiable requests. A roll-up at time t_i compiles the predicted effects of time $t_{i-\epsilon}$ into a new initial situation I, forgets hypotheses and goals before that time, and removes several of the upcoming goals and observations. Usually, the incomplete I must be a subject to (reasonably domain-restricted) abduction, too, which is straightforwardly expressible in FFI and *PrHEC*.

5.4.2 Plan Execution: On-line Decomposition by ALP & *HEC*

Plan execution processes complete the LPL's perception-action cycle by mediating between the complementary functions of mental model and local planning. Their input consists of abstract intentions Δ' decided by the planner and a temporal trace of believed situations G predicted by the mental model, e.g.,

$$\Delta' ::= happens(e_1, transport(rob_1, box_1, parking, truck, shelf_3), t_1, t_2, l_2)$$
$$G ::= holds(atPos(rob_1, \dot{6}, \dot{2}), t_1, t_4, nil, \dot{0}, E, l_4) \wedge \ldots$$

Their output consists of a decomposition Δ of the given intentions, for example $\Delta ::= happens(e_2, moveArea(rob_1, truck), t_1, t_3, l_4) \wedge \ldots$, which is in concordance with the given situation trace. Since conflicts between high-level intentions should have been resolved or will be resolved concurrently by the local planning process, their separation into separate and newly initialised execution processes avoids the combinatorics inherent to optional decompositions and allows for a quick and interactive inference.
In Section 4, we have demonstrated that *HEC* contains a sub-theory that comprises an 'interpreter' for a macro language that includes primitive statements, test actions, sequential and interleaved compositions, disjunctive choices, and finally procedural abstraction by the instrument of *DOMAAB*. The respective sub-theory, $ExHEC ::= HEC6 \wedge HEC10 \wedge HEC11 \wedge ECK3 \wedge ECK4$, purely relies on constraints maintaining the decomposition of macros, the uniqueness and duration of events, and their partial temporal ordering. The Abstraction Execution Problem (AExP) formalised in Definition 14 is thus solved soundly and completely by using IFF inferences $\vdash(plan\_execution) ::= \vdash_{ExHEC}$ (Theorem 13).

Definition 14 (AExP) *Let $\mathcal{L} ::= \{l_0, l_1, \ldots, l_n\}$ be a set of incremental levels of abstraction. Let DOM, Δ', G be a domain axiomatisation, an abstract commitment, and a situation trace.*
The Abstraction Execution Problem $AExP(DOM, \Delta', G, \mathcal{L})$ is the problem of finding a series of incrementally refined plans and substitutions $\{(\Delta_j, \sigma_j) \mid 0 \leq j \leq n\}$ such that $\Delta_j(\{l_0, \ldots, l_j\}) = \Delta_{j+1}(\{l_0, \ldots, l_j\})$, $\Delta_j \supseteq \Delta'(\{l_0, \ldots, l_j)$, and σ_{j+1} is identical to σ_j restricted to variables in $G(\{l_1, \ldots, l_n\})$. It must hold

$$G \wedge \Delta_j \wedge \Phi(\{l_j\}) \models_{ExHEC} \sigma_j$$

where Φ be the set of ground decomposeMacro literals such that $DOMAAB \wedge \Delta_n \models \Phi$. The AExP is said to be solved completely if for all $\{(\Delta'_j, \sigma'_j) \mid 0 \leq j \leq n\}$ which solve the AExP, we can find a solution $\{(\Delta_j, \sigma_j)\}$ such that $\Delta_j(\{l_j\}) \subseteq \Delta'_j(\{l_j\})$ and $\sigma_j = \sigma'_j$.

Theorem 13 (Sound and Complete AExA) $\Delta' \wedge DOMAAB \wedge G \vdash_{ExHEC}^{\Delta,\sigma}$ *is a sound and complete Abstraction Execution Algorithm (AExA) to solve an* $AExP(DOM, \Delta', G, \mathcal{L})$

.

Making Persistent and Present-Oriented Execution On-Line Incremental IFF & *ExHEC* even provide on-line execution processes, i.e., besides early outputting portions of the decomposed plan Δ (ranging through all necessary levels of abstraction), they are able to integrate new situation information G' as well as concurrently refined planner commitments Δ' both transmitted through the mental model on the fly.
To decompose a reasonable initial portion of the plan, a similar selection to the one of Subsection 5.3.2, but with higher preference to expressions with a due time close to the current clock. This can be reflected by looking at the $\dot{<}$ abducibles (and the more practical temporal representation discussed in the next section). For example, it is not necessary to get muddled by doing things too early, hence restricting our further possibilities for pursuing other goals and for refining the already made commitments. On the other hand, we should not wait too long, since any perturbation could render the execution impossible. The strategy therefore depends on the likelihood of interruptions and the usefulness of idle time in the domain.
LPL plan execution processes have to decompose particular multi-agent plans, such as two forklift's mutual exchange of a box $happens(e_1, exchange(rob_1, rob_2, box_1, shelf_3), t_1, t_2, l_2)$. Hereby, those those parts of the multi-agent macro must be decomposed in which the agent has an active role, while passively waiting for contributions of other agents being reported as new evidences Δ' by the mental model. Hence, *HEC*6 is operationally restricted to macros with arguments referring to, e.g., rob_1.
On the most primitive level of abstraction, selected actions will be 'time-stamped' by extending the current substitution with $T_1 \doteq t_i$. Usually, such primitive actions are signalled to an external environment (the agent's body) which reports, e.g., end times and status information back. Since LPL actions represent parameterisations of BBL computation, they rather invoke updates Δ', G' and feedback within the world model by way of the mental model (see the following subsection). From the logic perspective, this does however not make any difference.
Test actions which have been decomposed into *holds* statements are reported to the mental model as requests. Looking up particular situation features on-demand saves a lot of bandwidth in contrast to letting the mental model advertise its complete content to all execution processes. Hence, the executer temporally assumes particular unknown occurrences while proceeding with its decomposition. If such assumptions, similar to awaited cooperative actions of other agents, turn out to be inconsistent, the selected execution option fails, invokes exception handling and roll-up, i.e., a clean-up of the situation trace and the scheduled events. This treatment will play an important role in the following subsection when realising negotiations between multiple agents at their SPL.

5.5 InteRRaP-R: Horizontally Interacting Layers

The preceding subsection demonstrated the vertical modularisation of a logic-based agent (or LPL) allowing for a simultaneous solution to prediction, planning, and exe-

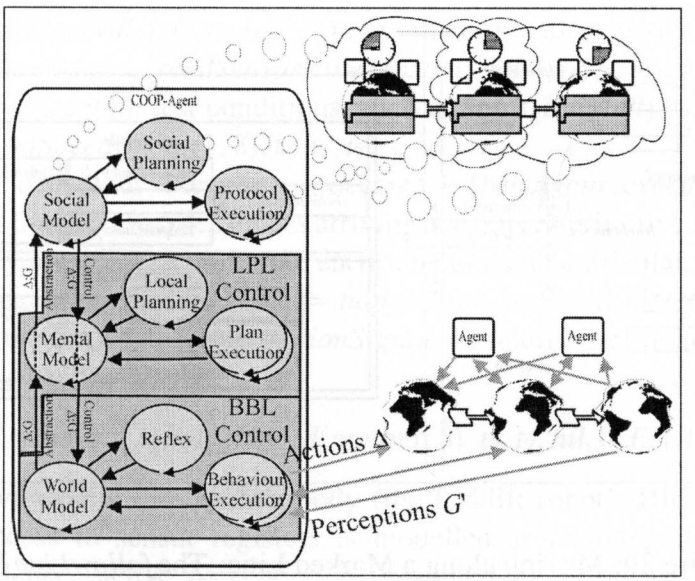

Figure 28: Inference Processes in the Horizontally Modularised InteRRaP-R

cution problems. Such deliberative capabilities are, but, just one aspect of broad agents which also have to solve the reactive and social instances of these problems. To this end, the InteRRaP-R design additionally envisages a horizontal modularisation into a hierarchy of layers, the LPL being the intermediate one (see Figure 28).
In the following, we will express reactive capabilities in the BBL (Subsection 5.5.1) and social capabilities in the SPL (Subsection 5.5.3) by the same logic instruments used to obtained the LPL. Since uniformly representing complete perception-action mappings, these layers are no longer distinguishable by functional criteria. Instead, their processes are separated by temporal and representational differences which do not differ in their theoretical and inferential background:

$$\begin{gathered}\wr\!\!-(\mathit{world\_model}) = \wr\!\!-(\mathit{mental\_model}) = \wr\!\!-(\mathit{social\_model}) \\ \wr\!\!-(\mathit{reflex}) = \wr\!\!-(\mathit{local\_planning}) = \wr\!\!-(\mathit{social\_planning}) \\ \wr\!\!-(\mathit{behaviour\_execution}) = \wr\!\!-(\mathit{plan\_execution}) = \wr\!\!-(\mathit{protocol\_execution})\end{gathered}$$

Instead, layer characteristics are determined by choosing the domain representation *DOM* accordingly. Figure 28 illustrates the meta-control principle of InteRRaP-R in which the representations of an upper layer build on abstractions of the representations of the lower layer. This is quite according to the way that abstraction hierarchies are formalised in *HEC*. In addition, control representations become accessible — abstract resources and their parameterisation — that influence the computation of the underlying layer by means of the computational model, here: dedicated control processes. Subsection 5.5.2 discusses this at hand of the interaction between LPL and BBL.
Meta-control is not to confuse with full-fledged meta-reasoning in which the meta-level possesses the actual computational model of the object-level as its representational basis. Even in the SPL, where some higher-level reflection about the mental model of various agents has to be derived, we do not allow for complete computational introspection because of tractability reasons.

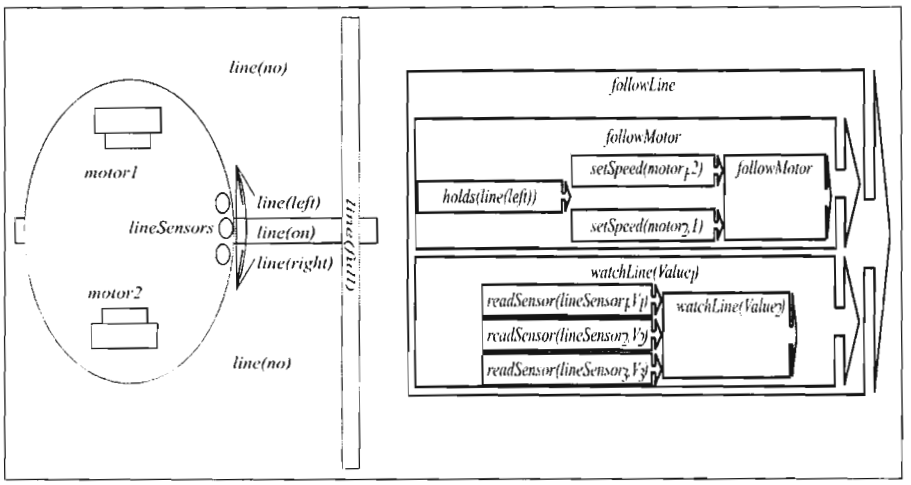

Figure 29: Moving along a Marked Line: The *followLine* Macro

5.5.1 BBL: Behaviour Execution, World Model, and Reflex

A typical reactive control problem of the forklift robots in the Automated Loading Dock is the movement along a marked line by mapping concrete perceptions, e.g., readings from infrared sensors, onto concrete actions, e.g., wheel motor commands (Figure 29). For this purpose, the processes inside their Behaviour-Based Layer must be able to perform computations very quickly within the fraction of a second not to loose sight of the line and not to bump into an obstacle at the same time. Inferences are usually regarded as far too expensive for this task because of tending to elaborate exponentially many options within the problem specification.
Having a closer look at the required reactive fluents (sensor readings, motor status) and actions (sensing and commanding actions), a causal axiomatisation can be quite simple such that *HEC* and ALP mainly rely on situation abstraction in *DOMSAB* and action decomposition in *DOMAAB*. Since these definitions usually introduce just a bounded number of mostly deterministically resolvable options, the above specified prediction, planning, and execution procedures can reasonably operate as a quick, though logically specified BBL. This BBL is very close to the agents of [KS96b, Dav96] and to PRS of [GL87]. Its design is able to master settings, such as the Automated Loading Dock and the RoboCup simulation, which are, say, modestly reactive with respect to timing (close-to-real-time) and physics (small number of reliable sensors and effectors).
HEC and ALP surely do not ultimately address issues such as real-time behaviour [AZ87], continuous control [Kha83], uncertainty [DvLV96], sensor fusion [DW88], vision [Mar82], and object recognition [Hum96] to which dedicated research areas in AI exist and for which a final logic treatment, if desirable, is not to be found. However, what we outline in the following is that the declarative boundary can be pushed quite far such that well-defined interfaces to these efforts become apparent.

Behaviour Execution: Primitive Actions and Active Perception One focus of the BBL is the quick execution of behavioural 'routines' such as the line-following macro $happens(e_1, followLine, t_1, t_2, l_5)$ that invokes concurrent 'sub-routines' $happens(e_3, followMotor, t_1, t_2, l_6)$ and

$happens(e_2, watchLine(Val_1), t_1, t_2, l_6)$. These, in turn, tail-recursively call sensing actions, such as $happens(e_4, readSensor(lineSensor1, Val_2), t_1, t_3, l_7)$, and motor commands in concordance with test conditions, such as $holds(line(left), t_1, t_5, nil, \dot{0}, E, l_6)$ and $happens(e_6, setSpeed(motor1, \dot{3}), t_6, t_7, l_7)$.
The responsible behaviour execution processes $\vdash\!(behaviour\_execution)$ go quite along the lines of Subsection 5.4.2 unless arriving at the most primitive level of abstraction in which they must interact with the environment using particular types of COOP signals. Primitive actions, such as $\Delta ::= happens(e_6, setSpeed(motor1, \dot{3}), t_6, t_2, l_7)$, are published to the environment via *ActionSignal*. In return, the environment reports its status via *PerceptionSignal*, e.g.,

$$G' ::= t_4 \doteq t_i \wedge holds(speed(motor1, \dot{3}), t_2, t_2, nil, \dot{0}, E, l_7)$$

From the logic viewpoint, the agent's body (the forklift robot's BIOS, in this case) including the access to sensor registers is modelled as an integrative part of the environment. Hence, active perception is modelled by primitive actions $\Delta ::= happens(e_4, readSensor(lineSensor1, Val_2), t_1, t_3, l_7)$ that will be returned a reading $G' ::= t_3 \doteq t_j \wedge holds(reading(lineSensor1, 1\dot{2}3), t_3, t_3, nil, \dot{0}, E, l_7)$. The perceived information are signalled to the world model whose predictions in turn steer the decomposition of active behavioural macros.

World Model: Sensor Fusion and Object Recognition The second focus of the BBL is the world model that subjects incoming observations (executed actions Δ', status reports and sensor feedback G') to a prediction process quite along the lines of Subsection 5.4.1. In fact, G' is not immediately telling the agent what the world looks like, rather it is a sensor-specific symptom caused by sensing actions in Δ' under particular environmental conditions. In fact, these environmental conditions stem from the having issued motor commands in Δ' which do not immediately change the state of the world, but rather perturb a continuous trajectory of movement in interaction with the trajectories of other agents and objects.
An approach for reasoning about continuous activities [Sha96, San97] could be incorporated into *HEC* in order to connect Δ' and G' by the prediction of the world model. To keep this enterprise simple and effective, however, we can also shift to a discrete level of representation by interchanging the causal role of sensing and acting:

$$\begin{aligned}(DOMCAU1)\quad & \tilde{\forall} causes(readSensor(Sensor, Val_1), F, T_1, T_2, C, B, E, l_7) \equiv \\ & \quad \tilde{\exists} F \doteq reading(Sensor, Val_1) \\ & \quad \vee \tilde{\exists} \neg Val_1 \doteq Val_2 \wedge F \doteq not(reading(Sensor, Val_2)) \\ & \qquad \wedge holds(reading(Sensor, Val_2), T_1, T_2, C, B, E, l_7)\end{aligned}$$

Using *DOMCAU*1, the world model now deterministically builds up a history of sensing samples to which situation abstraction can be applied: The stimuli from several line sensors are fused into an estimation of the robot's position with respect to the observed line at level l_6:

$$\begin{aligned}(DOMSAB1)\quad & \tilde{\forall} decomposeHolds(line(left), T_1, T_2, C, B, E, l_6) \equiv \tilde{\exists} Val_1 \dot{\geq} Val_2 \dot{\geq} Val_3 \\ & \quad \wedge holds(reading(lineSensor1, Val_1), T_1, T_2, C, B, E, l_7) \\ & \quad \wedge holds(reading(lineSensor2, Val_2), T_1, T_2, C, B, E, l_7) \\ & \quad \wedge holds(reading(lineSensor3, Val_3), T_1, T_2, C, B, E, l_7)\end{aligned}$$

$$
\begin{aligned}
&\tilde{\forall} causes(watchLine(Val_1), F, T_1, T_2, C, B, E, l_6) \equiv \\
&\quad \tilde{\exists} F \doteq line(Val_1) \\
&\quad \vee \tilde{\exists} \neg Val_1 \doteq Val_2 \wedge F \doteq not(line(Val_2)) \\
&\qquad \wedge holds(line(Val_2), T_1, T_2, C, B, E, l_6)
\end{aligned}
\qquad (DOMCAU2)
$$

Already at this level, a feedback loop operating within the fraction of a second is closed, because the world model's prediction of the line status is sent to steer the conditional decomposition of *followMotor* in a behaviour execution process. Similarly, loosing sight of the line immediately triggers a desire of the form *DES*1 that is signalled to the reflex process to find some appropriate counter measure, such as the activation of the *searchLine* behaviour:

$$
(DES1) \quad \tilde{\forall} holds(line(no), T_1, T_2, nil, \dot{0}, E_1, l_6) \supset holds(safePos, T_3, T_4, nil, \dot{0}, E_2, l_5)
$$

This rapid inference avoids to pursue combinatorially many options and becomes possible by the modelling of independent and reliable sensors. For noisy settings in which various perceptions from different sensors can be caused by exactly the same stimulus, this is not appropriate. The required fusion of sensors is usually performed by Bayesian methods such as they have been introduced into abductive logic by [PK94] and the deductive Situation Calculus by [BHL95]. Although Bayes allows to revert causality, such as we have proposed above, the update of probability distributions within such logics is still rather expensive.
[Sha97b] has proposed to express uncertainty as bounded and equally distributed nondeterminism, i.e., the 'hypothetical' stimuli are supposed to lie within a specified range δ from the observed ones. By using compact numerical representations and corresponding background theories (see, e.g., Section 6), the enumeration of inherent options is avoided, nevertheless plausible abstract information can be derived:

$$
\begin{aligned}
&\tilde{\forall} decomposeHolds(line(left), T_1, T_2, C, B, E, l_6) \equiv \tilde{\exists} Val'_1 \dot{\geq} Val'_2 \dot{\geq} Val'_3 \\
&\quad \wedge Val_1 \dot{-} \delta \dot{\leq} Val'_1 \dot{\leq} Val_1 \dot{+} \delta \\
&\quad \wedge holds(reading(lineSensor1, Val_1), T_1, T_2, C, B, E, l_7) \wedge \ldots
\end{aligned}
\qquad (DOMSAB1')
$$

For both line-based navigation in the loading dock and landmark-based navigation in the RoboCup, this approach installs a robust low-level of control. Hence, we simply assume failure-safe moving and turning on the next level of abstraction in order to trace the positioning, the orientation, and the surrounding of the robot:

$$
\begin{aligned}
&\tilde{\forall} causes(atPos(X_1, Y_1), followLine, T_1, T_2, C, B, E, l_5) \equiv \tilde{\exists} D \doteq west \\
&\quad \wedge Y_1 \doteq Y_2 \wedge X_1 \doteq X_2 \dot{-} \dot{1} \wedge holds(atPos(X_2, Y_2), T_1, T_2, C, B, E, l_5) \\
&\quad \wedge holds(orient(D), T_1, T_2, C, B. E, l_5) \\
&\quad \wedge holds(freeAhead, T_1, T_2, C, B, E, l_5)
\end{aligned}
\qquad (DOMCAU3)
$$

The *atPos* fluent is an example of a 'phenomenal' fluent which is not immediately grounded in observable primitives, thus without an identifiable 'physical' substrate. Such representations play a major role in philosophical debates about qualia and intentionality. They are technically relevant to connect sub-symbolic values with deliberative representations, e.g., for building coordinate systems, identifying objects, and maintaining and communicating cognitive maps.

This way, but, the world model cannot detect that the robot has lost the line and has recovered at a wrong position. For that purpose, we introduce miracle events which explain the sudden displacements:

$$(DOMCAU4)\quad \begin{aligned}&\tilde{\forall} causes(atPos(X_1, Y_1), displacePos(X_1, Y_1), T_1, T_2, C, B, E, l_5) \equiv \\ &\quad \tilde{\exists} \neg X_1 \doteq X_2 \wedge holds(atPos(X_2, Y_2), T_1, T_2, C, B, E, l_5) \vee \ldots\end{aligned}$$

$$(DOMCAU5)\quad \tilde{\forall} causes(orient(D), displaceOrient(D), T_1, T_2, C, B, E, l_5) \equiv \ldots$$

Miracles are annotated with extensive costs. The world model does not use these without getting strong evidences that positioning has gone mad, for example, by unexpectedly arriving at a unique landmark. Because miracles do hardly interact with other actions, the world model simply seeks the most plausible place in the collected history where the displacement could have occurred and re-interprets its prediction.
For practicability purposes, the world model renounces to represent other objects and agents on the same low-level of state and activity. Given the usual small-bandwidth sensors, it is a hard problem to recognise objects and their actions. Hence, we deal with the representation of multiple agents mainly on the basis of communication facilities, e.g., basic behaviours $say(Ag, W)$ and $hear(Ag)$ and basic fluents $heard(Ag, W)$ and $said(Ag, W)$ by which agents transmit who and where they are and what they are doing (see Subsection 5.5.3). To enable box recognition in the loading dock, we have added an extra 'virtual' sensor that is communicated the identifications.
Nevertheless, some of these issues can be addressed by the instruments of background knowledge, hypothetical reasoning, and plausibility within our abductive logic. For example, the existence of an object is inferable from scanning proximity sensors (the *watchAhead* behaviour). If an object is not exactly identifiable by a stimulus, the world model introduces an existentially quantified variable for its identity:

$$(DOMSAB2)\quad \begin{aligned}&\tilde{\forall} decomposeHolds(ahead(O), T_1, T_2, C, B, E, l_6) \equiv \\ &\quad \tilde{\exists} holds(reading(proxSensor1, Val_1), T_1, T_2, C, B, E, l_7) \wedge \ldots\end{aligned}$$

Triggered by this information, a collision-avoidance desire of the form

$$(DES2)\quad \begin{aligned}&\tilde{\forall} holds(ahead(X), T_1, T_2, nil, \dot{0}, E, l_6) \supset \\ &\quad holds(freeAhead, T_3, T_4, nil, \dot{0}, E_2, l_5)\end{aligned}$$

can invoke some *dodge* behaviour by way of the reflex process. Object positions and movements are even inferable from the agent's own position and status:

$$(DOMSAB3)\quad \begin{aligned}&\tilde{\forall} decomposeHolds(atPos(O, X_1, Y_1), T_1, T_2, C, B, E, l_5) \equiv \\ &\quad \tilde{\exists} holds(ahead(O), T_1, T_2, C, B, E, l_6) \\ &\qquad \wedge holds(atPos(X_1, Y_1, T_1, T_2, C, B, E, l_5) \\ &\qquad \wedge holds(orient(D), T_1, T_2, C, B, E, l_5) \\ &\qquad \wedge D \doteq west \wedge Y_1 \doteq Y_2 \wedge X_1 \doteq X_2 \dot{-} \dot{1}\end{aligned}$$

$$(DOMCAU6)\quad \begin{aligned}&\tilde{\forall} causes(atPos(O, X_1, Y_1), displacePos(O, X_1, Y_1), T_1, T_2, C, B, E, l_5) \equiv \\ &\quad \tilde{\exists} \neg X_1 \doteq X_2 \wedge holds(atPos(Box, X_2, Y_2), T_1, T_2, C, B, E, l_5) \vee \ldots\end{aligned}$$

By minimising its predictions, the world model then assumes the most plausible environmental actions that could have occurred in order to consistently explain the current situation in relation to the agent's history. Using background knowledge, such as about sorts of objects and occupancy restrictions, this is massively simplified. Still, it turns out a demanding enterprise in the case of displacement (loading dock) or many unidentifiable, moving objects (RoboCup). This is why we restrict the world model to a reasonable temporal horizon ϵ in the range of a few seconds whose imposed roll-ups frequently force to forget about unidentified objects and actions.

Reflex Since the reactive control of the BBL is mainly seated within prediction and execution processes, the decision-making reflex process experiences the representationally most restrictive and computationally most focusing variation of functionality by *DOM*:

$$
\begin{aligned}
&\tilde{\forall} causes(F, A, T_1, T_2, C, B, E, L) \equiv \\
&\tilde{\exists} L \doteq l_5 \wedge F \doteq safePos \wedge A \doteq searchLine \\
&\vee \tilde{\exists} L \doteq l_5 \wedge F \doteq freeAhead \wedge A \doteq dodge
\end{aligned}
\qquad (DOMCAU7)
$$

DOMCAU7 models very restricted consequences and preconditions of just a few behaviours, such as *searchLine* to recover a lost marking line and *dodge* to avoid collisions with facing objects. This way, the reflex process does not actually plan action sequences, but quickly activates macros that are deterministically coupled to the low-level goals transmitted from the world model. While the world model traces their execution over sensory sub-routines shared with other behaviours, such as *watchLine* shared by *searchLine* and *followLine*, the reflex process cannot causally analyse the necessity of its decisions: It cannot issue, e.g., *turnLeft* and *followLine* actions by itself. It cannot decide to issue *downArm*, *openCloseGripper*, and *upDownArm* instead of *dodge* in order to pickup an obstacle *Box*. And finally, it cannot resolve conflicts between, e.g., an active *followLine* behaviour which tries to approach an object for picking it up and a *dodge* decision.
Indeed, these tasks are delegated to the deliberative control of InteRRaP-R's LPL that configures the state and the computation of the presented BBL processes.

5.5.2 LPL: Abstraction and Abstract Resources

Roughly, the deliberative problem of forklift robots in the Automated Loading Dock consists of performing (joint) delivery tasks. The representations of the inference-based LPL that solves the corresponding (multi-agent) prediction, decision-making, and execution problems have already been extensively discussed in Subsection 5.4 apart from their connection with the just outlined BBL by the coupling of mental model and world model processes.
In the first place, this connection is established by stapling further deliberative levels of representation ($l_2, \ldots, l_4$, roll-up interval $\sim$ 30 seconds) on top of the just outlined BBL levels $l_5, \ldots, l_7$ by the instrument of *HEC* abstraction hierarchies. For example, the distinction of ego-centred and environmental representations, e.g., of $atPos(\dot{6}, \dot{2})$ and $atPos(box_1, \dot{1}, \dot{1})$ and of *upArm* and $upArm(rob_1)$, is released for the sake of an objective multi-agent perspective:

$$\tilde{\forall} decomposeHolds(atPos(O, X_1, Y_1), T_1, T_2, C, B, E, l_4) \equiv$$
$$(DOMSAB4) \quad \tilde{\exists} holds(atPos(O, X_1, Y_1), T_1, T_2, C, B, E, l_5)$$
$$\vee \tilde{\exists} O \doteq rob_1 \wedge holds(atPos(X_1, Y_1), T_1, T_2, C, B, E, l_5)$$

$$\tilde{\forall} decomposeMacro(pickup(O, Box), T_1, T_2, l_4) \equiv$$
$$(DOMAAB1) \quad \tilde{\exists} O \doteq rob_1 \wedge happens(E_1, downArm, T_1, T_3, l_5)$$
$$\wedge happens(E_2, closeGripper(Box), T_4, T_5, l_5)$$
$$\wedge happens(E_3, upArm, T_4, T_5, l_5)$$

As aforementioned, the lifted information from the world model serves to install the agent's positioning, orientation, and surrounding rather than to recognise hardly perceivable multi-agent interactions. Nevertheless, the mental model is able to resolve displacements and object movements by relying on additional background knowledge in the form of cognitive maps and common-sense:

$$(I) \quad \tilde{\forall} initially(F, L) \equiv$$
$$\tilde{\exists} F \doteq category(box_1, toys) \wedge L \doteq l_4 \vee \tilde{\exists} F \doteq at(box_1, truck) \wedge L \doteq l_4 \vee \ldots$$

$$\tilde{\forall} decomposeHolds(at(O, truck), T_1, T_2, C, B, E, l_4) \equiv$$
$$(DOMSAB5) \quad \tilde{\exists} holds(atPos(O, \dot{1}, \dot{1}), T_1, T_2, C, B, E, l_4)$$

Particular LPL primitives, such as $pickup(rob_1, box_1)$ at l_4, are executed in the plan execution processes by procedurally decomposing them into high-level behaviours at l_5, such as the sequence *downArm*, $closeGripper(box_1)$, and *upArm*, which could not be decided by the quick, but myopic BBL alone. Respective signals are transmitted through mental and world models to the final behaviour execution processes.
This is not sufficient regarding deliberation as a decoupled decision problem to minimise conflicts and optimise the performance of the BBL machinery. For example, the BBL collision-avoidance desire *DES*2 is inconsistent with the deliberative goal of approaching another robot for box exchange. For example, tactical soccer actions in the RoboCup simulation are hardly expressible as ordinary abstractions of soccer skills. For this purpose, LPL actions, such as the *suppressDodging* action at l_4, represent parameterisations of the BBL rather than simply activating BBL behaviours.
Abstract resources are the representation device that amalgamates the principle of meta-control with a coherent abstraction hierarchy. Their status, their allocation, and related parameters of the BBL's control process are maintained quite as regular fluents in the world model. As such they are accessible by the mental model:

$$holds(resourceValue(affect, intensity, \dot{2}), T_3, T_4, nil, \dot{0}, E, l_4)$$
$$holds(resourceValue(battery, charge, 5\dot{3}4), T_1, T_2, nil, \dot{0}, E, l_4)$$
$$holds(resourceAllocation(reflex, affect, intensity, \dot{2}), T_5, T_6, nil, \dot{0}, E, l_4)$$
$$holds(processPriority(behaviourExecution(dodgeLeft), l_6, 0.\dot{5}), T_7, T_8, nil, \dot{0}, E, l_4)$$

affect is an abstract resource that is consumed by a forklift's reactive reflex process for deciding about low-level goals. Depending on the selected inference option, e.g., invoking a moderately 'dangerous' *searchLine* or a heavily interacting *dodge* behaviour,

a respective amount of its *intensity* item is consumed. This value is $\dot{1}$ in the case of *searchLine* and $\dot{2}$ in the case of *dodge*. The BBL control process acknowledges the consumption of resources by incorporating *resourceValue* updates into the world model. The history of resources (and other control parameters, such as *resourceAllocation* and *processpriority*) is deterministically maintained by the following definition (and analogue definitions for, e.g., *changeResourceAllocation* and *changeProcessPriority*):

$$\begin{aligned}
&\tilde{\forall} causes(F, changeResourceValue(Resource, ResourceItem, Val_1),\\
&\qquad T_1, T_2, C, B, E, l_4) \equiv\\
(DOMCAU8)\quad &\tilde{\exists} F \doteq resourceValue(Resource, ResourceItem, Val_1)\\
&\vee \tilde{\exists} \neg Val_1 \doteq Val_2 \wedge F \doteq not(resourceValue(Resource, ResourceItem, Val_2))\\
&\wedge holds(resourceValue(Resource, ResourceItem, Val_2), T_1, T_2, C, B, E, l_4)
\end{aligned}$$

Abstract resources, their allocation, and other control parameters are now subject to regular situation abstraction within the mental model and participate the deliberative reasoning in prediction, planning, and plan execution. By the following axioms, the forklift's local planning process can decide that, in order to successfully exchange a box, *suppressDodging* must be invoked which in turn decomposes into an update of the allocation of affect intensity. Then, the BBL reflex process will be able to decide *searchLine*, but no longer *dodge*. Since *suppressDodging* makes safe navigation impossible, the local planner must use the *allowDodging* action which reinstalls the original allocation before planning any further *moveArea* macro.

$$\begin{aligned}
(DOMSAB6)\quad &\tilde{\forall} decomposeHolds(safeExchange, T_1, T_2, C, B, E, l_4) \equiv\\
&\tilde{\exists} Val \leq 1 \wedge holds(resourceAllocation(reflex, affect, intensity, Val), T_1, T_2, C, B, E, l_4)
\end{aligned}$$

$$\begin{aligned}
(DOMCAU9)\quad &\tilde{\forall} causes(F, suppressDodging, T_1, T_2, C, B, E, l_4) \equiv\\
&\tilde{\exists} F \doteq safeExchange \vee \tilde{\exists} F \doteq not(safeNavigation)
\end{aligned}$$

$$\begin{aligned}
(DOMAAB2)\quad &\tilde{\forall} decomposeMacro(suppressDodging, T_1, T_2, C, B, E, l_4) \equiv\\
&\tilde{\exists} happens(changeResourceAllocation(reflex, affect, intensity, \dot{1}), T_1, T_2, C, B, E, l_4)
\end{aligned}$$

5.5.3 SPL: Social Model, Social Planning, and Protocol Execution

In a heterogeneous domain, the social problem of broad agents is not to be defined globally, rather appears as the economical adaption of the agent's deliberation (Von Neumann and Morgenstern's *theory of games and economic behaviour* [vNM44]) to the behaviour and especially to the conversations of other agents and humans (Searle's *speech act* theory [Sea69]). Even if restricting to artificial societies [Sin91] with a predefined communication language, such as KQML [LF97], this still appears to be an inherently complex exercise involving the prediction of other agent's state by communication *performatives* [CL95], full introspection into deliberative computations [Pet92], and decision-making about performatives [GS90, CP79, PA80]. Although possibly to formalise within a logic, such as BDI [RG91], this will most unlikely be an executable one.

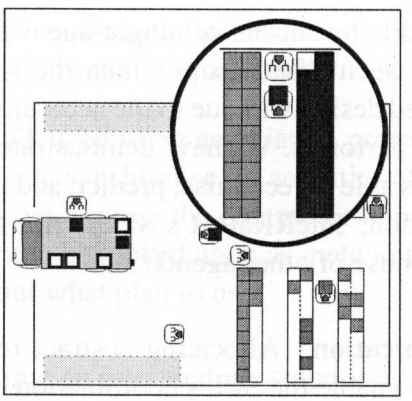

Figure 30: The Need for Social Coordination in the Automated Loading Dock

The forklifts in Figure 30 can be seen as a fully collaborative instance of a multi-agent system that needs to maximise deliveries throughout the whole loading dock. In the depicted case, one robot is blocking the other's path to the shelf and the robots can agree upon a multi-agent plan to exchange the box. This benevolent setting is similarly true for RoboCup, in which the overall aim of the game, to beat the opponent, is specified on the team level and can be reached by having players incarnating particular strategic roles.
In this context, [Mül96] constructed a runnable SPL for forklifts using *negotiation protocols* [RZ94]. Negotiation protocols, such as the *contract net protocol* [Smi80], are artificial conventions to coordinate distributed representations, e.g., to reach an agreement upon a multi-agent plan. They pre-structure the otherwise enormous space of possible interactions. Moreover, due to the bounded number and the fixed status of performatives within such a protocol, the modelling of complex communication semantics is largely abandoned.
To revisit all of these issues within the presented inferential framework exceeds the scope of this report and will presumably be a research topic on its own, such as other approaches to the matter demonstrate [DST99]. In the following, we describe the preliminary results that we obtained in 'emulating' InteRRaP's SPL within InteRRaP-R. Our ability to build working teams of agents in loading dock and RoboCup indicates that thinking about social processes in terms of causality may be an adequate enterprise. Indeed, most of the existing research around coordinating multi-agent activities [vM92, DL89, CLL93] relies on narrative-based representations in one or the other way. There is even no sensible alternative to treating other agents via default persistence; more than for regular fluents in the frame problem, the space of options for such 'mental fluents' otherwise prevents any reasoning. And finally, if we assume all agents to be InteRRaP-like artifacts, their mental attitudes are subject to causality by design. The rationale behind the resulting ALP & *HEC*-based SPL condenses into the four following points whose alternatives must be left to future investigations:

Coordination vs. Cooperation [MC91] distinguishes between jointly coming to an agreement on doing some action such as exchanging a box — the *coordination* problem which is doubtlessly the domain of the SPL — and jointly executing the action afterwards — the *cooperation* problem. In original InteRRaP [Mül96], the latter was

partly addressed in the LPL by means of single-agent plans, but partly also in the SPL[16] by means of multi-agent plans from which the single-agent plans have been extracted. This complicated design was due to the necessity of coordinating most of the cooperation that a forklift performs. We have demonstrated that this is not the general case: InteRRaP-R's LPL is able to recognise, predict, and assume multi-agent activities without explicit coordination. InteRRaP-R's SPL hence focuses just on coordinating the underlying LPL with those of other agents.

Reflection and Communication Allocating abstract resources, such as roles in a soccer team, is one tool to enable the SPL's coordination of the LPL. The other tool is a moderate reflection facility built into situation abstraction (see the paragraph on *The Social Model* below) which allows to create *mental fluents*, 'snap-shots' of regular fluents, events, and time-points in the LPL of various agents. Mental fluents are the basis for specifying performatives, for tracing social conditions , and for expressing coordination goals — currently under the support of highly domain-dependent background knowledge.

Flat Coordination One particular limitation of mental fluents is their not being nested, i.e., they do not range over the agents' social model processes. This is reasonable in terms of practicability, but problematic with respect to expressiveness. As [HM90] and others have elaborated, the 'when' and 'what' to coordinate suffers a severe bootstrapping problem, that is, in order to determine whether the mental states of two agents interact, there must be first some mutual agreement on what their actual mental states are. Similarly, negotiations can fail to reach an absolute consensus for which appropriate social goals ('try to coordinate as much as possible') must be defined on a higher level. Finally, social situations are often symmetrical such that it is not clear which agent should start the coordination.
In practical settings where the depth of the required agreement is bounded, e.g., forklifts should agree upon some joint LPL action, it suffices that agents mutually know a part of their mental models. The bootstrapping problem can then be resolved using an initial (meta-) negotiation about incomplete information (the QUERY protocol in [Mül96]) and election (the election protocol in [CR79]). We will not address reasoning about information exchange, failure of negotiations, and election in this thesis, as it would require another level of reflection (the *Conscious Planning Layer*?). Rather we handle these problems via domain representations that avoid symmetry, divergent negotiations, and lack of information.

Negotiation Protocols and Strategies Finally, we introduce SPL negotiation protocols as non-deterministic multi-agent macros built upon communication performatives and LPL-coordination actions by using the action abstraction facilities of *HEC*. By interactively planning and executing these multi-agent macros in the given social setting, the SPL realises agent-specific negotiation *strategies*. We especially focus on protocols for coordinating plans, such as box *exchange*, and resources, such as *roles*.

[16]This layer was thus called Cooperation Planning Layer (CPL). The above clarification suggests the change of name.

The apparent relation between multi-agent plans and negotiation protocols has already been anticipated in [Mül96] (Chapter 3.6, page 108):

> "There are strong parallels between negotiation protocols and joint plans: whereas plans restrict and synchronise the activities of agents in the world, protocols restrict and synchronise the communication process itself; thus, negotiation protocols can be looked upon as meta joint plans whose result is a decision, e.g., about what plan to use."

The Social Model Each time an intermediate solution is found in the mental model process, the social model receives a decoupled 'snap-shot' of several deliberative levels of representation. This is accomplished by a moderate reflection facility which extends first-order logic by (meta-)constants $'x$ referring to the partial content of a variable X, i.e., the established equalities, orderings, etc., which have been placed onto the variable. This is just one piece of second-order logic [Pet92] to which inference procedures have already been presented [CL94]. We do not discuss theoretical aspects here, rather remark that this is compliant with our (higher-order) implementation platform (see Section 6).
From the SPL perspective, mental *believes* fluents that are constructed within the following situation abstraction definitions at time t_i can be seen as ordinary terms that reflect the status of regular LPL fluents (*holds*), events (*happens*, *not*(*happens*)), and time-points $\dot{<}$. We sketch the *holds* and *not*(*happens*) cases with respect to two deliberative levels of abstraction:

$$\tilde{\forall} decomposeHolds(believes(rob_1, holds('f,' t_1,' t_2,' e,' l)), t_{i-1}, t_i, nil, \dot{0}, E, l_1) \equiv$$
$$\tilde{\exists} L \doteq l_4 \wedge holds(F, T_1, T_2, nil, \dot{0}, E, L) \qquad (DOMSAB7)$$

$$\tilde{\forall} decomposeHolds(believes(rob_1, not(happens('e,' a,' t_1,' t_2,' l))),$$
$$t_{i-1}, t_i, nil, \dot{0},' e, l_1) \equiv \qquad (DOMSAB8)$$
$$\tilde{\exists} L \doteq l_5 \wedge \neg happens(E, A, T_1, T_2, L)$$

Tolerating the restricted expressiveness, these definitions do not build up *believes* in an arbitrarily nested and compositional fashion, e.g., believes about believes or believes about disjunctive formulae. Partly, compositional information is implicitly present in more abstract representations. Partly, conjunctions can be reflected in order to make the mechanisms presented in this subsection more practical, e.g., by coordinating complex plans instead of actions.
We obtain a simple history mechanism of the social model in which updates are modelled via *changeBelieve* miracles initiating new believes and terminating *incompatible* ones. *incompatible* is a rule of thumb which determines what must have changed in the commitments of an agent's mental model, but which is not constructing a necessarily consistent history.

$$\tilde{\forall} causes(changeBelieve(Ag, F_1), F_2, T_1, T_2, C, B, E, l_1) \equiv$$
$$\tilde{\exists} F_2 \doteq believes(Ag, F_1)$$
$$\vee \tilde{\exists} F_2 \doteq not(believes(Ag, F_3)) \qquad (DOMCAU10)$$
$$\wedge holds(believes(Ag, F_3), T_1, T_2, C, B, E, l_1)$$
$$\wedge incompatible(Ag, F_1, F_3, T_1, T_2, C, B, E)$$

$$
\begin{aligned}
&\tilde{\forall} incompatible(Ag, F_1, F_2, T_1, T_2, C, B, E) \equiv \\
&\quad \tilde{\exists} F_1 \doteq holds(F, T_3, T_4, E_1, L) \wedge F_2 \doteq holds(not(F), T_5, T_6, E_2, L) \\
&\qquad \wedge holds(believes(Ag, \dot{<}(T_3, T_6)), T_1, T_2, C, B, E, l_1) \\
&\qquad \wedge holds(believes(Ag, \dot{<}(T_5, T_4)), T_1, T_2, C, B, E, l_1) \\
&\quad \vee \tilde{\exists} F_1 \doteq happens(E_2, A, T_3, T_4, L) \wedge F_2 \doteq not(happens(E_2, A, T_5, T_6, L) \\
&\qquad \wedge holds(believes(Ag, \dot{<}(T_3, T_6)), T_1, T_2, C, B, E, l_1) \\
&\qquad \wedge holds(believes(Ag, \dot{<}(T_5, T_4)), T_1, T_2, C, B, E, l_1) \\
&\quad \vee \tilde{\exists} F_1 \doteq \dot{<}(T_3, T_4) \wedge F_2 \doteq \dot{<}(T_4, T_3)
\end{aligned}
\tag{INCOMP}
$$

It is to note that our formalism does not introduce mental fluents for goals and intentions. According to BDI logic [RG91] and its Kripke-style semantics, goals are particular 'sub-worlds' of optional beliefs, intentions are sub-worlds of optional goals. By this construction, it can be distinguished between unavoidable situations and future goals, between anticipated occurrences and decided intentions. We cannot uphold this distinction, because *believes* do always refer to one particular option within the mental model. Hence, we could only assume all believes about future events to be the intentions of the agent, about future states to be its goals.
Communication is needed in order to derive mental fluents about other agents' LPL. Basic communication facilities for perceiving and issuing performatives are behaviour execution processes within the BBL ($hear(Ag)$ and $say(Ag, W)$ at level l_7). These are are allowed to send *ActionSignal*

$$\Delta ::= happens(say(rob_1, tell(holds(at(box_1, shelf_3),' t_1,' t_2,' e, l_2))), T_3, T_4, l_7)$$

and to receive *PerceptionSignal*

$$G' ::= holds(heard(rob_2, tell(happens('e_3, downArm(rob_2),' t_5, t_i, l_4))), T_7, T_8, E, l_7)$$

to and from the environment. As a short-cut, their results are immediately lifted from the world model into the social model (*DOMAAB*2 and *DOMSAB*12) where the causal interpretation presumes benevolence, i.e., a proper belief is the necessary precondition for uttering a *tell* performative (*DOMCAU*11).

$$
\begin{aligned}
&\tilde{\forall} decomposeMacro(hear(Ag), T_1, T_2, l_1) \equiv \\
&\quad \tilde{\exists} happens(hear(Ag), T_1, T_2, l_7)
\end{aligned}
\tag{DOMAAB2}
$$

$$
\begin{aligned}
&\tilde{\forall} decomposeHolds(heard(Ag, W), T_1, T_2, C, B, E, l_1) \equiv \\
&\quad \tilde{\exists} holds(heard(Ag, W), T_1, T_2, C, B, E, l_7)
\end{aligned}
\tag{DOMSAB9}
$$

$$
\begin{aligned}
&\tilde{\forall} causes(hear(Ag), heard(Ag, tell(F)), T_1, T_2, C, B, E, l_1) \equiv \\
&\quad \tilde{\exists} holds(believes(Ag, F), T_1, T_2, C, B, E, l_1)
\end{aligned}
\tag{DOMCAU11}
$$

Benevolence usually goes further by possibly adopting the thus reported believes, for example by integrating an execution report of a partner agent into the mental model to support cooperative prediction, planning, and plan execution. Another example are user-given delivery commands ($tell(user, unloadCommand(box_1, l_2))$) which we expect to be carried out by the forklifts LPL according to the preceding subsection. But, it is important not to so naively, because partner agents have an as incomplete

view of the world, pursue different goals, and work on possibly inconsistent intentions. Since the social model is decoupled from the LPL, it is however able to force a (non-monotonic) re-orientation of the mental model in order to accept observations, remove goals, and change intentions there.

$$\begin{array}{ll} & \tilde{\forall} holds(believes(Ag, F), t_{i-1}, t_i, nil, \dot{0}, E_1, l_1) \wedge \neg Ag \doteq rob_1 \\ (ADOPTE) & \wedge F \doteq happens('e_2, downArm(Ag),' t_3,' t_4,' l) \supset \\ & \quad \tilde{\exists} happens(E_2, Ag, T_3, T_4, L) \end{array}$$

To decide what is worth of being integrated and what is not, we again rely on a rule of thumb. The above *ADOPTE* constraint (and analogues for fluents and time-points) integrates other agents' reported believes about past, present, and future events where the criterion is that each agent is held capable of controlling its own actions. More general incorporation principles could emerge from ongoing research on distributed databases [Sub94].

Equally important to the adoption of information is to forget about mental states over time; the temporal horizon ϵ for rolling-up the social model ranges about one minute back and forth. Mental states of agents are likely to change unobservably which somehow stands in contrast to strict inertia. The roll-up mechanism can be thus be understood as a way of circumventing the implausible infinite persistence of any fluent.

Social Planning: Social Goals and Initiating Negotiation How to incorporate told information is a fundamental part of the agent's coordination strategy. When to engage into some communication, which coordination convention (negotiation protocol) to use, and what to say during such a conversation are the other parts. Because of incomplete perception, these decisions can take a complicated shape: To determine that exchanging a box is useful, forklifts usually have to first exchange information about their current environment, their related goals, and their relevant intentions, i.e., they have to coordinate their social models before coordinating their mental models. For that purpose, [Mül96] used an additional level of meta-negotiation (the QUERY protocol) whose formalisation exceeds the expressiveness of mental fluents. Instead, we install desires (*DES*3) into the forklifts' SPL which aim at informing (the *said* fluent) a facing robot of relevant parts of their mental model. By *DOMCAU*12, the social planner invokes a *say* action that immediately decomposes into a corresponding BBL activity.

$$\begin{array}{ll} & \tilde{\forall} holds(believes(rob_2, holds(ahead(Ag), T_3, T_4, E_1, l_2)), T_1, T_2, nil, \dot{0}, E_2, l_1)) \\ (DES) & \wedge holds(believes(rob_2, holds(at(Ag, shelf_3), T_3, T_4, E_1, l_2)), T_1, T_2, nil, \dot{0}, E_2, l_1) \\ & \wedge holds(believes(rob_1, holds(free(\dot{8}, \dot{7}), T_3, T_4, E_1, l_3)), T_1, T_2, nil, \dot{0}, E_2, l_1) \supset \\ & \quad \tilde{\exists} said(Ag, tell(holds(free(\dot{8}, \dot{7}), T_3, T_4, E_1, l_3)), T_3, T_4, nil, \dot{0}, E_2, l_1) \end{array}$$

$$\begin{array}{ll} (DOMCAU12) & \tilde{\forall} causes(say(Ag, tell(W)), said(Ag, tell(W)), T_1, T_2, C, B, E, l_1) \equiv \\ & \quad \tilde{\exists} holds(believes(rob_1, W), T_1, T_2, C, B, E, l_1) \end{array}$$

By this design and the benevolent incorporation strategy, forklifts are able to complete their social models, at the same time re-orientate their LPL. For example, as soon as the robot delivering the box adopts the reported belief $free(\dot{8}, \dot{7})$, its local planning process

refines the *transport* commitment at level l_3 (that is dealing with the reachability of areas) with the multi-agent macro $happens(exchange(rob_1, rob_2, box_1, shelf_3), T_1, T_2, l_3)$. This is because involving the other agent became less expensive than doing it on its own (where inability amounts to infinite costs).
By comparing its own mental state with the one reported by the partner agent, the delivering forklift then notices that it must reach an agreement about the generated multi-agent plan. These statements are treated at the level of abstraction l_0. The fluent $not(coor(Ag_1, Ag_2, F_1, F_2))$ (explained by the miracle *disCoor* in *DOMCAU*11) describes that agents Ag_1, Ag_2 exhibit believes F_1, F_2 which are uncoordinated, such as different opinions about the occurrence of the joint *exchange* (*DOMSAB*10). The unique agent Ag_1 then initiates the coordination effort by the SPL desire $\forall holds(not(coor(rob_1, Ag, F_1, F_2)), T_1, T_2, nil, \dot{0}, E, l_0) \supset holds(coor(rob_1, Ag, F_1, F_2), T_3, T_4, nil, \dot{0}, E, l_0)$ and lets the social planning process to enter the coordination protocol $coorProt(rob_1, Ag_2, F_1, F_2)$ (*DOMCAU*12).

(*DOMSAB*10)
$$\begin{aligned}&\tilde{\forall} decomposeHolds(not(coor(Ag_1, Ag_2, F_1, F_2)), T_1, T_2, B, E, l_0) \equiv \\ &\quad \tilde{\exists} F_1 \doteq happens(exchange(Ag_1, Ag_2, Box, Area), T_3, T_4, l_3) \\ &\quad \wedge holds(believes(Ag_1, F_1), T_1, T_2, C, B, E, l_1) \\ &\quad \wedge holds(believes(Ag_2, not(F_1)), T_1, T_2, C, B, E, l_1)\end{aligned}$$

(*DOMCAU*13)
$$\begin{aligned}&\tilde{\forall} causes(disCoor(Ag_1, Ag_2, F_1, F_2), F, T_1, T_2, C, B, E, l_0) \equiv \\ &\quad \tilde{\exists} F \doteq not(coor(Ag_1, Ag_2, F_1, F_2))\end{aligned}$$

(*DOMCAU*14)
$$\begin{aligned}&\tilde{\forall} causes(coorProt(Ag_1, Ag_2, F_1, F_2), F, T_1, T_2, C, B, E, l_0) \equiv \\ &\quad \tilde{\exists} F \doteq coor(Ag_1, Ag_2, F_1, F_2)\end{aligned}$$

Protocol Planning and Execution: Coordinating Believes In [Mül96], a negotiation mechanism for reaching an agreement about multi-agent plans is developed, the *joint plan protocol* (Figure 31, lower-left part). Its principle goes back to the *monotonic concession protocol* of [RZ94] and must not be restricted to plans, but could cover any representation such as mental fluents. One of the involved agents (the *leader* as opposed to the *follower*) that has first recognised the coordination problem constructs a *negotiation set* which is a finite set of optional solutions to the coordination problem. In our example, these options are that either both agents adopt the *exchange* event or they both reject it. More complex negotiation sets can be constructed by *plan critique* methods [vM92].
The state diagram in Figure 31 is a shared convention between the agents according to which they take initiative (the state nodes) and perform communication performatives (the transitions) about the negotiation set. Non-determinism in the protocol has its source in the actual choice of the performative and the selection of its content (plan, element of the negotiation set). The individual *negotiation strategies* of the agents determine concrete 'runs' through the protocol during which the negotiation set is incrementally shrinked until both agents agree on a particular solution.
Reasoning about state and transition is central to our logic-based reconstruction of InteRRaP-R. Hence, it is suggesting to think of diagrams such as used to describe negotiation protocols of the SPL as multi-agent macros quite similar to the *transport* and *exchange* abstractions installed in the LPL. In the upper part of Figure 31, the

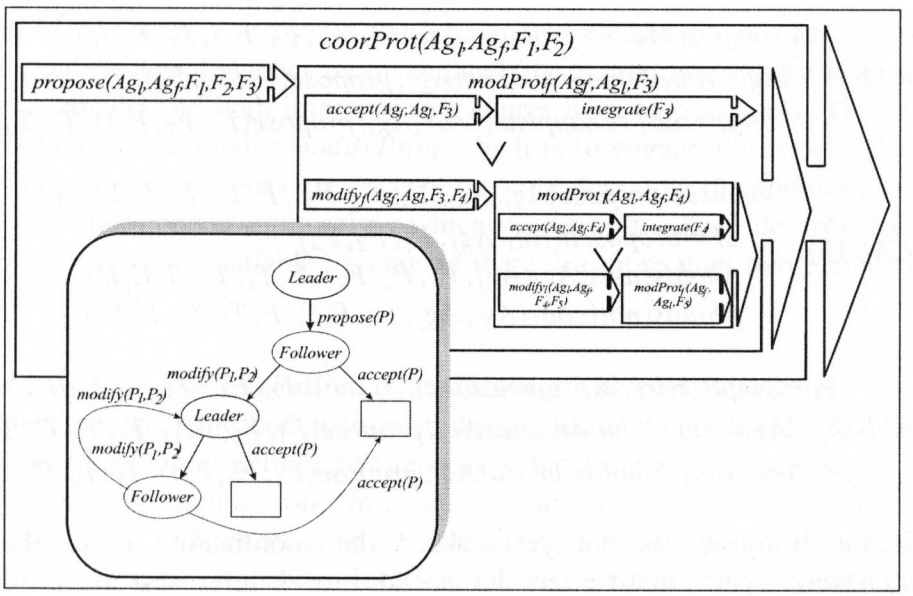

Figure 31: The Coordination Protocol: State Diagram and Multi-Agent Macro

logic structure of the *coordination protocol* macro (*coorProt*) is outlined that transfers the transitions of the original diagram into communication and coordination primitives (*propose*, $modify_l$, $modify_f$, *accept*, *integrate*). Sub-graphs of the diagram are realised as recursive sub-macros ($modifyProt_l$, $modifyProt_f$). *coorProt* models non-determinism by disjunctive decomposition and quantified arguments of action types.
To emulate the agents' negotiation strategy, the idea is to introduce the (intermediate) negotiation set as a set of high-level fluents $nset(Ag_l, Ag_f, F_1, F_2, F_3)$ at level l_0 relating the possible solutions F_3 to the actual coordination problem. These fluents are initiated by the miracle *buildNset*.

$$\begin{aligned}
&\tilde{\forall} decomposeHolds(nset(Ag_l, Ag_f, F_1, F_2, F_3), T_1, T_2, B, E, l_0) \equiv \\
(DOMSAB14)\quad &\tilde{\exists} F_3 \doteq F_1 \wedge holds(not(coor(Ag_l, Ag_f, F_1, F_2)), T_1, T_2, B, E, l_0) \\
&\vee \tilde{\exists} F_3 \doteq F_2 \wedge holds(not(coor(Ag_l, Ag_f, F_1, F_2)), T_1, T_2, B, E, l_0)
\end{aligned}$$

$$\begin{aligned}
&\tilde{\forall} causes(buildNset(Ag_l, Ag_f, F_1, F_2, F_3), F, T_1, T_2, B, E, l_0) \equiv \\
(DOMCAU12)\quad &\tilde{\exists} F \doteq nset(Ag_l, Ag_f, F_1, F_2, F_3)
\end{aligned}$$

Then, the protocol can be run distributedly by the situated interplay of both agents' social model, social planning, and protocol execution processes that make inferences about how the protocol actions manipulate the negotiation set. Hence, the individual negotiation strategy is mapped to the individual selection strategy applied by the social inference processes. For example, the *propose* action decomposes in *DOMAAB*3 into either active (for the leader) or passive (for the follower) communication. *propose* refers to some member of the negotiation set in its precondition. This member is picked by the search rule of the social planner and marked as *proposed*. Since the leader's planner furthermore chooses the most cost-effective hypothesis, it simply assumes the follower to accept; its protocol executer hence issues *propose* and waits for *accept*.

$$\tilde{\forall} decomposeMacro(propose(Ag_l, Ag_f, F_1, F_2, F_3), T_1, T_2, l_0) \equiv$$
(*DOMAAB*3) $$\tilde{\exists} Ag_l \doteq rob_1 \wedge happens(say(Ag_f, propose(F_1, F_2, F_3)), T_1, T_2, l_1)$$
$$\vee \tilde{\exists} Ag_f \doteq rob_1 \wedge happens(hear(Ag_l, propose(F_1, F_2, F_3)), T_1, T_2, l_1)$$

(*DOMCAU*15)
$$\tilde{\forall} causes(propose(Ag_l, Ag_f, F_1, F_2, F_3), F, T_1, T_2, B, E, l_0) \equiv$$
$$\tilde{\exists} F \doteq proposed(Ag_l, Ag_f, F_1, F_2, F_3)$$
$$\wedge holds(nset(Ag_l, Ag_f, F_1, F_2, F_3), F, T_1, T_2, B, E, l_0)$$
$$\wedge holds(not(coor(Ag_l, Ag_f, F_1, F_2)), F, T_1, T_2, B, E, l_0)$$

$$\tilde{\forall} decomposeHolds(proposed(Ag_l, Ag_f, F_1, F_2, F_3), T_1, T_2, C, B, E, l_0) \equiv$$
(*DOMSAB*13) $$\tilde{\exists} Ag_l \doteq rob_1 \wedge holds(said(Ag_f, propose(F_1, F_2, F_3)), T_1, T_2, C, B, E, l_1)$$
$$\vee \tilde{\exists} Ag_f \doteq rob_1 \wedge holds(heard(Ag_l, propose(F_1, F_2, F_3)), T_1, T_2, C, B, E, l_1)$$

Even if the follower has not yet realised the coordination need, the arrival of the *proposed* performative lets its social model infer that there is a conflict with respect to F_1, F_2 pending, lets it create the identical negotiation set *nset*, and lets the social planner engage into the *modProt*1 sub-protocol (*DOMCAU*16) by the desire $\tilde{\forall} holds(not(coor(Ag, rob_2, F_1, F_2)), T_1, T_2, nil, \dot{0}, E, l_0) \supset holds(coor(Ag, rob_2, F_1, F_2), T_3, T_4, nil, \dot{0}, E, l_0)$.
While the SPL of the leading agent waits, the follower's planner has the option to *accept* and *integrate* the result. *integrate* (*DOMAAB*4) is a coordination action which re-orientates the mental model. *accept* decomposes into a performative whose transmission resumes the leader's protocol execution process that integrates the information there. In this case, the protocol is terminated and both partners adopt $believes(rob_1, F_3) \wedge believes(rob_2, F_3)$.

(*DOMCAU*16)
$$\tilde{\forall} causes(modifyProt_f(Ag_l, Ag_f, F_3), F, T_1, T_2, B, E, l_0) \equiv$$
$$\tilde{\exists} F \doteq coord(Ag_l, Ag_f, F_1, F_2)$$
$$\wedge holds(proposed(Ag_l, Ag_f, F_1, F_2, F_3), F, T_1, T_2, B, E, l_0)$$

(*DOMAAB*4)
$$\tilde{\forall} decomposeMacro(integrate(happens('e,' a,' t_3,' t_4,' l)), T_1, T_2, l_0) \equiv$$
$$\tilde{\exists} happens(E, A, T_3, T_4, L)$$

The follower's planner has also the option to $modify_f$ the proposal, i.e., to make a counter-proposal from the negotiation set according to its selection strategy and the estimated costs of the coordination solution. The preconditions of $modify_f$ require some initial proposal from the leader and some counter-offer that is a different element of the negotiation set. Its effects are that the new solution is now *proposed* and that the previous proposal is no longer in the negotiation set *nset*. The follower passes the initiative then back to the leader by suspending on the sub-protocol $modProt_l$ under the assumption that the leader accepts the counter-proposal.

(*DOMCAU*17)
$$\tilde{\forall} causes(modify_f(Ag_f, Ag_l, F_3, F_4), F, T_1, T_2, B, E, l_0) \equiv$$
$$\tilde{\exists} \neg F_3 \doteq F_4 \wedge F \doteq not(nset(Ag_l, Ag_f, F_1, F_2, F_3))$$
$$\wedge holds(nset(Ag_l, Ag_f, F_1, F_2, F_4), F, T_1, T_2, B, E, l_0)$$
$$\wedge holds(proposed(Ag_l, Ag_f, F_1, F_2, F_3), T_1, T_2, B, E, l_0)$$
$$\vee \tilde{\exists} \neg F_3 \doteq F_4 \wedge F \doteq proposed(Ag_f, Ag_l, F_1, F_2, F_4) \ldots$$

At the time that $modify_f$ arrives at the leader, the selected options in both protocol execution and the social planner are invalidated (the assumption *nset* for *integrate* does not hold anymore) and both inference procedures have to backtrack. They take the second option to decompose $modifyProt_f$ which is to engage into $modifyProt_l$. The social planner decides about either accepting and integrating the counter-proposal or making a counter-counter-proposal by which we return to above described situation. Counter-proposals are available as long as there are more than two elements in the negotiation set left.

Protocol Planning and Execution: Coordinating Abstract Resources Especially with respect to realising RoboCup team strategies, social control is thought as the long-term parameterisation of deliberative processes that realise, e.g., local soccer tactics. Abstract resources, such as *roles* in a soccer team, that describe constraints on the possibilities of LPL inferences are a suitable interface for this part of the coordination aspect. Resource value, resource allocation, related performance data, and parameters of the LPL control process are accessible as ordinary fluents in the mental model:

$$holds(resourceValue(roles, attacker, \dot{1}), T_3, T_4, nil, \dot{0}, E, l_2)$$
$$holds(resourceValue(roles, goalie, \dot{1}), T_3, T_4, nil, \dot{0}, E, l_2)$$
$$holds(processPerformance(attackTactic, \dot{0.2}), T_1, T_2, nil, \dot{0}, E, l_2)$$

LPL processes frequently access abstract resources in order to make inferences. For example, to plan some offensive LPL tactic (*attackTactic*) there must be some positive value for the *attacker* item of *roles*. In order to enable movement in the penalty area and catching the ball (*keepGoal*) there must be some *goalie* item accessible. An agent which has both *goalie* and *attacker* resources at its disposal will most probably behave not optimal, i.e., try to run across the whole field when possessing the ball. This is revealed by an expectedly bad performance of executing *attackTactic*. The task of the SPL is to improve the average performance of LPL processes by coordinating the resource allocations throughout the whole agent team. For this purpose, the social model is able to represent resource values and average performances of several agents.

$$(DOMSAB14)\ \tilde{\forall} decomposeHolds(resourceValue(Ag, R, I, V), T_1, T_2, C, B, E, l_1) \equiv \tilde{\exists} Ag \doteq player_1 \wedge holds(resourceValue(Ag, R, I, V), T_1, T_2, C, B, E, l_2)$$

$$(DOMSAB15)\ \tilde{\forall} decomposeHolds(performance(Ag, P), T_1, T_2, C, B, E, l_1) \equiv \tilde{\exists} Ag \doteq player_1 \wedge P \doteq (P_1 \dot{+} P_2 \dot{+} \ldots \dot{+} P_i) \dot{/} i \wedge holds(processPerformance(attackLeft, P_1), T_1, T_2, C, B, E, l_2) \wedge holds(processPerformance(local\_planning, P_2), T_1, T_2, C, B, E, l_2) \ldots$$

Finding an optimal solution to this resource distribution problem is a complex task as it requires a lot of detailed information about the global situation, the internal operation of agents, the behaviour of the opponent, etc. It is to question whether such data can be gathered, even more whether the result of such a computation can have any impact in a steadily changing domain. Just as a coach frequently re-arranges a soccer team's strategy by observation, we are better off with a robust mechanism which helps to dynamically adjust resource assignments according to performance reports. One

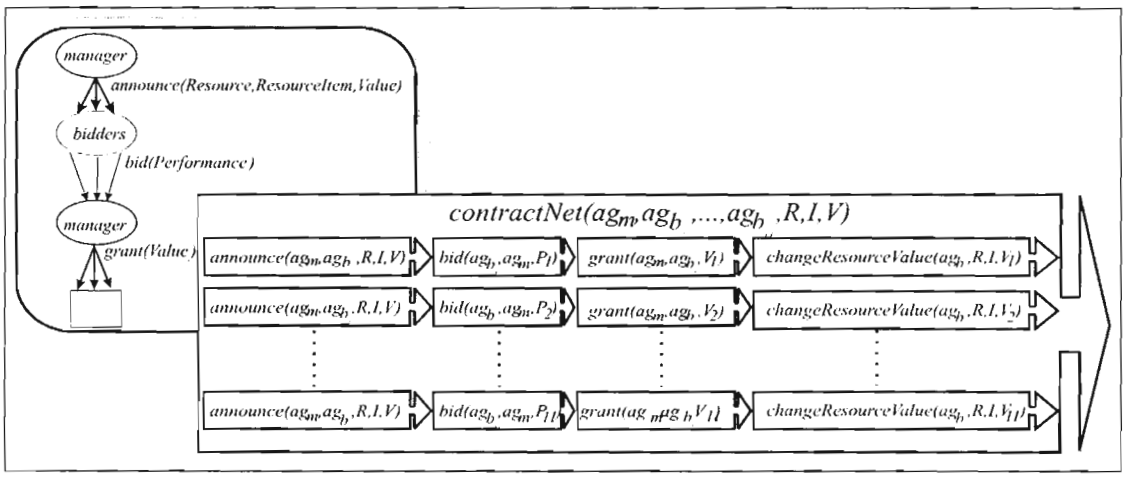

Figure 32: The Contract-Net Protocol: State Diagram and Multi-Agent Macro

such mechanism is the *contract-net protocol* [Smi80] (Figure 32) in which a dedicated manager agent ag_m (the coach) coordinates a set of bidder agents $ag_1, \ldots, ag_{11}$ (the soccer players). Thanks to its ability to represent arbitrarily interleaved actions, *HEC* is capable of representing the contract-net as a compact multi-agent macro.
Originally invented for task assignment, the contract-net principle is suitable for distributing any type of representation, including abstract resources. Applied to our domain, the manager keeps a central (strategic) representation of team resources, i.e., how much overall *defender*, *attacker*, and *goalie* items should be available as team *roles*, and of team performance. There is a domain-dependent heuristic which indicates when performance goes down ($not(optimal(R,I))$, *DOMSAB*16) and which lets the manager issue an *announce* performative (*DOMCAU*18) for these items. This informs the bidders that resource coordination is pending to which they react by reporting their average performance by the *bid* performative (*DOMCAU*18). According to its coordination strategy, the manager distributes the announced items to the best performing agents.

(*DOMCAU*18)
$$\begin{aligned}&\tilde{\forall} causes(announce(coach, Ag, R, I, V), F, T_1, T_2, C, B, E, l_0) \equiv \\ &\quad \tilde{\exists} F \doteq announced(coach, Ag, R, I, V) \\ &\quad \wedge holds(not(optimal(R, I)), T_1, T_2, C, B, E, l_0) \\ &\quad \wedge holds(resourceValue(R, I, V), T_1, T_2, C, B, E, l_0)\end{aligned}$$

(*DOMSAB*16)
$$\begin{aligned}&\tilde{\forall} decomposeHolds(not(optimal(R, I)), T_1, T_2, C, B, E, l_0) \equiv \\ &\quad \tilde{\exists} holds(resourceValue(Ag, R, I, V), T_1, T_2, C, B, E, l_1) \\ &\quad \wedge holds(performance(Ag, P), T_1, T_2, C, B, E, l_1) \\ &\quad \wedge \neg V \doteq \dot{0} \wedge P \dot{\leq} 0.\dot{3}\end{aligned}$$

(*DOMCAU*19)
$$\begin{aligned}&\tilde{\forall} causes(bid(Ag, coach, P), bade(Ag, coach, P), T_1, T_2, C, B, E, l_0) \equiv \\ &\quad \tilde{\exists} holds(performance(Ag, P_2), T_1, T_2, C, B, E, l_1)\end{aligned}$$

5.6 Bottom Line

Underlying a theory of time and action, such as *HEC*, with an advanced inferential framework, such as ALP [KKT93], delivers executable specifications of domain-independent agent functionalities, such as prediction, planning, and execution, at several representational horizons, such as found in reactive, deliberative, and social reasoning. This aspect of Cognitive Robotics completes the rigorously formal description of hybrid InteRRaP-R a sound and efficient implementation of which should be derived in the following section. At the same time, we are provided with a clean programming surface in terms of domain representations whose adequacy for our representative scenarios will be documented in Section 7.

The feasibility of the approach is supported by two factors. The first one is the situated reasoning that is performed by ALP, more specifically the presented IFF and FFI proof procedures. To make a closed-world assumption rather than to introduce a tremendously complex frame of possibilities is important for practical inferences, such as realised in SLDNF [Cla78]. ALP's additionally staying incremental with respect to particular occurrences in the world is important because of ubiquitous and unavoidable ignorance.

The second factor concerns the operational considerations which make the inferences fast and mutually compatible. We have presented persistent, but complete selection strategies for the various inference services. We have discussed their connection to COOP exception handling for detecting inconsistent states, for recovery and for roll-up. Finally, we have envisaged encapsulation, i.e., the individuation of the otherwise too general logic framework into the adaptive interplay of specialised, but interactive reasoning processes. These issues are usually neglected by relevant Cognitive Robotics research, such as [Esh88, KS96b, Sha89, Sha97a].

6 Implementation: CP

Broad agents are a type of software systems that take too much effort to be programmed from scratch every time again. Moreover, they cannot be easily understood at the level of program code. This led us to introduce high-level design concepts to organise state and computation of InteRRaP-R in both declarative and operational ways. Since the applied design methodology is mainly formal, we have gathered by now a largely unambiguous 'agent interpreter' in terms of COOP and embedded ALP & *HEC*. This interpreter determines a set of domain-dependent data structures and describes their step-wise treatment by domain-independent transitions.

The aim of this enterprise is to create concrete systems running on standard computers programmed in a standard programming language, hence to *implement* the developed specification of architecture, computational model, theory, and inference. At this point, we could rely on the reader's trust in the presented concepts being implementable and on his or her intuition about how they have been implemented. However, for an approach to should avoid the fuzziness of previous research, this would be too simpleminded.

On the other hand, installing a verified connection between specification and final program, such as propagated by formal engineering, may be possible for critical and small sub-modules of a software system. For a compositional framework, such as hybrid

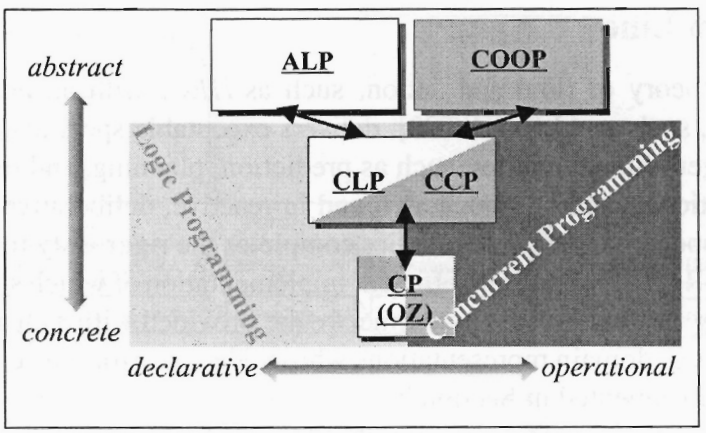

Figure 33: Constraint Programming as an Implementation Methodology

InteRRaP-R with its several thousand lines of source code, this would certainly exceed the scope of this thesis. Hence, the current section takes an intermediate stance and discusses useful realisations, limitations, and extensions of the high-level concepts within modern programming languages, but does not engage into verification issues at the code level.
For this purpose, we look at 'Constraint Programming' (CP, Subsection 6.3) [MR95, Fre96, Smo97, MP98] languages, such as Oz [Smo95], because the fruitful reconciliation of declarative inferences and operational computations lies at their very heart (Figure 33). Oz builds on traditions from 'Constraint Logic Programming' (CLP, Subsection 6.1) that investigate the operational control of logic-based settings. CLP's close relation to ALP (Subsection 6.4) is the key to implement *HEC* inferences. Oz also builds on traditions from 'Concurrent Constraint Programming' (CCP, Subsection 6.2) that provide a unified inferential frame for, e.g., operations research, graph theory, and distributed computing. By its emphasis on concurrency, the CCP model is predestined for the implementation of interactive COOP processes (Subsection 6.5).
Hence, the current section demonstrates our conceptual framework to be implementable in modern programming environments in a sound way. It presents basic recipes how to efficiently implement the agent-level concepts within a constraint-based environment. And it finally discusses in Subsection 6.7 to what extend the derived implementation is compliant to the computational and inferential model of InteRRaP-R developed so far.

6.1 Constraint Logic Programming: CLP

We have already motivated the central role of unification in logic programming (LP) for which a quasi-linear algorithm [MM82] can be deployed. Conceptually, however, unification is still thought of as a deductive inference upon logic formulae, i.e., $\doteq$ literals, that are interpreted over the Herbrand universe, i.e., under the *CET* theory.
The general usefulness of combining a declarative problem-solving machinery, such as definite clauses, with an operationally efficient treatment of a logic sub-language, such as handling equality by unification, has been recognised and formalised by Jaffar & Lassez [JL87] in their 'Constraint Logic Programming' (CLP(X) — Figure 34) frame-

$BasicConstraint ::= Constant(Term^*)$
$ProperLiteral ::= Constant(Term^*) \setminus BasicConstraint$
$ComplexConstraint ::= \bot \mid \top \mid BasicConstraint \wedge ComplexConstraint$
$Goal ::= \top \mid ProperLiteral \wedge Goal$
$Clause ::= \tilde{\forall} ProperLiteral \subset \tilde{\exists} ComplexConstraint \wedge Goal$

Figure 34: Definite Constraint Logic Clauses

work. CLP(X) identifies particular literals within the logic program, such as equalities, that are called basic *constraints*.
Theoretically, constraints are not defined by the program, but parameterised via a given theory X or, alternatively, by a set of corresponding first-order structures X. Hence, CLP(*CET*) is the instantiation of CLP to the usual Herbrand semantics of LP.
Inferentially, CLP applies variation of SLD-resolution $\vdash_P$: $ComplexConstraint \wedge Goal \leftrightarrow ComplexConstraint \wedge Goal$ that operates on regular literals and conjunctions of basic constraints *BasicConstraint*. In the following SLC step where $Co_i : ComplexConstraint$, constraints are not resolved by P, but by a separate check for satisfiability $\vdash_{tell}$: $ComplexConstraint \times ComplexConstraint \rightarrow ComplexConstraint$.

$$\frac{\begin{array}{lll} Co_1 \wedge G_1 \wedge & \pi(P) \ni & Te_i\sigma = Te'_i\sigma \\ C(Te_1, \ldots, Te_n) & \tilde{\forall} C(Te'_1, \ldots, Te'_n) \subset \tilde{\exists} Co_2 \wedge G_2 & (Co_1\sigma, Co_2\sigma) \vdash_{tell} Co_3 \end{array}}{Co_3 \wedge G_2\sigma \wedge G_1\sigma} \text{ SLC}$$

$\vdash_{tell}$ can be thought of as a 'canonicaliser' that takes two composite constraints as its arguments and outputs a new representation that is equivalent to the conjunction of the given constraints. In the case of unsatisfiability, it outputs $\bot$ and fails the actual derivation. Otherwise, $\vdash_{tell}$ outputs a constraint that is at least as strong as each of the input constraints. In other words, new information Co_2 is 'told' to the intermediate constraint 'store' Co_1 and results in a new constraint store Co_3. That is why $\vdash_{tell}$ is called the *tell* operation. A successful SLC derivation starts with a goal G under an empty constraint store and terminates with a satisfiable store $G \vdash_P^* C$.
It is apparent that algorithms to implement $\vdash_{tell}$ have to address particular semantical demands in order to support a sound and complete inference service, namely they have to be sound and complete with respect to the constraint language by themselves. The unification algorithm is such a procedure that has been extended in Prolog II to cover inequality constraints $\neg X \doteq Te$ and rational, infinite trees according to the *RAT* theory [Mah88].
Another example is the *Simplex* algorithm in Prolog III's CLP(Q) framework that can be used to handle linear constraints of the form $\Sigma_i a_i \dot{*} X_i \dot{\geq} b$ over rational numbers. For maintaining a partial order $T_1 \dot{<} T_2$ on integer-valued variables (finite domains, CLP(*FD*)), a quadratically-complex subset of the *Warshall*-procedure can be employed.
We recognise that $\vdash_{tell}$ is an interface between LP and widespread research traditions in AI, operations research, and mathematical programming that run under the title *constraint satisfaction algorithms* [Wal97, Mac88]. Hence, the primary advantage of

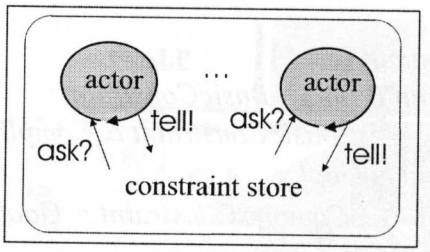

Figure 35: Concurrent Constraint Programming: The Actor Model

separating constraints from defined predicates and of separating the constraint store from the problem solver is that one may use specialised satisfaction algorithms for particular domains that are otherwise only ceremoniously expressible as inferences over the Herbrand universe.
The second advantage of this separation is that the balance between problem solver and constraint solver may be usefully manipulated by the language designer. It is possible to initially experiment with a purely clause-based theory. By gaining more insight into the problem specification, more and more specialised inferences can migrate from the problem solver into the constraint-solver. Thus, CLP is the straightforward descendant of LP in terms of rapid-prototyping.

6.2 Concurrent Constraint Programming: CCP

Experiences with CLP languages of the first generation such as Prolog II, Prolog III, and CHIP have shown that languages that should be extendible also by application programmers and that should contain more sophisticated constraints, such as over finite domains, finite sets, record-like feature structures, pseudo-booleans, and real numbers, the control facilities of the tell operation are too weak. For example, checking the satisfiability of finite domain constraints is an NP-complete problem and renders CLP(*FD*) unusable in practice.
How could we build complete inference systems from approximate and specialised constraint solvers that cannot immediately output an equivalent expression for their input constraints because of tractability and representational purposes? Already [Mah87] has proposed to regard constraint solvers not as one-shot procedures that release all of their information at once, but rather as persistent *actors*[17] that simultaneously *watch* the constraint store upon changes and alternately *propagate* a portion of their information to the shared store (Figure 35).
Because of deviating from the opaque and mostly depth-first selection strategy of standard (C)LP, the programming languages that have been developed in this philosophy are attributed to 'Concurrent Constraint Programming' (CCP) [Sar93]. While propagation pretty much resembles the tell operation from CLP, CCP actors that watch upon the constraint store require an additional *ask* operation that 'blocks' until relevant changes have been made.
The introduction of ask ('freeze' in PROLOG-II, 'suspend' in ECLIPSE, 'demons' in CHIP, 'residuation' in LIFE) has a great impact on the underlying operational model

[17]Often, the term 'agent' is used instead of 'actor' in the CP literature which is not to confuse with our notion of intelligent agents.

$AskConstraint ::= ask\langle ComplexConstraint\rangle$
$Clause ::= \tilde{\forall} ProperLiteral \equiv \tilde{\exists} AskConstraint \wedge ComplexConstraint \wedge Goal$

Figure 36: Definite Concurrent Constraint Equivalence Clauses

and brings CCP very close to concurrent [Mil80], functional [Mil92], and distributed [Tel94] programming. Consequently, Saraswat [Sar93] and others [KTW94] pointed out that this relationship allows for a 'glass-box' approach to constraint solving in which actors must not be identified with built-in 'black-box' constraint solvers, but are rather modelled as procedural, i.e., clausal abstractions of atomic ask and tell operations. Based on a set of well-defined atomic operations upon a useful class of basic data structures given by the language designer, the application programmer can thus prototype more sophisticated, probably incomplete constraint procedures that are tailored to the problem domain.

In Figure 36, we have sketched a minimal CCP language whose equivalence clauses each incorporate a set of blocking *AskConstraint* to be asked, a set of *ComplexConstraint* to be told and a set of regular sub-procedures *Goal* to be called. We can easily extend SLC in order to handle this language: To resolve a goal, we impose the additional restriction that the intermediate constraint store Co_1 first has to contain (or disagree with) the asked constraints A. This is established by an embedded ask procedure $\vdash_{ask}$: $ComplexConstraint \times AskConstraint \leftrightarrow \{\top, \bot\}$ that decides about (dis-)entailment and that 'suspends' until $T : \{\top, \bot\}$ can be determined.

$$\frac{\begin{array}{ccc} Co_1 \wedge G_1 \wedge & \pi(P) \ni \tilde{\forall} C(Te'_1, \ldots, Te'_n) \equiv & Te_i\sigma = Te'_i\sigma \\ C(Te_1, \ldots, Te_n) & \tilde{\exists} A \wedge Co_2 \wedge G_2 & (Co_1\sigma, A\sigma) \vdash_{ask} T \\ & & Co_1\sigma \wedge Co_2\sigma \vdash_{tell} Co_3 \end{array}}{Co_3 \wedge G_2\sigma \wedge G_1\sigma \wedge T} \text{ SLCC}$$

Saraswat's complete cc(X) language [Sar93] analyses a gamut of additional control constructs, such as 'don't care commitment', 'eventual publication', and 'global ask', which all build onto the basic notions of ask and tell and relate to features of other concurrent logic programming languages, such as Concurrent Prolog [Sha87], Parlog [CG86], and ALPS [Mah87]. The richness of the resulting framework made constraint solving in, e.g., finite domains — cc(FD) — and floating-point arithmetics — cc(R)— feasible, at the same time allowed for new and elegant implementations of standard algorithms.

6.3 Constraint Programming: Reconciling CLP and CCP

As can be seen from the myriad of languages that have emerged, the discovery of the intermediate space between declarative and operational programming has been a deep and fruitful insight. It is thus no surprise that in recent years, even 'Constraint Programming' (CP) environments in the intersection of CLP and CCP have been pushed forward.

A special focus is placed on reconciling the rich actor model of CCP with 'speculative' computations from CLP that are introduced by optional resolvent clauses. Speculative

computation is an important means to deal with incomplete constraint solvers that get stuck on a problem when all actors are blocked by asking. Speculative computation plays a role when actors should be allowed to ask not only for basic constraints, but also for complex expressions. And speculative computation is an elegant way for describing conditionals, that are ubiquitous in functional and imperative programming, within the constraint-based ask & tell framework.

The Oz Model of Computation Spaces One of the first systems moving into this direction has been AKL [HJ90] which pretty much builds on Warren's abstract machine for Prolog [War83]. AKL's notion of *encapsulated search* has been overtaken to the design of Oz [Smo95] which is a uniform and modern basis for functional, concurrent, constraint, and object-oriented programming. Figure 37 introduces its kernel notation in accord with the already presented syntax. In the Oz model (Figure 38), the constraint store is generalised to a set of hierarchically arranged computation *spaces* each of which carries a set of actors called *threads*. Subordinate spaces carry out speculative computations on the information that is already present at or that is elaborated concurrently within their ancestor spaces.
For example, a conditional $cond\langle E_1, E_2, E_3\rangle$ spawns a subordinate computation space in which the asked condition E_1 — a constraint or an arbitrary expression called the *guard* — is elaborated within a new thread. The conditional will block until the subordinate space is entailed (all threads vanished and have not added any further information to the parent space) or disentailed (some inconsistency has occured). The conditional either reduces to E_2 in the case of entailment, whereby the computation space is *merged* with its parent, or to E_3 in the case of dis-entailment, whereby the computation space and its threads are deleted.
Procedures ($proc\langle V_1, V_2, \ldots, V_n, E\rangle$) are the Oz counterpart of clausal definitions. Just as threads ($thread\langle V_1, E\rangle$) and spaces ($space\langle V_1\rangle$), they are first-class citizens of the Oz universe that are assigned to logic variables (here: V_1) and that are manipulated in a higher-order fashion, such as by *applying* a procedure $apply\langle V_1, T_2, \ldots, T_n\rangle$, by suspending $suspend\langle V_1\rangle$ or resuming $resume\langle V_1\rangle$ a thread, and by inspecting $inspect\langle\rangle$ or influencing $inject\langle\rangle$, $clone\langle\rangle$, $merge\langle\rangle$, $commit\langle\rangle$ the status of a space.
This results in a very expressive language, at the same time provides useful control constructs for, e.g., programming search strategies in the language itself. This is because spaces can be used to trace the optional ways of elaborating an expression of the form $choice\langle E_1, E_2\rangle$. As controlled from, e.g., the top-level space, a stable space (all active threads are suspended) in which above disjunction appears can be *cloned* into

$Expression ::= BasicConstraint \mid Expression \wedge Expression \mid \tilde{\exists} Expression \mid$
$cond\langle Expression, Expression, Expression\rangle \mid choice\langle Expression, Expression\rangle \mid$
$proc\langle Variable^*, Expression\rangle \mid apply\langle Variable, Term^*\rangle \mid$
$thread\langle Variable, Expression\rangle \mid suspend\langle Variable\rangle \mid resume\langle Variable\rangle \mid$
$space\langle Variable\rangle \mid inspect\langle Variable, Term\rangle \mid inject\langle Variable, Variable\rangle \mid$
$clone\langle Variable, Variable\rangle \mid merge\langle Variable, Term\rangle \mid commit\langle Variable, Term\rangle \mid \ldots$

Figure 37: The Oz Kernel Language

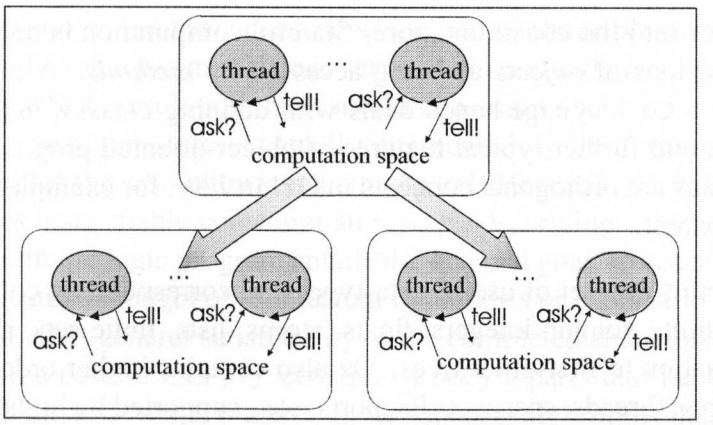

Figure 38: Computation Spaces in the Oz Model

a separate space that contains equivalent, but named apart information and copies of active threads. The disjunction is then *committed* to E_1 in the original in to E_2 in its clone. This way, a computation that is originally blocked, e.g., due to employing an incomplete constraint solver, is reactivated in both successor spaces. The successors collectively derive a complete set of solutions that are to be merged with the top-level space on success.
This is the so-called 'propagate-and-distribute' search technique that proved to be highly suitable for combinatorial problems: As much information as possible is added to the store by constraint solving before any expensive branching, i.e., cloning, of the search tree is performed. In a logic interpretation, 'propagate-and-distribute' is closely related to least-commitment selection.

State-of-the-art Constraint Programming In the current Mozart implementation of Oz, there are many additional constructs and operational variations to the above described mechanisms. The philosophy is that 'everyday' constructs are highly optimised in the kernel while convenient special-purpose expressions are added as syntactic sugar. We can only give an overview here and refer the interested reader to [Smo95, Hen97, Sch98, Meh99]. Oz is nowadays presented as the extension of concurrent functional programming with constraint techniques.

- Oz threads are lightweight. This is because of its computational model being significantly different to the one of imperative languages such as Java. It is possible to run up to 10.000 threads fairly scheduled on a single processor. At the same time, explicit threading gives the programmer control about concurrent composition and synchronisation of expressions. *Locks* preserve an user-level of consistency.

- Oz incorporates an *exception handling* mechanism by which a flexible recovery from, e.g., inconsistent tell operations, is reached. It is possible to raise user-defined exceptions as a convenient way of structuring computations.

- Besides logic variables which do not change their assigned value, Oz provides stateful data-structures, so-called *cells*, modelled as special actors with changing

references into the constraint store. Stateful computation is necessary to model the transitions of *objects* as being accessed by *methods*. A major part of the full-blown Oz language hence deals with defining *classes*, *inheritance*, *object creation*, and further typical features of object-oriented programming. Objects and threads are orthogonal concepts in Oz. In Java, for example, this is not quite as transparent.

- Oz has built-in a set of useful data-types and corresponding constraint systems, such as finite domain integers, floats, atoms, lists, finite sets, and records as a generalisation to first-order trees. Oz also contains higher-order types, such as procedures, threads, spaces, cells, ports, etc., supported by higher-order builtins, e.g., for inspecting the (partial) value of logic variables to effectively control search. There is a comprehensive C++-interface to extend Oz with user-defined types and constraints.

- Oz is one of the first *distributed* constraint languages that allows several virtual machines to construct a transparent and consistent constraint store. Programming abstractions to control the network operation are provided. As a by-product, Oz is able to export and import any type of stateless data (including procedures) to the network and the file system.

Mozart and its precursors Oz-1 and Oz-2 are among the most advanced interpreted programming languages. The successful applications realised so far cover as different areas as scheduling [SW98], time-tabling [HW96], natural language understanding, planning [JFB96a], and multi-agent systems [Mül96, FKM+95].

Oz for Implementing Agents Today, intelligent agents are proposed as a vehicle to describe coarse-grained constraint-solving facilities that are, e.g., distributed over the Internet. In the other direction, CP is viewed as the right tool to bridge theoretical and practical aspects of multi-agent systems. To quote Les Gasser:

> "There are currently only a few abstract models that can be useful for reasoning both theoretically and practically about distributed, multi-agent problem solving, and one of these is distributed constraint satisfaction."

The usability of Oz for implementing broad agents has already been recognised in [Mül96, Ros96] where object-orientation and concurrency were used to pragmatically implement the modular InteRRaP structure. Indeed, the COOP processes from Section 3 are quite close to lightweight actors or threads communicating over a shared constraint store (Figure 36). A not quite as obvious, but more sophisticated correspondence can be observed between abductive inferences and constraint-based computation when identifying the residue with the constraint store. For that purpose, compare the similar treatment of abducibles in IFF with the one of constraints in SLC.

6.4 On the Similarities of ALP and C(L)P

Several authors have commented on the similarities of ALP and CLP with respect to a unified reasoning framework (see [KKT98] for a comprehensive overview). Their

common observation is that basic constraints as well as abducibles are predicates that are undefined in the logic program, hence serve as not regularly resolved completions. If we look at how SLC resolution treats a told constraint such as an ordering $T_1 \dot{<} T_2$ we see that the constraint store is actually augmented by the formula, i.e., the constraint must hold for the resolution step being sound. Hence, a success node of a SLC derivation carries a satisfiable constraint store, e.g., describing a partial order, which in conjunction with the logic program entails the original goal. Consequently, the final constraint store can be thought of as a factual hypothesis or explanation for the goal.
In both CLP and ALP, general satisfiability of the completed abducibles/constraints is not enough. In the case of CLP(X), constraints obey a particular background theory X which is implemented by the plugged-in satisfaction algorithms that, e.g., compute the transitive closure and check the anti-symmetry of ordering constraints. This is an identical function to the integrity conditions *ECK*3 and *ECK*4 of our *HEC* theory that are placed onto the abducibles by ALP.
It is thus not surprising that these integrity conditions are called constraints in IFF and other abduction frameworks, such as ACLP [KM97b]: Their operation of 'asking' information from the 'store' of abducibles and 'telling' conclusions in turn is just a declarative version of what is happening during constraint propagation[18]. Hence, IFF constraints can be understood as an instance of the 'glass-box' approach to CP and the theoremhood of integrity can be seen as the theorem proving equivalent to satisfiability in the constraint universe.
As [KKT98] has remarked, CP goes even further than ALP by allowing to simplify constraints/abducibles in the store. Therefore, IFF should be reasonably implementable upon CP languages, such as Oz, by translating a logic program, such as *HEC*, into a corresponding constraint program. In [JFB96a], we presented a syntax translation scheme for particular classes of ALP programs. It maps, e.g., clausal definitions onto Oz procedures, conjunctions into composed expressions, existential quantification into lexical scoping, disjunctions into choices, equalities onto Oz-unification, and goals into either basic constraints or procedure applications. We shall not repeat this result which followed the abduction as deduction idea of [Fun96]. Rather, the current subsection concentrates on the improvements that we could reach for implementing *HEC* due to 'abduction as constraint solving'.

6.4.1 Implementing Time

One example for improvement is the realisation of the partial-order $\dot{<}$ by means of interval-based finite-domain constraints in Oz. Time constants can be mapped onto Oz integers, time variables onto finite-domain variables, and abducibles of the form $T_1 \dot{\leq} T_2$ onto the predefined Oz constraint $T_1 \dot{+} \dot{0} =< : T_2$. This provides us with a representation that is effective in avoiding to elaborate particular disjunctions — remember that $\dot{\leq}$ was just a short-cut for $\dot{<} \vee \doteq$ — and in using negative information constructively — $\neg T_1 \dot{\leq} T_2$ can be modelled as $T_2 \dot{+} \dot{1} =< : T_1$[19].

[18] Whether we define propagation as transferring information from the store to the constraint (IFF), as releasing information into the store (CLP), or as the combined activities (CP) is merely a question of terminology.

[19] In theory, this treatment presupposes the axiom $\tilde{\forall} T_1 \dot{<} T_2 \vee T_1 \doteq T_2 \vee T_1 \dot{>} T_2$, hence a total order of time points, but still allows for partially-ordered solutions with simultaneous or interleaved events.

This representation is open in that it can be synchronised to the actual system clock for specifying dead-lines and detect unreachable goals. The representation is expressive in that it allows to describe minimal and maximal durations of events by $T_1 \dot{+} d_{min} =<: T_2$ and $T_2 \dot{-} d_{max} =<: T_1$. And the representation is supported by higher-order built-ins which *reflect* the state of partially instantiated finite-domain variables, e.g., to steer search and to communicate temporal narratives.

However, there is a danger in all too naively using available 'black-box' constraint solvers, such as $=<:$, that are apparently suitable for particular applications, such as *HEC* planning, because one must always be aware of hidden operational considerations. Early experiments revealed spurious performance anomalies when facing temporal inconsistencies, such as by being told $T_1 \dot{<} T_2 \wedge T_2 \dot{<} T_1$. In the course of planning, the detection of such inconsistencies is crucial for resolving persistence threats by either demoting or promoting events.

The reason for the stated 'anomalies' is that the Oz constraint has been designed to provide local consistency in the first place, i.e., each single $T_1 \dot{+} \dot{1} =<: T_2$ expression corresponds to a single thread that is asking for information just about the related finite-domain variables (e.g., $T_1 \geq 10\dot{8}7$ and $T_2 \leq 12\dot{6}49716$). Its possibilities for telling (e.g., $T_1 \leq 12\dot{6}49715$ and $T_2 \geq 10\dot{8}8$) are similarly restricted. In order to arrange global consistency in the constraint store, some interleaved effort of several $=<:$ propagators is necessary: $T_1 \dot{<} T_2$ each time pushes the lower bound of T_2 and the upper bound of T_1 by one, hereby awaking the complementary propagator of $T_2 \dot{<} T_1$ which triggers the first one in turn, etc. If we choose our finite domain quite large, such as we need to represent a fine-grained time scale, the detection of a single inconsistency in a single computation space can consume up to several minutes of propagation!

A similar situation occurs in constraint-based scheduling when serialising tasks. For that purpose, [Wür98] proposed to use *global constraints* which are actors that are incrementally extendible by basic constraints. Global propagators collect all the constraint information in a specialised data structure within the constraint store, such as a lazy list or even a pointer into conventional memory. Global constraints are thus able to perform much stronger inferences than local ones. In turn, their propagation is also more complex such that a reasonable trade-off between expressivity and effectiveness has to be investigated in their design.

In Algorithm 1, we have sketched a global propagator for temporal constraints $X \dot{+} d \dot{\leq} Y$ that allows to quickly detect inconsistencies, at the same time upholds a tractable (quadratic) complexity. It maintains a matrix *dist* and two vectors *up*, *dn* storing the intermediate distance between finite-domain variables and the current interval bounds, respectively. The matrix is 'initialised' with $-\infty$ for distinct variables, otherwise with $\dot{0}$. Initial upper bounds *up* are set to ∞, lower ones to $-\infty$. The final implementation allocates matrix and vector entries on-demand.

As an invariant, the propagator installs distance transitivity $dist(V_1, V_2) + dist(V_2, V_3) \leq dist(V_1, V_3)$ and bound adequacy $dn(V_1) + dist(V_1, V_2) \leq dn(V_2)$; $up(V_1) \leq up(V_2) - dist(V_1, V_2)$. As a by-product, the $dist(V_1, V_1) = \dot{0}$ requirement also preserves anti-symmetry, hence the satisfiability of a presumed total order. Whenever a new basic constraint is told, the algorithm enters a nested loop ranging over the variables constrained by $V_1 \dot{+} d_1 \dot{\leq} X$ and those constrained by $Y \dot{+} d_2 \dot{\leq} V_2$ in order to combine the maximal distances, detect inconsistencies, and re-arrange interval bounds. For simplification purposes, our above presentation has omitted an ask & tell-interface

Algorithm 1 Global Propagator $\vdash_t(X \dot{+} d \dot{\leq} Y)$

Require: X, Y : *Variable*; $d : \mathbb{Z}$; $dist$: *Variable* $\times$ *Variable* $\rightarrow \mathbb{Z}^\infty$;
up, dn : *Variable* $\rightarrow \mathbb{Z}^\infty$

Ensure: $dist(X,Y) \geq d$; $dist(V_1,V_2)+dist(V_2,V_3) \leq dist(V_1,V_3)$; $dist(V_1,V_1) = 0$;
$dn(V_1) + dist(V_1,V_2) \leq dn(V_2)$; $up(V_1) \leq up(V_2) - dist(V_1,V_2)$

```
for all V1 : Variable and dist(V1, X) ≠ −∞ do
  for all V2 : Variable and dist(Y, V2) ≠ −∞ do
    M := max(dist(V1, V2), dist(V1, X) + d + dist(Y, V2))
    if V1 = V2 and M > 0 then
      ⊥
    else
      dist(V1, V2) := M; dn(V2) := max(dn(V2), dn(V1) + M)
      up(V1) := min(up(V1), up(V2) − M)
    end if
  end for
end for
```

to other constraints through the constraint store, such as the unification of variables triggering matrix and vector simplification and such as conditions of the form $dn(V) = up(V)$ invoking unification $V \doteq up(V)$. Similarly, interval bounds can be shared with other Oz-builtins, such as =<:.

For building propagators and associated portions of the constraint store in Oz, an extensive C++-interface [Mül99b] is available that includes hooks into unification and garbage collection, at the same time transparently caters for the management of computation spaces. Because of being a higher-order 'glass-box' language, Oz already provides a built-in support for stateful data, such as the *dist* matrix. Oz provides locking which supports a consistent interleaved access from the level of programmable threads. And Oz allows to reflect the status of bounds on finite domain variables. For prototyping purposes, we thus chose to implement the above propagator as a procedural abstraction in the Oz language itself.

6.4.2 General Abducibles, Negation, and Integrity

The treatment of temporal constraints demonstrates that the very principle behind implementing constructive ALP upon deductive LP is to reify as much as possible the relational information in the residue, e.g., $T_1 \dot{<} T_2$, by functional information in the 'constraint store', e.g., $T_1 \doteq 18\dot{4}93$.

This principle has been used in [Fun96, JFB96a] and similarly in the meta-programming approach of [Sha97a] to realise general abducibles, such as *happens*, by means of standard clausal or procedural definitions: Additional arguments H_1 and H_2 are introduced to represent the residue as a list of terms where H_1 refers to the state of the residue before entering the clause and where H_2 refers to its state after leaving the clause:

$$\tilde{\forall} happens(E, A, T_1, T_2, L, H_1, H_2) \equiv$$
$$\quad \tilde{\exists} member(happens(E, A, T_1, T_2, L), H_1) \wedge H_2 \doteq H_1$$
$$\quad \vee H_2 \doteq Cons(happens(E, A, T_1, T_2, L), H_2)$$

$$proc\langle Happens, E, A, T_1, T_2, L, H_1, H_2,$$
$$\quad choice\langle apply\langle Member, happens(E, A, T_1, T_2, L), H_1\rangle,$$
$$\quad\quad H_2 \doteq Cons(happens(E, A, T_1, T_2, L), H_2)\rangle\rangle$$

By routing the intermediate residues from the initial goals through any corresponding clausal definition of a definite program, we can emulate an abductive proof using ordinary SLD or the basic search machinery of Oz. For example, the *clipped* predicate and its corresponding Oz procedure *Clipped* can be implemented as follows:

$$(HEC2') \qquad \begin{array}{l} \tilde{\forall} clipped(F, T_1, T_2, C, B, E, L, H_1, H_2) \equiv \\ \quad \tilde{\exists} happens(E_t, A_t, T_3, T_4, L, H_1, H_3) \wedge E_t \dot{\neq} E \\ \quad\quad \wedge intersects(T_1, T_2, T_3, T_4) \wedge dual(F, F_-) \\ \quad\quad \wedge causes(A_t, F_-, T_3, T_4, cons(E, C), B, E_t, L, H_3, H_2) \end{array}$$

$$proc\langle Clipped, F, T_1, T_2, C, B, E, L, H_1, H_2,$$
$$\quad \tilde{\exists} apply\langle Happens, E_t, A_t, T_3, T_4, L, H_1, H_3\rangle \wedge cond\langle E_t \doteq E, \bot, \top\rangle$$
$$\quad \wedge T_1 \dot{\leq} T_4 \wedge T_3 \dot{\leq} T_2 \wedge apply\langle Dual, F, F_-\rangle$$
$$\quad \wedge apply\langle Causes, A_t, F_-, T_3, T_4, cons(E, C), B, E_t, L, H_3, H_2\rangle\rangle$$

This is realisable on any LP platform. However, apart from temporal constraints, there are some issues which prefer the C(L)P approach: Due to the routing of intermediate residues, we have implicitly sequentialised the sub-goals in the above *clipped* predicate. Having a computation rule that does not proceed in exactly that order, e.g., that tries to solve $causes(\ldots, H_3, H_2)$ before H_3 is bound by $happens(\ldots, H_1, H_3)$ will cause an explosion in the search space and even leads to non-termination. Since Oz elaborates conjuncts by a Prolog-like depth-first computation rule, such behaviour is prohibited. In its higher-order language, we can even program the search rule upon computation spaces such as an all-solution depth-first rule by the following *SearchAll* procedure:

$$proc\langle SearchAll, Spaces, Solutions,$$
$$\quad cond\langle \tilde{\exists} Spaces \doteq cons(Space, R_1),$$
$$\quad\quad inspect\langle Space, I\rangle \wedge cond\langle I \doteq succeeded,$$
$$\quad\quad\quad \tilde{\exists} merge\langle Space, Solution\rangle \wedge Solutions \doteq cons(Solution, R_2) \wedge$$
$$\quad\quad\quad\quad \wedge apply\langle SearchAll, R_1, R_2\rangle,$$
$$\quad\quad\quad cond\langle I \doteq failed, apply\langle SearchAll, R_1, Solutions\rangle,$$
$$\quad\quad\quad\quad \tilde{\exists} clone\langle Space, SpaceClone\rangle \wedge commit\langle Space, \dot{1}\rangle \wedge commit\langle Space, \dot{2}\rangle$$
$$\quad\quad\quad\quad\quad \wedge apply\langle SearchAll, cons(Space, cons(SpaceClone, R_1)), Solutions\rangle\rangle,$$
$$\quad\quad Solutions \doteq nil\rangle\rangle$$

However, *HEC* is not a definite program because of dealing with negated literals in its definitions and because of containing integrity constraints. In the above *Clipped* procedure, the implementation of inequalities ($\neg E_t \doteq E$ is replaced by $cond\langle E_t \doteq E, \bot, \top\rangle$)

and negative temporal constraints ($\neg T_1 \dot{\leq} T_2$ is replaced by $T_2 \dot{<} T_1$) presents a very restricted address of negation in the light of abduction.
In the LP context, this has been generally tackled by relying on the 'Negation-as-Failure' (NF) principle [JFB96a, Sha97a]. For example, the $\neg$*clipped* sub-goal of the *holds* predicate in *HEC*1 can be implemented as the failure of a purely deductive proof that operates on the 'frozen' residue H_2. This is equivalently expressible using encapsulated search in Oz: A nested search using *SearchAll* is conducted whose finite failure (the search procedure returns *nil*) is required to proceed with the computation. The safeness restrictions of SLDNF are implicit to the Oz model in which a subordinate computation space blocks if trying to strengthen information of one of its ancestors' variables.

$$
\begin{array}{ll}
 & \tilde{\forall} holds(F, T_1, T_2, C, B, E, L, H_1, H_2) \equiv \ldots \\
 & \quad \tilde{\exists} T_1 \dot{\leq} T_2 \wedge \neg member(E, C) \wedge happens(E_i, A_i, T_3, T_4, L, H_1, H_3) \\
(HEC1) & \quad \wedge T_4 \dot{<} T_1 \wedge causes(A_i, F, T_3, T_4, cons(E, C), B, E_i, L, H_3, H_2) \\
 & \quad \wedge flip(B, B_2) \wedge \neg clipped(F, T_3, T_2, C, B_2, E, L, H_2)
\end{array}
$$

$$
\begin{array}{l}
\tilde{\forall} proc\langle Holds, F, T_1, T_2, C, B, E, L, H_1, H_2, \ldots \\
\quad \tilde{\exists} \ldots \wedge space\langle S \rangle \wedge proc\langle P, R, apply\langle Clipped, F, T_3, T_2, C, B_2, E, L, H_2 \rangle \rangle \\
\quad\quad \wedge inject\langle S, P \rangle \wedge apply\langle SearchAll, cons(S, nil), nil \rangle \rangle
\end{array}
$$

The trouble in combining abduction with NF is that NF presumes a generally closed world, while abduction must stay open with respect to abducibles. In particular, the above scheme must assume that H_2 is the final residue of the proof. This is not true if we are to prove a further *holds* goal whose extensions of the residue (additional assumptions) could invalidate the already obtained negations. The way-out that is proposed in [JFB96a, Sha97a] is to collect the intermediate negated literals in a dedicated data structure. Until all goals have been resolved, these literals are frequently re-checked each time the residue is extended.
Doing the same finite-failure checks over and over again is a costly enterprise, at the same time not very constructive in its use of negative information. For this purpose, a more principled way of treating negations $\neg\tilde{\exists} Lit$ has been proposed in the IFF procedure by means of universally quantified integrity constraints $\tilde{\forall} Lit \supset \bot$ (see the SMP1 step). We have already demonstrated at hand of cases from *HEC* ($\neg E \doteq E_T$, $\neg T_1 \dot{\leq} T_2$, *ECK*3, and *ECK*4) that the interleaved operation of integrity constraints — asking information from the residue store and splitting new goals — can be reasonably emulated by CP languages, such as Oz.
The important representational change to make is the switch from a sequence of fully specified intermediate residues H_1, H_2 to a single partially specified data structure H, such as a lazy list. Due to the ask & tell framework, CP languages provide extensive support to controlledly access and enlarge such data structures without exploding search spaces. For example, the following *ForAll* procedure applies a given unary procedure to all the elements of a lazy list. It simply blocks as long as the substructure of the list is not determined.

$$
\begin{array}{l}
proc\langle ForAll, L, P, \\
\quad cond\langle \tilde{\exists} L \doteq cons(H, T), apply\langle P, H \rangle \wedge apply\langle ForAll, T, P \rangle, \top \rangle
\end{array}
$$

Due to their concurrent computational model, CP languages moreover enable an interleaved access to this common data structure from several threads of control, such as from the following Oz implementation of the 'integrity constraint' *NotClipped*. A combination of *SearchAll* and *ForAll* is used to imitate the propagation of abducibles into the constraint — P_1 is the procedure that looks up events from H that are not equal to E in a nested search — and the splitting of new goals from the constraint — P_2 either demonstrates the event not being a destroyer or pushes it out of the persistence interval. Hence, *NotClipped* installs itself as a concurrent thread that is constructively operating upon changes in the residue. This gives us a reasonable propagate & distribute scheme for general integrity constraints and negated literals in particular. In the vocabulary of partial-order planning, *NotClipped* implements a link between two actions that constantly shields persistence from possible destroyers.

$$
\begin{array}{l}
proc\langle NotClipped, F, T_1, T_2, C, B, E, L, H, \tilde{\exists} apply\langle Dual, F, F_- \rangle \\
\quad \wedge proc\langle P_1, R, apply\langle Happens, E_t, A_t, T_3, T_4, L, H\rangle \wedge cond\langle E_t \doteq E, \bot, \top\rangle \\
\qquad \wedge R \doteq happens(E_t, A_t, T_3, T_4)\rangle \\
\quad \wedge proc\langle P_2, happens(E_t, A_t, T_3, T_4), \\
\qquad choice\langle apply\langle NotCauses, A_t, F_-, T_3, T_4, cons(E, C), B, E_t, L, H\rangle\rangle, \\
\qquad\quad apply\langle Disjoint, T_1, T_2, T_3, T_4\rangle\rangle \\
\quad \wedge space\langle S\rangle \wedge inject\langle S, P_1\rangle \wedge thread\langle apply\langle SearchAll, S, L\rangle\rangle \\
\quad \wedge thread\langle apply\langle ForAll, L, P_2\rangle\rangle\rangle
\end{array}
$$

For special applications, such as *HEC* planning, this scheme can be further improved. Cloning of computation spaces, hence the elaboration of disjunctions, is a costly operation to be avoided as far as possible. In the above *NotClipped* procedure, we get rid of the nested *SearchAll* because its task of looking up elements from the lazy residue is equivalently performed by, e.g., a *Filter* procedure only relying on conditionals.
It is also wise to shrink the amount of propagated data, in this case the possible destroyer events, in order to reduce the overhead of distribution afterwards, here: the choices generated by applying *ForAll*. To this end, an additional *CouldCause* procedure is used in P_1 which has not the full functionality of *Causes*, but just asks whether the action potentially (without checking preconditions) has the particular effect. By singling out the non-relevant parts of a plan quite early, we gain a significant speed-up. The actual *NotClipped* implementation can also 'peek' how events behave with respect to the persistence interval by inspecting the already imposed temporal constraints. This way, some of the optional maintenance measures can be excluded ahead of time.
Finally, it is important to remove redundant symmetries in the search space. An example is the above choice in *NotClipped* which either tries to prevent the unwanted effect (although the event could happen outside the persistence interval) or pushes the event out of the persistence interval (although it could be a non-destroyer). Adding an additional *Causes* goal to the second choice makes both options incompatible, at the same time strengthens the information in the second option.

6.4.3 Representing Hierarchical Actions and Events

To remove further symmetries in *HEC* we switched to a more structured representation of abducibles. The main drawback of the flat list H is that it models the incremental construction of residues in ALP by the monotonic behaviour of unification in

LP. Any permutation of elements in H refers to the same set of abducibles and leads to redundant computations. Any access to H, especially by the integrity constraints that maintain the duration (*HEC*10), the identity (*HEC*11), and the (de-)composition (*HEC*6, *HEC*8) of events, has to completely map through the global data structure.
As seen in Subsection 6.4.1, a reasonable functional representation of an intermediate hypothesis Δ, such as a partial-order matrix, often looks non-monotonic (or stateful) from the Herbrand point of view. Secondly, the domain representation *DOM* containing action conditions and hierarchical relations is not conveniently described by converting clauses into ordinary Oz procedures. Abstract action types and concrete macro events have a far closer relationship in that the latter can be regarded as instances of the first.
The Oz object system [Hen97] is able to address these issues. Oz objects carry incrementally refineable logic information in terms of a *feature* structure. Oz objects carry dynamically changeable states in terms of an *attribute* structure. *Methods* are special-purpose procedures that are coupled to a respective object and, during whose application, features are accessed by means of unification and object attributes are accessed by means of read/write operations. Features, attributes, and methods are defined by an inheritance hierarchy of *classes* that objects instantiate upon creation. [Hen97] reduces these concepts to well-defined kernel primitives, such as logic variables, cells, and procedures.
It is quite natural to define the action types of the given domain as Oz classes that are instantiated by concrete event objects that build the residue of a constraint-based *HEC* proof. Domain representations, i.e., pre- and postconditions of an action and its hierarchical decompositions, can be encoded into the features of action classes and are able to refer to other (sub-) classes of actions for that purpose. The derived event objects encode references to sub-events and temporal 'distance' relationships between them in their attribute structure.
As a result, the previously flat residue is now modelled as a hierarchical data structure that behaves monotonic where it is adequate and stateful and stateful where it is more efficient. The interface between *HEC* and this data structure is built by predefined methods of the super-class `BaseEvent` from which every action and every event is derived and that is sketched in Figure 39 (now using proper Oz syntax). Its methods are called from the *HEC* procedures and provide recursive access to features and attributes of the possibly complex macro-event, such as its temporal information (`start`, `duration`, `fin`, `times`, `dist`), its conditions (`prec`, `fx`), and its substructure (`options`, `events`).
The `initialise()` method is called upon creation of a each new object of type `BaseEvent`. `initialise()` serves to set the object-specific `events` attribute, i.e., the extendible record of sub-events according to the class-defined `options` feature which carries the decomposition structure of the associated domain action. `initialise()` recursively instantiates the sub-events of a macro, hereby implementing *HEC*6 and *HEC*8. The identity of Oz objects automatically realises the *HEC*11 constraint. Finally, `initialise()` sets temporal information `start`, `duration`, `fin`, `times`, and `dist` according to *HEC*10, *ECK*3, and *ECK*4.
Action and event references in `options` and `events` are organised such that symmetries are avoided. We have extended this representation to even reify the disjunctive decomposition of a macro within a single data structure. By that design and corre-

```
class BaseEvent
      feat                          %% the 'logic' features:
         start duration fin              %% fd timepoints and interval
         parameters                                %% parameters/roles
         options                             %% decomposition structure
         fx prec                       %% effects and their conditions
         initiators                          %% precondition initiators
      attr                          %% the 'stateful' attributes:
         events                                              %% sub-events
         times                                    %% times of sub-events
         dist                                %% temp. precendence matrix
         cost                                   %% current cost of macro
                                  %% the methods:
      meth initialise() end                        %% initialisation method
      meth removeBranch(...) end                   %% delete decomp. option
      meth chooseBranch(...) end                   %% commit decomp. option
      meth happens(...) end                           %% look up sub-events
      meth addEvent(...) end                       %% add some new subevent
      meth isBefore(...) end                  %% look up temp. constraints
      meth before(...) end                         %% add temp. constraint
      meth couldCause(...) end                       %% look up uncond. fx
      meth causes(...) end                             %% look up cond. fx
      meth execute(...) end                        %% decompose and execute
      meth setTime(...) end                  %% instantiate time variables
      meth openDisplay() end                               %% display content
      ...
end
```

Figure 39: The `BaseEvent` Class (Sketch)

sponding interface methods to *HEC* (`removeBranch()`, `chooseBranch()`), the distribution effort of search can be diminished.

The `happens()` and `addEvent()` methods read and extent the `events` attribute according to the two modes of accessing the residue in our previous definition of the abducible *happens*. Similarly, `isBefore()` and `before()` implement the check of entailment and the telling of a new temporal constraint upon `dist` and `times` according to Algorithm 1. `couldCause()` and `causes()` access the effects and conditions of the event and its substructures. `execute()` is used in plan execution for (recursively) synchronising with particular external events and instantiating time points with the current system time via `setTime()`.

Oz classes allow locally-scoped variables in features and attributes, e.g., to implement `parameters`, such as the *Robot* and *Box* roles of a *pickup*(*Robot*, *Box*) action. These variables are not created until the creation of a new object and have been used to introduce additional references (`initiators`) to the initiating events of particular preconditions. In *HEC*, the relationship between producer and consumer of a fluent is not explicitely encoded. A naive inference strategy hence repeats the effort to find possible precondition initiators, every time that the success of a particular effect is revisited. Due to the theory, however, a producer-consumer connection, once established, remains valid forever, thus can be used as a lemma in the further proof. By establishing references in `initiators`, repeated `causes()` calls immediately succeed without

any search effort.

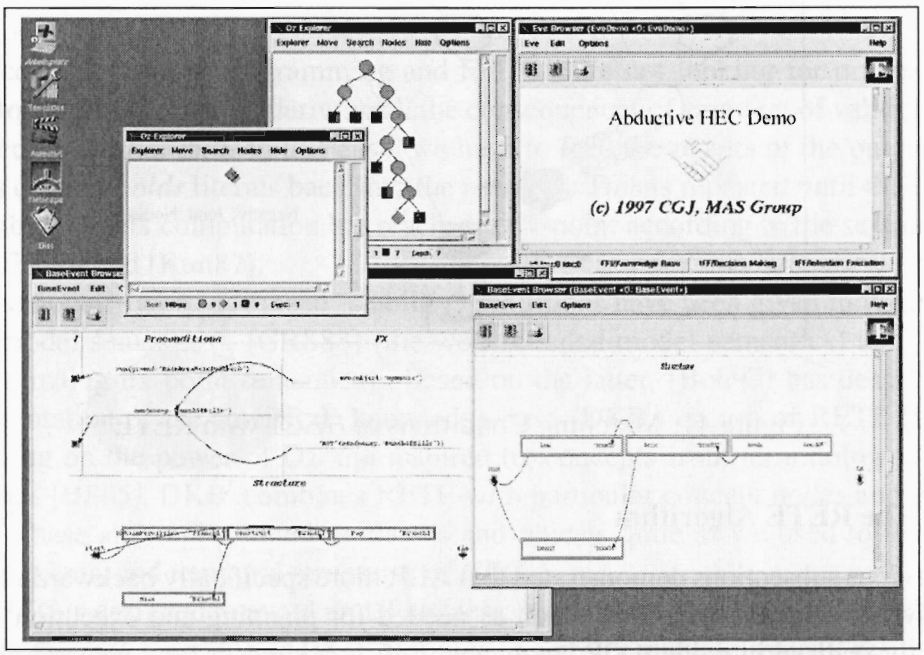

Figure 40: The Implemented Planning Process (Screen-shot)

In combination with the previously discussed improvements, this extension unleashes the possibilities of ask & tell as well as propagate & distribute leading to significant speed-ups for larger planning problems. In Figure 40, we have depicted a screen-shot of our *HEC* implementation that has planned the house-building domain of [RN95]. We recognise that the search for refining the top-level solution into eight complexly arranged actions on the second level of abstraction has collapsed into a purely deterministic(!) problem that is solved within 0.18 sec. Figure 40 also demonstrates the intuitive graphical user interface that is attached to `BaseEvent` by the `openDisplay()` method. `openDisplay()` accesses the data encapsulated into the event object to visualise its conditions, its hierarchical decomposition, and its references to other events.

Similarly, the computation of residue costs, `cost`, can be delegated to the event representation itself where all the relevant data is stored and updated. In ordinary LP, such a measurement would have to be syntactically reflected by the proof procedure in order to steer the selection of disjuncts. In higher-order Oz, the selection can be influenced from within the computation space, because choices can be generated on the fly and according to the different costs of options that they enumerate. By this design, it is also possible to specify a-priori bounds on the costs for a top-level plan to implement the iterative deepening. We have omitted the respective methods in `BaseEvent` for reasons of simplicity.

To render the initial situation I abducible, too, it is given a stateful representation similar to the one of narratives. It also possesses an interface for access and extension that is called from *HEC* and that uses information about predicate names, arity, and levels of abstraction for speed-up.

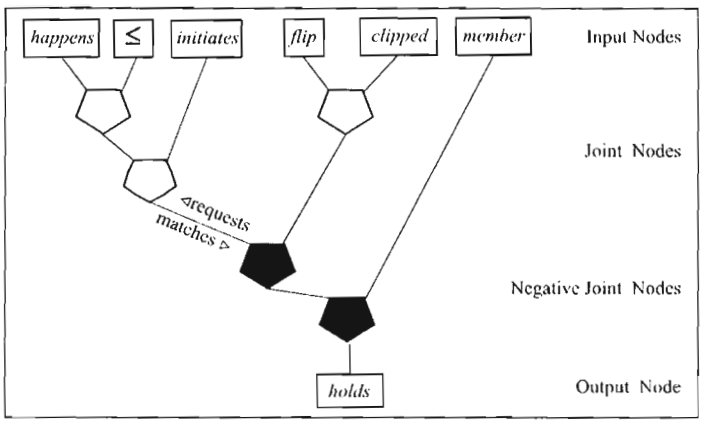

Figure 41: Matching Conditions of *HEC*1 with RETE

6.4.4 The RETE Algorithm

The previous subsections demonstrated that ALP, more specifically backward-chaining abductive inferences under *HEC* such as needed for planning and execution, can be efficiently realised in modern CP languages. Left to address is how forward-chaining prediction in terms of the FFI proof procedure can be programmed. As we have already discussed, this especially requires an effective algorithm for matching the complex conditions in the bodies of definitions and of integrity constraints. FFI's folding, for example, should avoid recomputing already derived information, e.g., literals that have already been demonstrated valid and sub-conditions that have already been matched.
In [For82], an algorithm is given that recognises a set of valid conditions, i.e., formulae composed of literals, conjunctions, and negations from a set of given facts. The important property of this algorithm is that it maintains and updates matched sub-conditions in an incremental data structure: a network of so-called matching nodes (Figure 41 shows the network for one particular condition of *HEC*1). Input nodes, such as those at the top of Figure 41, operate as filters upon the given facts by extracting instantiations of particular literals. Joint nodes, such as those in the middle of Figure 41, represent conjunctions that combine the instantiations from their predecessor nodes, e.g., by finding corresponding variable assignments. Special negative joint nodes test non-validity by only passing those instantiations of one predecessor that do not coincide with the results of the other. Finally, output nodes, such as the one at the bottom of Figure 41, produce the resulting instantiations that match the encoded condition.
Because nodes store intermediate instantiations, only the deltas of a changing set of valid facts have to be routed through and recomputed in the network to update the conditions. If the network is partitioned appropriately, small changes in the facts only amount to small changes in the network, hence computation time is optimally traded off against memory. Because of this data structure, the algorithm is called RETE (lat.: network). An Oz implementation of RETE that uses constraints to compute instantiations and object-oriented programming to implement the network structure has been developed in [Leh96].
RETE is the basis of many rule-based languages, such as the production system OPS-5 [KR87], the SOAR language [New90], the ACT-R architecture [And93], and the agent system MAGSY [Fis93a]. In these systems, the output of RETE is further condensed

by means of *conflict resolution* that triggers a unique external action. The result of externally acting is again fed into the RETE network to perform the next cycle of operation.
In the context of logic programming and FFI, we are not looking for one particular action to take, but rather for deriving all the consequences of some set of valid facts, the so-called *inferential hull.* In this case, we have to feed the results of the output nodes such as derived *holds* literals back into the network. This is repeated until the network gets stable, i.e., its computation has reached a fix-point according to the semantics of, e.g., [vEK76] and [Kun87].
Computationally more advanced 'stability' conditions have been given in [GL88] (the stable model semantics), [GRS88] (the well-founded model semantics) and [Van89] (the alternating fix-point semantics). Based on the latter, [Boh97] has described the implementation of the InteRRaP knowledge base (DKB^I) on top of RETE [Leh96]. By relying on the power of Oz and inspired by concepts from terminological representations [BS85], DKB^I combines RETE with particular concept nodes and instance nodes. These are implemented as classes and objects quite as we used to realise the actions, events, and temporal structures of *HEC*.
Since *HEC* is designed to avoid undefinedness, its execution under the more advanced forward-chaining principle of DKB^I amounts to a pretty much 'classical' result. Some additional considerations are required to derive the final implementation of forward-chaining *HEC*: First, there is the necessity to restrict the (recursive) computation of *member* literals to a reasonable, finite subset, i.e., to lists with at most one cycle and only containing events that have been already introduced via *happens*. Secondly, events and temporal relations between time points are not static as in typical monotonic settings, but extendible. This is because we are ideally looking for an abductive proof procedure that is able to additionally generate and maintain several optional hypotheses for given observations on the fly.
The idea is to extend RETE's flow of control into a lazy bidirectional scheme (Figure 41): Whenever a particular goal, such as a *holds* literal or a *member* literal, is hypothesised to be (in-)valid (it appears on the left-hand side of a range-restricted clause in FFI) its corresponding node in the RETE network generates appropriate explanations in the form of instantiation requests to its predecessors.
Optional requests are elaborated by the predecessors within separately cloned computation spaces where they are checked for consistency and where they invoke further requests in turn. This finally leads to newly instantiated abducibles within the input nodes of the network (*happens*, $\leq$, *member*) which are connected to the top-level plan object instance via method calls. Integrity constraints are implemented in the same way: Their conditions are constantly watched by a dedicated part of the RETE network. Upon instantiation, the conclusions of the constraint lead to new requests for the relevant goal nodes.
This support of abduction is still at a preliminary stage. In our testbeds, plan recognition plays indeed not a major role due to too restricted sensing facilities for creating necessary observations. Nevertheless, the further development of FFI and its implementation is an ongoing topic of our research that might turn out fruitful in application domains with a different emphasis.

```
class Process
   feat layers domain  %% id of control procs & domain action classes
   attr state active                              %% state of the search
        resourceValue performance            %% local resources, profile
        signals_in                                %% incom. signal queue
   meth initialise(layer:Layers domain:Domain ...) end     %% creation
   meth rollUp(initial:I plan:P goals:G                 %% insert fresh
               bound:B) end                          %% computation space
   meth search(...) end                                  %% search method
   meth pushException(...) end                        %% manipulate current
   meth popException(...) end                           %% exception stack
   meth sendSignal(recipient:R initial:I              %% transmit signal
                   plan:P goals:G ...) end    %% over control process
   meth rcvSignal(sender:S resources:R        %% incorporate resources
                  initial:I plan:P goal:G ...) end      %% and signal
   meth reportFailure(recipient:R ...) end          %% uncaught failure
   meth openDisplay() end                                  %% GUI stuff
   ...
end
```

Figure 42: The `Process` Class (Sketch)

6.5 Implementing Inference Processes

A central role in building processes based on the inferences presented in the preceding subsection is played by dedicated higher-order search procedures, such as the previously described *SearchAll*. Search procedures initialise inference problems in subordinate computation spaces. They implement search rules by inspecting and cloning spaces. And they extract solutions by merging spaces upon success. According to the COOP computational model, InteRRaP-R hence consists of the concurrent interplay of several search procedures running in separate threads (as in Figure 38), interacting with each other over the shared memory of the constraint store, and being steered by additional control processes. In the current subsection, we want to take a closer look at how inference processes are implemented as search procedures. Afterwards, we turn to the implementation of control processes, i.e., the InteRRaP-R layers.

The depth-first *SearchAll* procedure gives us a first starting point, since we have already pointed out the importance of persistent search within the agent context. Moreover, most of the residue-cost related selection strategy is already realised by the Oz implementation of ALP & *HEC*. However, *SearchAll* has to be extended, because we require an iterative deepening procedure that is able to switch between planning from scratch and planning from second principles. Moreover, we are not interested in generating all concrete solutions at once, but in continuously refining and outputting parts of the currently optimal solution. Finally, a search procedure to implement COOP processes should be able to interact with other processes and the environment at any time by possibly revising a former partial result in the light of incremental evidence.

For this purpose, the `Process` class (Figure 42) has been developed from which, e.g., knowledge base processes, decision making processes, and intention execution processes are derived. The `Process` class implements the *Process* schema of Section 3 by means of object features and attributes. The `Process` class realises the concrete

Δ*Process* operations in the overall *COOP* specification by means of appropriate methods,

The `initialise()` method is called upon creation of a process in order to register the superior control processes (the `layers` feature), to register the domain representation (the `domain` feature), and to initialise an 'empty' inferential `state` according to the *InitProcess* schema. `initialise()` makes use of the `rollUp()` method that is a general means for 'restarting' (a part of) the `state` attribute by inserting a new computation space using *space*$\langle\rangle$. Using *inject*$\langle\rangle$, `rollUp()` provides the new computation space with a 'logic program' consisting of a start situation `I`, an initial narrative object `P`, the set of available actions in `domain`, and the current cost-bound `B`.

'Logic goals' `G` that drive the computation inside the computation spaces are applications to the globally defined *HEC* procedures, such as the previously discussed *Holds*. The recursive `search()` method incrementally injects these goals via *inject*$\langle\rangle$ according to their level of abstraction into the computation spaces within `state`. It constructs the resulting search space by *clone*$\langle\rangle$ and *commit*$\langle\rangle$ quite similar to the depth-first *SearchAll* procedure. `search()` is activated within a dedicated thread due to changes in `state` caused by, e.g., commitments, roll-ups or incoming signals (see below).

Upon the intermediate success of a selected computation space, i.e., goals of a particular level of abstraction have been solved, `search()` merges the relevant parts of its encapsulated constraint store and signals them to the outside. For that purpose, the `sendSignal()` method transmits (via the control processes referenced in `layers` — see below) intermediate abducibles about initial situation `I` and narrative `P` and intermediate goals `G` to a recipient process.

Search is continued down to the most primitive (or up to the highest one in case of the knowledge base process) level of abstraction in which case the process suspends and the `active` flag goes down. `search()` also suspends in the case that the amount of local abstract resources — `resourceValue` is frequently accessed by `search()` — becomes too small. The amount of consumption is domain-specific and depends on the computed partial solutions. Similarly, the average `performance` profile is updated by measuring the improvement of the partial solution against the elapsed time. Failed computation spaces in `state` dissappear during `search()`, unless they are shielded by a non-empty exception stack. In this case, they can be recovered by calling the `rollUp()` method. To this end, the exception stack that is manipulated by `pushException()` and `popException()` carries alternative situations `I`, narratives `P`, goals `G`, and bounds `B`. The iterative deepening of bounds can be elegantly implemented in this manner.

Concurrently to `search()`, incoming signals are invoking the `rcvSignal()` method, hereby possibly refreshing the amount of abstract resources. For this purpose, `rcvSignal()` shortly locks the required data structures in order to not interfere with `search()`. In particular, `rcvSignal()` accesses `state` to incorporate the signal content into the search space. It uses *inject*$\langle\rangle$ to incrementally feed additional procedure applications into the computation spaces. These procedures either immediately influence the situation (`I`) and event (`P`) representations in the encapsulated constraint store in the case of *InformationSignal*. They represent additional *HEC* goals (`G`) to be pursued in the case of *RequestSignal*. `rcvSignal()` also stores the identity of the in-

coming signals as annotations to `state` such that `search()` is able to report failures back to the sending process via the `reportFailure()` method.
For particular domains, the prediction of the knowledge base requires no hypothetical reasoning. Also the decision making of the reflex process could lack any choices to make. Furthermore, the decomposition of intention execution processes could have no options to pursue. In these cases, the `Process` class can be further customised, since not the full power of speculative computations by encapsulated search is needed. Instead, the inferences can be immediately run, possibly by relying on conditionals $cond\langle\rangle$, within the top-level computation space. This greatly simplifies the computational needs of `rollUp()`, `search()`, and `rcvSignal()`. Especially for the time-critical computations in the BBL of forklift and RoboCup agents, this turned out to be a useful variation.

6.6 Implementing Control Processes

For transmitting signals and computational activity from a sender to a recipient `Process`, `sendSignal()` and `rcvSignal()` are not immediately coupled. Instead, they are connected over `Layer` objects (Figure 43) in charge of routing signals and abstract resources dedicated to the processes at a particular layer of InteRRaP-R. `Layer` installs the control of communication and hence of computation by implementing the *Layer* and Δ*Layer* schemas.
Within the `InteRRaP-R`class (Figure 43), three layers are combined with an environmental interface to the top-level schemas $InteRRaP - R$ and $\Delta InteRRaP - R$. To create an agent, an object of type `InteRRaP-R` is instantiated and its `initialise()` method is called. In this method, the three layers `bbl`, `lpl`, `spl` are created as particular instances of `Layer` and are equipped with references to each other. In a dedicated thread, `InteRRaP-R` then enters a loop by means of the recursive `perceive()` method that frequently checks for sensor input which it signals as *PerceptionSignal* to subscribed processes in the `bbl` (in case of low-level sensing) and the `spl` (in case of incoming communication) using the `sendSignal()` method. The perception thread will persist until the agent closes its operation.
`perceive()` allows for both passive and active sensing by switching input channels (e.g., serial port, TCP/IP streams, UDP sockets) and process subscriptions on the fly. Its current configuration is looked up in the `activeChannels` attribute and is accessed concurrently by `rcvSignal()`. An *ActionSignal* that has been generated inside `bbl` or `spl` invokes the `rcvSignal()` method. If its content is a sensing action, `rcvSignal()` locks and changes the `activeChannels` attribute. If its content is a proper action, such as a motor command, `rcvSignal()` invokes the `act()` method in turn.
It should be noted that particular data structures which are a part of the formal specification, such as *actionBuffer* in $InteRRaP - R$, are not explicitely present as features or attributes of the implemented objects, such as `InteRRaP-R`. They are rather implicit to the employed Oz programming constructs, for example, in the synchronisation of several method calls to `act()` that are suspended upon a shared lock.
This holds similarly for the *input* and *output* buffers of the *Layer* schema that are a by-product of scheduling the `signal()` method from `Layer`. `signal()` method is called from a sender process' `sendSignal()` method and updates the internal

```
class Layer
  feat bbl lpl spl                                  %% references to layers
       kb_process dm_process          %% kb and decision making processes
       domain costs                     %% domain and resource cost spec
  attr ie_processes                      %% intention executon processes
       signalBuffer triggerStore                        %% signal buffers
       resourceValue processUtility    %% global resources and profile
       processDiscount                          %% discount for process
       allocations discountFactor                  %% parameters for
       processEmphasis                         %% allocation algorithm
  meth initialise(layers:Layers ...) end             %% initialise agent
  meth signal(sender:S recipient:R ...) end     %% route/buffer signal
  meth allocate(...) end                       %% the allocation process
  meth openDisplay() end                                  %% GUI stuff
  ...
end

class InteRRaP-R
  feat bbl lpl spl                                          %% the layers
  attr activeChannels                      %% active perception channels
  meth initialise() end                                %% initialise agent
  meth perceive() end                                  %% perception cycle
  meth sendSignal(recipient:R ...) end      %% send perception signals
  meth rcvSignal(sender:S ...) end   %% receive action/sensing signal
  meth act(...) end                                     %% trigger action
  meth openDisplay() end                                  %% GUI stuff
  ...
end
```

Figure 43: The `Layer` and `InteRRaP-R` Classes (Sketch)

`triggerStore` and `signalBuffer` attributes. In the case that these attributes are non-empty, the layer's `allocate()` method is invoked within a separate thread. `allocate()` implements the filtering of signals and the allocation of abstract resources along the *ΔLayerControl* schema making use of the resource cost specification in the `costs` feature. `allocate()` invokes the `rcvSignal()` method of those processes that are granted signals and resources. `allocate()` is able to create new intention execution processes that are not yet referenced in `ie_processes` and equips them with the layer-specific `domain` representation.
The nonvolatile knowledge base and decision making processes are initiated upon creation of each `Layer` object (in the `initialise()` method) and are referenced in dedicated features `kb_process` and `dm_process`. `allocate()` makes use of special-purpose interfaces which we have omitted in Figure 42 and which allow to read and write information, such as performance profiles and resource parameterisations, immediately from and into subordinate processes' state.

6.7 Compliance of Specification and Implementation

We have now completed the overview of the Oz implementation of InteRRaP-R which showed that constraint-based programming techniques are suitable to efficiently realise

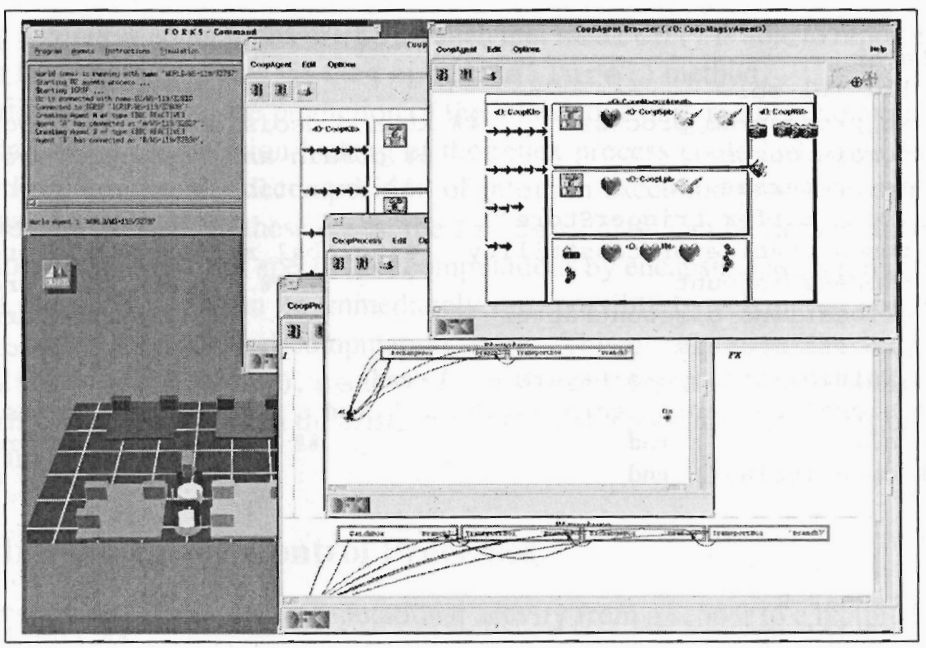

Figure 44: Implemented InteRRaP-R Agents (Screen-shot)

agent-based systems. A screen-shot of it can be found in Figure 44. In the following, we discuss a few particularities with respect to its compliance with the abstract model.

CET **versus** *FT* As a successor to CLP(X), Oz does not confine to a Herbrand universe in which the main data structures are finite trees and in which equality is axiomatised by *CET*. As other constraint-based languages and computational formalisms, the Oz universe has adopted convenient record-based data structures of imperative languages.

In a logic setting, records or henceforth called *feature structures*, such as $root(entry1 : branch(Var) \quad entry2 : leaf)$, provide a similar recursive construction to finite Herbrand trees. In addition, they allow to label the substructures with some atomic type (the feature, e.g., the constant $entry1$). For organising complex structures in computer memory, features provide significant advantages. Similar to the usage of pointers in imperative languages, feature constraint languages, such as Oz, can handle cyclic records such as $Self \doteq test(ref : Self)$.

For this purpose, the Oz unification algorithm omits the traditional 'occurs check' of *CET*4 and is extended to terminate on cycles and match to the features, too. To this end, it builds on the background theory *FT* that has been given in [BS95]. Due to its advanced CP model, Oz is able to unify feature structures that are only partially specified by so-called *open feature constraints* [Meh99].

Ordinary Herbrand terms, now called 'tuples', are still expressible as particular Oz records labelled with integers denoting the order of branches. This means that while resigning to finite data structures, the use of the Oz unification does not do any harm to the inferential semantics, i.e., *CET* appears as a restriction to *RAT* [Mah88] (the axiomatisation of rational trees) which is in turn a specialisation of *FT*. That cyclic references are useful for our purposes becomes apparent when reifying the recursive

decomposition of actions as explained in Section 4. Whether this recommends to base the semantics of *HEC* on *RAT* or even *FT* is a question of ongoing research.

Shared Memory versus Encapsulated Search The Oz model of computation spaces facilitates the implementation of a shared memory model between several inference processes, i.e., search threads. To keep a declarative reading of the interaction between computation spaces at different levels of the constraint store hierarchy, Oz has to apply *encapsulation* restrictions. These restrictions are currently not present in the COOP model.
For example, although a subordinate space is able to access information by references into the top-level store, it is not able to increase the information there, e.g., by instantiating a logic variable. Vice versa, a top-level space is not allowed to obtain any reference into an encapsulated space because this information represents purely speculative data that needs to be first confirmed and merged.
To install a bidirectional transfer between two computation spaces, such as between the states of two inference processes in the COOP model, some additional mechanisms had to be incorporated into `Process` which go beyond the already described `reportFailure()` method.
One of these mechanism is integrated into the injection of signals from top-level to encapsulated space. Instead of immediately injecting the possibly partially specified top-level data structures `I`, `P`, and `G`, the implemented COOP processes use a special-purpose 'cloning' procedure (in fact implemented by predefined methods of, e.g., `BaseEvent`) that reconstructs these data structures locally within the subordinate spaces.
`BaseEvent` does also have a special-purpose 'merging' method. It enables `Process` to recombine the intermediately obtained search results with the original top-level data in order to report the results to other processes for incorporation. Hence, the shared memory model of COOP (including the indication of failures) is reasonably restricted to the generation of partial results inside the inference processes.
It has turned out that the implementation does not make full use of the combined `I`, `P`, and `G` channels in a bidirectional manner. Instead, processes rather interact in a unidirectional service-oriented manner over particular of these data structures, e.g., the knowledge base 'hands out' a goal to the planner and is sent a plan in a separate signal in turn. We are currently investigating whether the computational model should be more specialised and closer to the implementation model in that respect.

Custom Exception Handling Oz-builtin exceptions follow the convenient try & catch semantics of modern multi-threaded programming languages. As such, Oz-builtin exceptions are raised and handled locally to a specific thread in a specific computation space. This required a custom facility in the `search()`, `pushException()` and `popException()` methods of `Process`.
Related to the above treatment of encapsulation, for example, Oz exceptions that are raised inside a space cannot be distinguished from the top-level. Furthermore, try & catch is very much tailored to capturing the conflicts that have been caused within a single thread of computation, such as a division by zero. It is not so much about reacting to external signals which have been concurrently injected and whose processing

must be temporarily protected against further interventions.
The implemented COOP exception handling in `search()` addresses these aspects where the failure of a subordinate computation space is just one of its conditions. Apart from that, it provides means of testing and reacting to inconsistencies within the `I`, `P`, and `G` annotations of `state` already on the top-level, such as for handling a passed deadline and inconsistent goals. To this end, the implementation specialises the purely stack-based COOP model into a more convenient stack- and pattern-based approach that is able to distinguish different sources of and reactions to inconsistencies. This should be reflected in future revisions of the computational model.

6.8 Bottom Line

This section described an implementation of the computational and inferential model of InteRRaP-R that we have developed in previous sections. We motivated and demonstrated that Constraint Programming [MR95, Fre96, Smo97, MP98] is able to realise both the declarative and the operational side of agents in unified implementation environments. The Oz language [Smo95] is an implementation platform that builds upon traditions from CLP(X) [JL87] and cc(X) [Sar93] and that enhances them with features, such as advanced constraints, stateful objects, and distribution.
We showed how abductive inferences upon *HEC*, including negation and integrity constraints, can be soundly and efficiently implemented using the Oz model of encapsulated search. On top of that, we showed how to build interactive inference processes as concurrent higher-order search procedures. These search processes are interconnected over the constraint store and special control objects to realise the layered structure of InteRRaP-R. The practicability of this implementation has been tested in three case studies which are documented in the following.

7 Three Case Studies

(Distributed) Artifical Intelligence derives its inspiration from engineering, natural, and cognitive sciences. Hence, its assessment can take place in many ways ranging from formal criteria in theoretical AI, over analytical criteria in architectural AI, up to empirical criteria in applied and cognitive AI.
This thesis has a design methodology at its core that aims at hybrid agents, i.e., rather domain-independent systems with several (sub-) cognitive functionalities. The design methodology incorporates several stages of research, such as architecture, theory, and implementation, in order to describe coherent models that bridge theory and practice.
Because this type of integrative research is relatively new, there is no agreed scheme for its evaluation. Hence, we propose to evaluate our methodological contribution in terms of the InteRRaP-R model which has been reconstructed according to that design stance and whose theoretical and analytical properties we have already extensively addressed in the previous sections. What remains to be documented is that this has indeed lead to an implementation that runs efficiently in realistic application settings.
To this end, we have chosen three prototypical scenarios, the Automated Loading Dock, the RoboCup simulation league, and the ROTEX space-robot, which already served to motivate the generic InteRRaP-R design. In the following, we describe the

efforts to customise the domain-independent agent core by respective domain representations. We give a comparison in adequacy and performance to the original forklifts of [Mül96], to other teams participating RoboCup'98, and to modified tele-robotics agents using a graph-based, algorithmic planning module.

7.1 Automated Transportation: The Loading Dock

In [Mül96], a real-world application for broad agents is presented: Today's industrial production such as the manufacturing of cars heavily relies on automated transportation facilities. The employed artifacts can be up to 2 meters in length and 1.5 meters in height. They are equipped with modern actuators and sensors, such as differential propulsion, flexible grippers, transponder detection, laser-scanners, ultrasonic sensors, and bumpers. Typically, up to 40 of these robots are serving a single production line. In spite of up-to-date hardware, their control software is extremely conservative: For example, there are hardly autonomous functionalities on-board besides low-level safety compliance. Instead, the navigation and tasking of the artifacts is computed centrally by a single optimisation process. Vehicle status and central decisions are exchanged between artifacts and optimisation process over radio-based network connections.
This design has disadvantages with respect to dynamics, robustness, flexibility, and scalability: The time that it takes to locally recognise a change of situation, to notify the central optimisation process, to integrate the reports of the dispersed artifacts into a decision, and to download these decisions on the local platforms, prohibits a suitable reaction to, e.g., human interception. The more transportation artifacts are to be controlled by the central optimisation, the longer one such cycle of operation will take. In the worst case, it will grow exponentially. Moreover, shortages and inaccuracies of the robot hardware and the unreliability of the communication network raise to a high power within the optimisation process.
For this purpose, the transportation systems are assigned exclusive and fixed navigation paths by the central computer, such that encounters with humans and other artifacts are avoided. Hence, changing the production line, e.g., if a new car series is introduced and if more transportation systems are employed, enforces a global redesign. In the case of failure, the complete production activity is halted. This is an inflexible and costly solution.

7.1.1 Scenario

An alternative design implements the decision making process at the local level of sensors and actors and thus gets rid of most of the unfortunate central processing duties. To analyse the requirements of this approach and to demonstrate its feasibility, [Mül96] has compiled the specification of the Automated Loading Dock scenario (Figure 45). In the loading dock, transportation artifacts are represented by forklift robots. The forklifts serve a set of rectangular shelves and a truck that is located in a loading ramp. The forklifts are able to unload boxes from the ramp and to store them on the shelves or vice versa by carrying one box at a time. The colours of boxes and shelves indicate different categories of goods to be stored and shipped.
Because of its sophistication that exceeds the typical research experiments with robots, the automated loading dock has been investigated in a series of experiments using

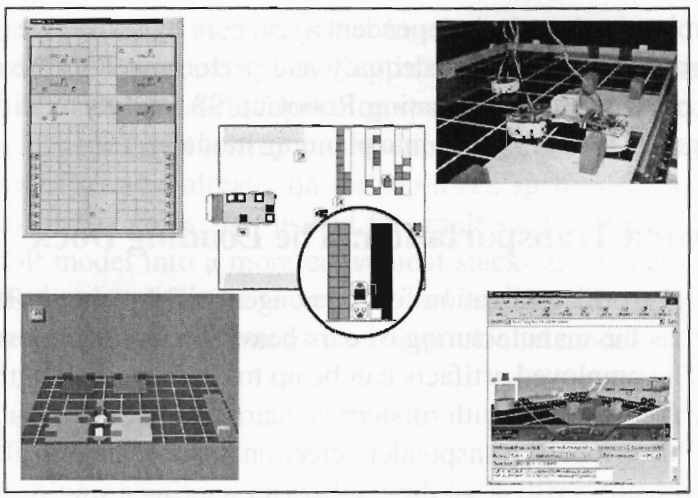

Figure 45: The Automated Loading Dock: Physical and Virtual Testbed

various versions of InteRRaP for controlling the forklifts. This ranges from two-dimensional discrete simulations to three-dimensional physical realisations with Khepera robots [KT93]. Recently, its simulation core has been extended into a web-based toolkit for distributed and interactive virtual worlds [JLG+99b]. Hence, the loading dock provides a demanding and comparative testbed for the practical impact of InteRRaP-R.

7.1.2 Programming Forklifts

Much of our forklifts' domain axiomatisation, in particular fluents, actions, and levels of abstraction, has already been anticipated in the previous sections to exemplify the general principles of InteRRaP-R. See the overview diagram (Figure 46) that uses the notation of the *Unified Modelling Language* (UML) [Oes97].
UML provides a semi-formal standard for the organisation of complex software. UML descriptions contain various *views* onto a software system, such as its physical realisation (*implementation diagrams* consisting of run-time *components*, *dependencies*, computing *devices*, and communication facilities) and its logical structure (*class diagrams* consisting of *packages*, *classes*, and *associations*). UML views are displayed and edited in a convenient graphical fashion. There are also automated tools for (re-)engineering available.
To present the domain-specific (Oz) representations of the Automated Loading Dock, the RoboCup soccer field, and the ROTEX working space, UML is an adequate device. Figure 46 displays a class diagram in which the nine levels of abstraction ($l_0, \ldots, l_8$) that InteRRaP-R forklifts are equipped with are modelled as UML packages. Fluents and actions are presented as UML classes. Their signature corresponds to the attributes and features of the implemented (Oz) constructs.
This diagram also describes the connections between these representations. For example, fluents and actions placed within a particular package are axiomatised at the corresponding level of abstraction, such as the `transport` macro being visible at level l_2 within the LPL. If a fluent decomposes into more primitive fluents (`at` decomposes into `atPos`) or if it is referenced in the conditions of a particular action

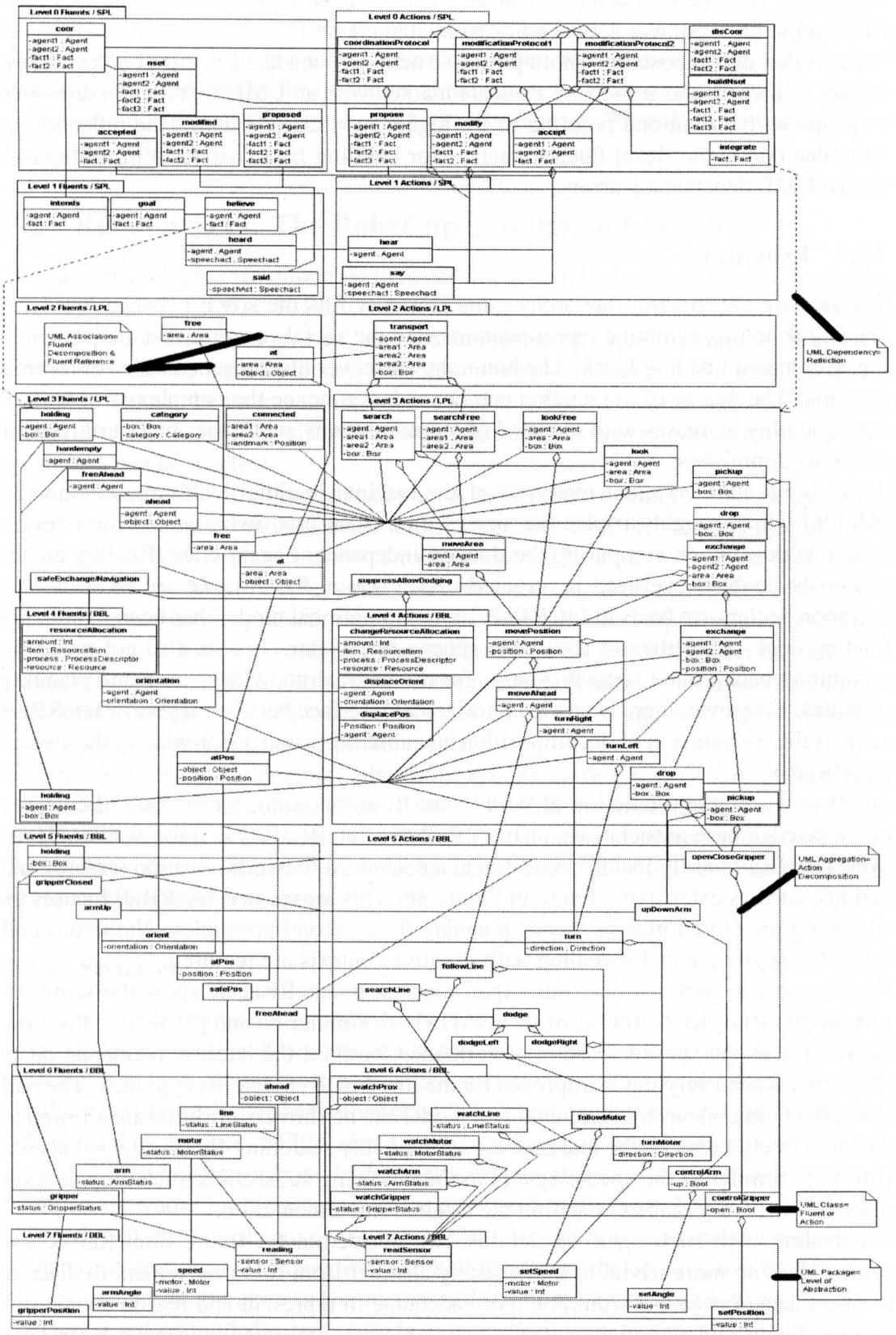

Figure 46: Fluents and Actions Representing the Automated Loading Dock

(`atPos` is precondition and effect of `moveAhead`), an UML association in the form of a simple line is drawn between the associated classes.
Macros that decompose into more primitive actions, such as `transport` decomposing into `search` and `searchFree`, are marked with an UML aggregation displayed as a line with a diamond pointing to the higher-level class. Finally, the reflection of formulae into higher-level fluents, such as for building `believes`, is modelled as a dotted UML dependency arrow.

7.1.3 Experience

The resulting abstraction hierarchy quite naturally fills the space between low-level sensing & acting, symbolic representations, and the social concepts that are present in the Automated Loading Dock. The hierarchy moreover fits the general idea of layered systems: The degree of association and dependency, hence the complexity of reasoning, smoothly increases with the level of representations, while the amount of relevant concepts diminishes.
There is no such coherent overview of the original loading dock representation in [Mül96] which roughly triples the size of ours (measured using `wc` on the relevant source code). When comparing the domain-independent agent code [Ros96], an additionally doubled overhead is revealed. As a reason for this, we see the lack of a common declarative basis in InteRRaP. Its 'computational model' has been tight to the loading dock rather than to a suitable generic interpreter. It does also not prescribe a common background semantics, e.g., for the vertical knowledge base and planning modules. Moreover, there is no clean horizontal interface between layers in InteRRaP, such as the instruments of decomposition and abstract resources provide in the case of InteRRaP-R.
To allow a suitable prediction of world state from incoming sensor data, the degree of knowledge base associations in InteRRaP and InteRRaP-R is quite similar. However, InteRRaP hardly installs conditional associations between situation features and actions, such as extensively found in Figure 46. This means that InteRRaP focuses on activating pre-stored macros, hence planning from second principles. Numerous and inflexible aggregations for dealing with situative contexts are required.
With respect to performance, our experiments with InteRRaP-R show the same degree of efficiency as the robots of [Mül96] in both simulation and physical realisation. We did not expect any increase in performance, because the latest implementation of [Ros96] has been very much improved for the line-based navigation approach. The real strength of our concurrent computational model can be shown if forklifts are allowed to navigate freely between the markers, e.g., for quickly searching for disoriented agents [Mor98]. Similar to the soccer agents in the following subsection, such emphasis on reactivity is hardly realisable within the original implementation.
For dealing with such open trajectories, two desiderata for future InteRRaP-R appeared: It is no more trivial to immediately abstract from robot movement to discrete position data. Instead, it would be more adequate to represent and reason about continuous fluents and activities in *HEC*, such as already presented in [Sha90, HT96] for other action calculi.
In such a setting, the LPL's deliberative control over the reactive BBL, such as by influencing the dodging reflex, must be enhanced. For example, in the current domain

axiomatisation, `suppressAllowDodging`is a more or less fixed part of the `exchangeBox` routine. If the LPL planning process would be able to predict the BBL's dodging reaction (or the reactive behaviour of other, well-known robots) as a kind of *natural event* triggered by the decisions of the agent, the planner could flexibly decide about the crucial usage of `suppressAllowDodging`. We will come back to these issues in Section 8.

7.2 Robot Soccer: The RoboCup Simulation League

Benchmark problems are important to foster research in key areas of computer science. For this purpose, [KWH+93] discusses the prospect of so-called *grand AI challenge problems* which are analogue to, say, the enterprise of landing a man on the moon. Grand AI challenges are visionary, long-term projects with an appealing goal and official rules that can be incrementally approached at different degrees and from different views. Grand AI challenges should have a significant social impact, such that public awareness can be reached.
For enforcing real-world orientation in agent design, i.e., dealing with high dynamics, continuity, incomplete and dispersed information, and team-orientation, one surprising challenge has been found in terms of robot soccer: The kind of machine intelligence needed to build a team of robots that can defeat the Brazilian national soccer team is closer to an industrial application profile than, for example, Deep Blue's undoubtfully significant contribution to computer chess. Moreover, the game of soccer has a largely competitive aspect which allows to measure research results in terms of tournaments and championships. Finally, soccer provides an excellent and motivating vehicle to transport research results to the public media.
To vitalise (D)AI in that direction, the *Robot World Cup Initiative* (RoboCup) has been founded in 1993 [KAK+97]. Since 1996, RoboCup organises annual world championships including workshops, tournaments, evaluation sessions, and publications. Today, there are several hundred researchers, assistants, and students from all over the world applying their concepts to that domain.

7.2.1 Scenario

RoboCup competitions are organised into five leagues. Small robots with maximally 15cm diameter, such as Kheperas, are placed on a kind of ping-pong table and play with a golf ball. A global camera delivers a global view to their individual processing facilities. In Figure 47, we have illustrated Sony's legged platform and the middle-size robots. In their leagues, a FIFA size 4 ball is used and on-board vision is required. In the still closed league for bi-pedal humanoid robots, there exists yet a single platform from Honda whose basic soccer skills are envisaged by the year 2002.
Currently, all these hardware platforms still suffer from very fundamental reactive deficiencies in recognition, navigation, and ball handling. Hence, adaptive higher capabilities such as deliberative soccer tactics and social soccer strategies would be out of question if not being dedicated a fifth league that builds on a more simplistic, but still realistic simulation of robot soccer [KTS+98].
To this end, a simulation engine [Nod95] has been set up to which 22 separate agent processes can connect from client machines via the UDP network protocol. This sim-

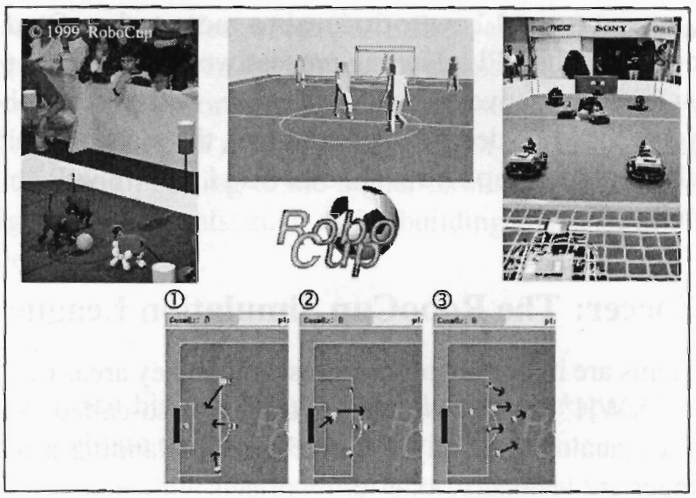

Figure 47: The RoboCup Challenge: Robot and Simulation Leagues

ulation server emulates the noisy and incomplete field of view of all the soccer players and sends them data packages with synthetic perception roughly every 100ms.
In turn, the server accepts fine-grained action commands, such as dashing, turning, kicking, and catching that influence the simulated physical bodies including velocity and stamina. The visualisation is done via separate 2D- and 3D-applets over the Internet (Figure 47). We have participated the simulation track of the RoboCup'98 world championships with a team of InteRRaP-R soccer agents (*CosmOz*).

7.2.2 Programming Soccer Players

Each CosmOz player is equipped with the identical domain representation incorporating five levels of abstraction $l_0, \ldots, l_4$ (see the UML class diagram in Figure 48). Minor excerpts have already been anticipated in the previous sections. The discrimination of players partly emerges due to their different situative contexts, but more effectively results from explicit social representations (team roles) and coordination.

BBL: Reactive Soccer Skills The CosmOz BBL implements basic soccer skills at levels l_3 and l_4. For example, the reception of synthetic perception is done via the recursive `processPerception`action at level l_4. Roughly every 100ms, the soccer server signals a `Perception` datagram containing the simulated field of view to the behaviour executing process. `Perception` contains relative positions (distance and angle) and movements (change in distance and angle) of the visible objects such as players, ball, goals, lines, and flags.
The perceived fluents `player`, `ball`, `goal`, `line`, and `flag` are explained by the world model as the precondition of `processPerception`and the effect of the `displace` miracle. The reasoning involved in the construction of level l_4 — including the firing of reactive desires and the elaboration of subsequent reflexes — is very simple and fast: Depending on the rapid trajectory of the near ball, low level events, such as `complexKick`, `catch`, and `approachBall` (decomposes into `turn` and `dash`) are invoked. We have indeed measured that, from receiving `Perception` to

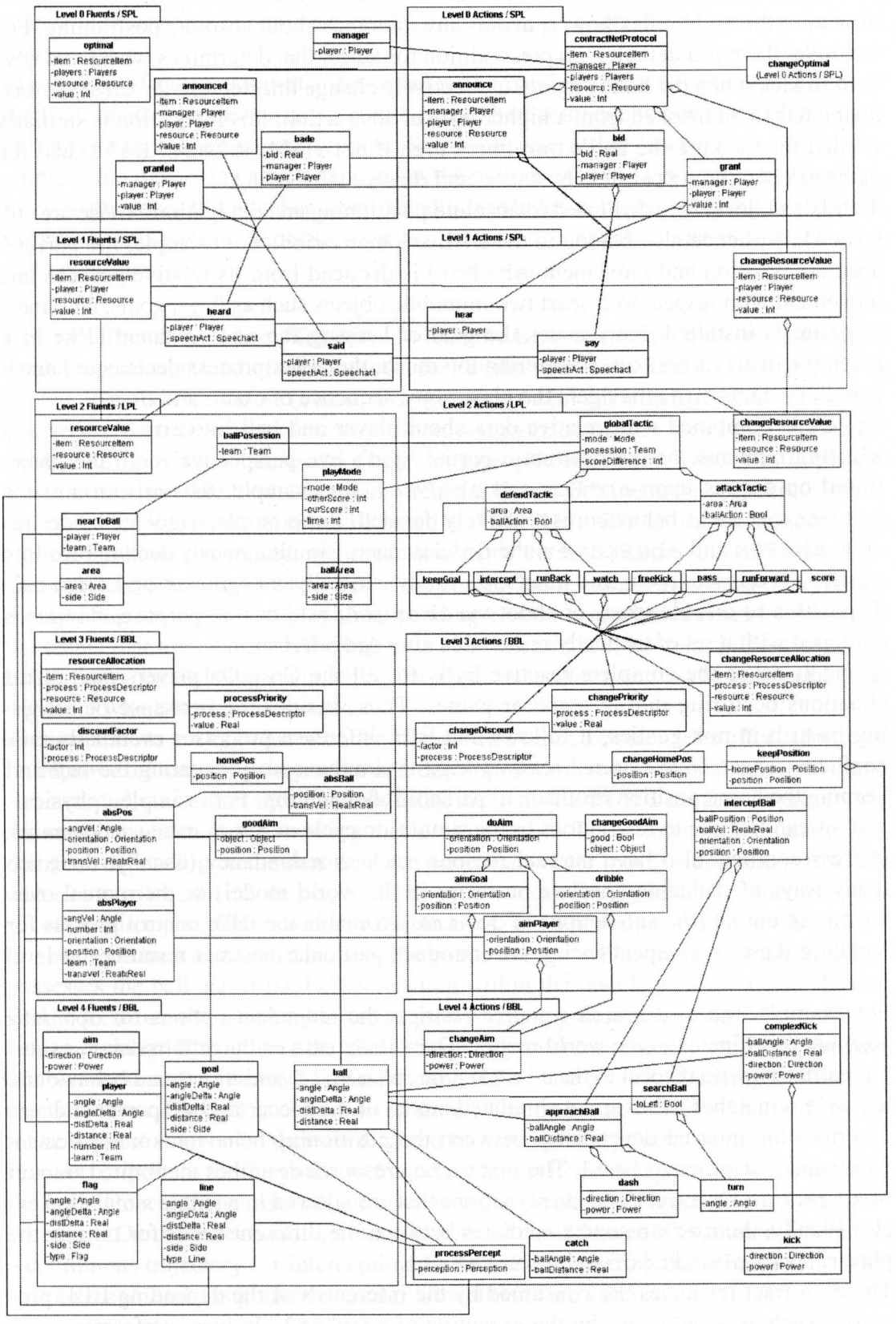

Figure 48: Fluents and Actions Representing the Simulated Soccer Game

issuing an action signal, only a fraction of the required 100ms elapses!
To realise this tight reflex bow, l_4 avoids any inference about absolute positioning. For example, there is a default `aim` precondition for `kick` that determines where and how hard to kick when the ball is near. To effectively change this default, the `changeAim` action has to be invoked from a higher level of abstraction; this invocation is similarly needed for tracking the ball's movement even if not visible (`searchBall`) and for navigating on the soccer field by `turn` and `dash`.
To this end, level l_3 is dedicated to absolute positioning and navigation. These are still basic skills, hence also belong to the BBL axiomatisation. For example, the player's absolute position and movement (`absPos`) is deduced from its relative position and movement with respect to at least two immobile objects such as flags, goals, and lines. A desire is installed that pursues the goal of keeping the `atPos` fluent close to a given (tactical) `homePos`. If differing too much, the reflex process decides to launch `keepPosition` to renavigate the player by a sequence of `turn` and `dash`.
`atPos` is combined with relative data about player and ball into `absPlayer` and `absBall` fluents that reconstruct a partial 'god's eye perspective' onto the game. Based on desires upon `absPos` and `absBall`, for example, the activation of the `interceptBall` behaviour is reactively decided. For example, triggered by desires upon `absPos` and `absPlayer`, the `doAim` macro simultaneously decomposes into `aimSelf`, `aimPlayer`, and `aimGoal` for dynamically changing `complexKick`'s disposition to dribble, pass, or shoot via `changeAim`. For that purpose, `doAim` is equipped with a set of tactically reasonable aims (`goodAim`).
l_3 and l_4 cover the complete reactive basis for all the CosmOz players and all the situations occurring during a soccer game. Thus, besides the nonsense of activating `catch` in non-goalies, it follows that BBL inference processes eventually raise conflicts. These have external reasons, e.g., if simultaneously haunting the ball and keeping the home position results in a 'paranoid' floundering. For example, 'physical' actions can fail due to restrictions in the simulation cycle or due to exhausted stamina. Process conflicts also have internal reasons, such as redundancy (usually, there are many ways of deducing absolute positions in the world model) or the mutual overwriting of output (the sub-actions of `doAim`). To enable the BBL control process for handling these interdependencies, we introduce particular abstract resources at level l_4.
For example, the `landmark` resource restricts the redundant options for obtaining absolute coordinates in the world model. The `aimFocus` resource introduces a similar restriction with respect to the `aimPlayer`, `aimSelf`, and `aimGoal` behaviours. `stamina` matches the external simulated model of the soccer agents' power and also describes the internal dependency between the positioning behaviours `keepPosition` and `interceptBall`. The `actuator` resource describes the limited amount of `kick`, `catch`, `turn`, and `dash` actions that are allowed in a single simulation cycle. Finally, the `neck` resource mediates between the different needs for turning the player in `keepPosition` and `searchBall`.
These abstract resources are consumed by the inferences of the depending BBL processes, such as `aimFocus` by the execution of `aimGoal`. In turn, performance reports are generated from the process status: In order to adjust the current `aim`, the `aimGoal` behaviour analyses the distance and the direction to the opponent's goal in the `atPos` condition. The greater the distance, the less likely the next kick will score

a goal and the less useful the current chunk of execution has been. Similarly, performance measures for executing `aimPlayer`, `aimSelf`, and other processes can be found.
Based on performance reports, the signal routing and resource allocation facilities of the BBL control process install a conflict-free short-term behaviour in which the concurrent inferences are smoothly interpolated into interesting 'soccer moves'. An example is the right-wing attack depicted at the bottom of Fig. 47 in which, by the `aimFocus` resource, the attacker agent steadily mediates between shooting to the goal, dribbling, and passing depending on goal distance, navigation space, and the team mate positions.

LPL: Deliberative Soccer Tactics To lead the BBL control process towards establishing a particular kind of soccer move, such as the right-flank attack, it is important to choose it parameterisation (`resourceAllocation` via `changeResourceAllocation`, `processPriority` via `changeProcessPriority`, and `discountFactor` via `changeDiscount`) appropriately. Moreover, it is important to identify a suitable `homePos` and a reasonable set of aims in `goodAim`. This is not in the responsibility of the skill-based and fast BBL. Rather, this is subject to the tactical prediction, planning, and execution inside the LPL making use of level l_2.
Tactical fluents are abstract features of the current game situation, such as `playMode` (comprising score, elapsed play time, pending kick-in's, etc.), `nearBall` (players of the teams that are nearest to the ball), `ballPosession` (the team in control of the ball), `area` (the field region in which the player is located), and `ballArea` (the field region in which the ball is located).
Tactical actions are aggregated under the `globalTactic` macro. Its effect should be to improve the current `playmode` and decomposes into two optional sub-goals (and sub-macros), i.e., either scoring goals using the `attackTactic` macro or preventing the opponent from doing so using `defendTactic`. These intermediate tactics decompose depending on the situation into particular soccer moves, such as `keepGoal`, `watch`, `runForward`, or `pass`. Each soccer move finally installs a characteristic set of resource parameterisations, valuable aims, and a home position in the BBL.
For example the `keepGoal` move of the goalie in Figure 47 is invoked if the opponent possesses the ball and if the ball is residing within the own half of the field. Herein, BBL's `processPriority` is set to prefer catching before kicking before turning before dashing. This allows the goalie to quickly intercept the movement of the ball, afterwards shooting it away — possibly to a team mate nearby. Catching and kicking are vital; their discount is chosen to be small. Since `aimGoal` and `aimSelf` do not make sense in this mode, they are disabled by setting their priority to zero.
It is also important for the goalie to frequently adjust its central position while tracking the ball. Hence, the `stamina` resource is available in small amounts for `keepPosition`. On the other hand, `interceptBall` should be able to suddenly generate a continuous trajectory for intercepting the ball if it is entering the penalty area. Thus `interceptBall` is granted a greater allocation of `stamina`.

SPL: Social Soccer Strategies Soccer is not a purely tactical game in which the goal-oriented behaviour of two or three players suffices. Soccer is an inherently strate-

gic game that requires the coordinated team-play of eleven players in order to beat the opponent. For example, having more than one soccer agent deciding to play goalie by activating `keepGoal` will most likely result in a disastrous outcome. The required soccer strategies are represented and reasoned about in the SPL at level l_0 and l_1. Strategies are realised by influencing the LPL control process' allocation of abstract resources. In this case, CosmOz players own a particular `roles` resource that is consumed by executing tactical actions.
For example, `keepGoal` requires a non-zero amount of the `goalie` item of `roles`.
For example, `score` is only available, if the player is assigned to be an `attacker`.
For example, the decomposition of `runBack` and `runForward` depends on the amount of roles available at that time.
A concrete soccer strategy then consists of reasonably distributing a set of role items to the whole team, such as an offensive $1-3-3-4$ strategy consisting of one player assigned to be a goalie, three players assigned as defenders, three middle-fielders, and four attackers. This can be technically established as a default for the `resourceValue` fluent at level l_2. This can be moreover dynamically negotiated between a manager agent (the trainer) and a set of bidders using the contract-net protocol as elaborated in Section 5 and depicted in Figure 48.

7.2.3 Experience

RoboCup was first intended only as a secondary domain for InteRRaP-R. When we fully recognised its importance for testing hybrid designs and resource-adapting systems, we decided to participate the official world championships in Paris, 1998. The rationale was not to implement a champion team by coding sophisticated skills, tactics, and strategies in C, but to demonstrate that our generic, resource-based framework with its constraint-based implementation can be practically customised to such a demanding and competitive setting.
In contrast to other teams that already started their activity back in 1996 and that engaged up to 10 developers in the meantime, the realisation of CosmOz took a total of 3 person-months. Its source code is about of the same size than the one of InteRRaP-R forklifts. At the time of the competition, the reorganisation of the team strategy was not yet ready and no support for the offside rule was present. But, we made heavy use of the Mozart-built-in distribution facilities to conveniently release the agent team to a designated pool of computers — this was remarkably easy to achieve thanks to the transparent Oz language.
The first observation was that the resulting CosmOz agents were evidently able to effectively operate in the close-to-real-time setting, exhibiting reactive skills, deliberative tactics, and social strategies. We could not adequately achieve this simultaneous combination of (soccer) facilities, for example, with the original activation & commitment principle of InteRRaP.
The unexpected result (because of our research objective) was that CosmOz reached the quarter-finals of the RoboCup'98 competition beating some low-level-coded teams with scores of up to 16:0. An analysis of the log-files has shown that this is partially due to the broad functionality covered by CosmOz when playing against teams focusing either on skills, tactics, or strategies (see [Asa99]). This result is also due to the explicit management of resources in InteRRaP-R: From 1997 to 1998, the RoboCup regulation

made the simulation of stamina more restrictive. Teams which did not address this resource constraint became visibly tired and lethargic soon after kickoff.
For improvement, we could identify many issues with respect to domain representation (that was not laid much emphasis on) and few aspects with respect to the agent model (that performed ways better than to be expected). For example, the recognition of the opponent's strategy will certainly be important in future RoboCup competitions. This is in-principle realisable in InteRRaP-R, i.e., its knowledge base processes, but requires social representations that generalise over many alternative team designs.
Currently, the BBL resources, such as stamina, and parameters for their allocation are abstractly hidden in the LPL representations. We noticed that most of these representations are robust with respect to changes in modelling, while some are quite sensitive, in particular the chosen performance scale. To enable the LPL and SPL planning to incorporate, e.g., the state of stamina into tactical and strategic decisions could significantly contribute to the adaptive behaviour of the multi-agent team; the Fluent Calculus [Thi99] is a theory of time and action that is capable of efficiently representing and reasoning about such resources. We could also think of incorporating a *learning* mechanism into the InteRRaP-R control processes to improve the estimation of process performance.

7.3 Tele-Robotics: The ROTEX Space Experiment

Service artifacts are used more and more in areas which are too hazardous or inaccessible to humans. Examples are robots for inspecting pipelines (the KURT platform, [fAIS99]), for patrolling poisoned parts of nuclear plants (the Soviet Lunokhod rover [Wie99] during the Chernobyl disaster), for exploring unstable buildings after earthquakes (the RoboCup Rescue Initiative [Kit99]), and for examining deep-sea cables. There are also space-robots that perform assembly tasks on planet's surfaces (see the NASA Pathfinder project [Gol98]) and in deep space [fRuS99] for, e.g., repairing satellites under zero-gravity.
Especially in the latter case, we recognise that the possibilities for establishing a high-bandwidth communication channel between the remote artifacts and a mission control site are highly restricted. For example, there is already an accumulated delay of about 6 seconds when tele-operating an orbiting system. Since these robots are additionally placed in rather open and dangerous environments, their control poses an ultimate research problem that the area of *Tele-Robotics* is dedicated to.
In the *Tele-Sensor* approach [BLSH95], agent technology has proven useful: The remote operation of semi-autonomous space robots is combined with a simulation-based ground operation. Remote agents are equipped with enough local autonomy to reactively decompose and execute given high-level tasks from the ground site. As the bases for issuing these commands, the ground site runs a predictive simulation that is frequently fed by the status of the remote system and that compensates the communication delay.

7.3.1 Scenario

The first Tele-Sensor experiment, ROTEX, was carried out on-board the Columbia shuttle flight ST55 from April 26 to May 6, 1993. In this experiment, a small six-

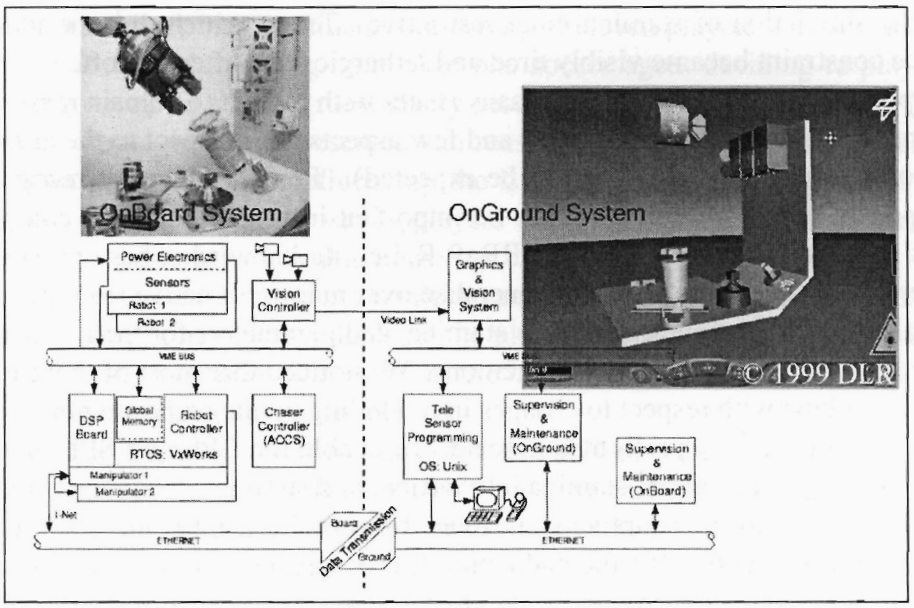

Figure 49: The ROTEX Space-Robot in the Tele-Sensor Approach

axis robot with a working space of about 1 m^3 was mounted inside a space-lab rack (Figure 49). Its gripper was provided with a number of sensors, such as force-torque wrist sensors, tactile arrays, grasping force control, laser-range finders, and stereo cameras. In order to demonstrate servicing prototype capabilities, three basic tasks were performed: The assembly of a mechanical grid structure, the (dis-)connection of an electrical plug (the so-called Orbit-Replaceable-Unit — ORU), and the grasping and stapling of cubes. In the experiment, the commands to pick, manipulate, and place particular parts were given by humans from the ground site. The humans were provided with a web-based 3D visualisation of the simulated working space.
In order to increase the degree of automation, we suggested the use of hybrid InteRRaP-R agents i) in the ground system to allow for a higher-level operational interface and ii) in the remote system to increase autonomy and thus the range of tolerable communication delays. The preliminary tests that we have conducted focus on the first aspect. They have been performed using the original working space and a variation of it, in which a variable set of up to 40 cubes was present.

7.3.2 Programming Ground-Control Agents

In the UML diagram of Figure 50, the domain representation for a ground-control ROTEX agent is illustrated. It turns out as a quite conventional single-agent domain description: Two levels of abstraction l_0 (LPL) and l_1 (BBL) describe states and movements in the three-dimensional working space.
The ground-control agent is connected via a TCP/IP network connection to the predictive simulation which is either coupled to the physical working space or runs in a special off-line mode. A human operator is provided with a browser displaying the state of the simulation (through a *Virtual Reality Modelling Language*-plug-in) and the agent (through the InteRRaP-R GUI). The browser allows the convenient specification of high-level goals which are then immediately delegated to the agent.

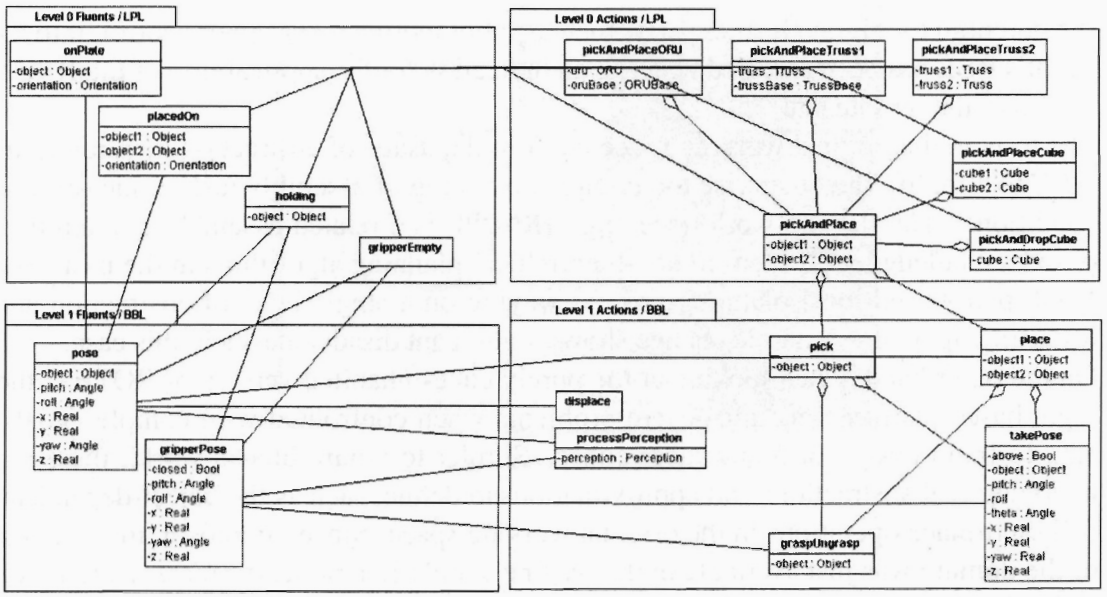

Figure 50: Fluents and Actions Representing the ROTEX Working Space

From the simulation, the agent frequently receives pose information of objects (`pose`) and robot (`gripperPose`). These contain positioning in three degrees of freedom (`x`, `y`, and `z`) and orientation in three degrees of freedom (`pitch`, `yaw`, and `roll`). The incoming signals are processed via the `processPerception` action at level l_0. The simulation server accepts commands for collision-free six-axis motion (`takePose`) and tactile gripper control (`graspUngrasp`). The agent's BBL aggregates these low-level commands into shape- and pose-dependent `pick` and `place` macros.

At level l_1, the pose data is abstracted into a blocks-world-like representation using the `onPlate`, `above`, `holding` and `gripperEmpty` fluents. For establishing particular object constellations according to user requests, l_1 distinguishes several pick-and-place macros: `pickAndPlaceORU` is to install an ORU device onto its base. `pickAndPlaceTruss1` puts a truss on a truss base, while `pickAndPlaceTruss2` serves to staple several trusses. `pickAndPlaceCube` staples the cubes, while `pickAndDropCube` places cubes on the working space plate.

For evaluation purposes, we also built a variation of InteRRaP-R in which the *HEC*-based local planning process was replaced by the graph-based planning algorithm of [BF95]. Its Oz implementation [Wen99] has been adapted to the COOP computational model and its logic data structures: Because planning-graphs are not incremental, the algorithm is restarted when receiving a signal. Because planning-graphs are non-hierarchical, the algorithm uses only the above `pickAndPlace` operator. In both variations, the ground control agents had their SPL completely disabled.

7.3.3 Experience

We have run a small test series by running each variation of InteRRaP-R in each of the two ROTEX working spaces. This has demonstrated that both variations of InteRRaP-R provide for a more convenient and declarative user interface than the pure Tele-Sensor approach. Moreover, in both versions of the ROTEX working space versions,

the potential to migrate further parts, such as robot motion and gripper control, into the agent's BBL has been revealed which is an indication for the applicability of InteRRaP-R also to the remote site.
The tests in the original working space confirm the usage of abstraction hierarchies in *HEC* to simplify the otherwise too complex planning of assembly tasks under on-line conditions. The flat tire world (see, e.g., [RN95]) is a related assembly domain that posed a challenging problem to non-hierarchical planning algorithms in the past. The graph-plan-based local planning process staying on a single level of abstraction and restarting upon any dynamic change shows significant disadvantages in this case.
On the other hand, when looking at the purely cube-inhabited version of ROTEX, the logic-based planner runs into severe problems when confronted with complex goals, e.g., to build towers of 5 and more cubes. Similar to a pure blocks-world, there are no longer real abstractions and approximations to define, such as the object-dependent pick-and-place operations in the original working space. Since, moreover, this version of the domain was just available in the off-line simulation mode, the incrementality of abductive *HEC* is an expensive and unnecessary feature. The state-based graph-plan approach has been designed to cover exactly these (artifical?) cases; its latest version, for example, solves the off-line Tower-Of-Hanoi with dozens of slices.
Experiments such as ROTEX are likely to be extended in future space robotics. For example, it is straightforward to introduce several robot arms that coordinately and efficiently service a shared working space. Another example is the NASA deep space project [Pro99] in which several autonomously controlled shuttles are to build up a kind of 'virtual satellite dish'. For coordinating their flight activity, a control architecture with an equivalent to InteRRaP-R's SPL will be needed.

7.4 Bottom Line

This section has documented the results of three case studies that we conducted in order to evaluate the practical impact of InteRRaP-R and its design methodology. The three scenarios that we have chosen are representative instances of application areas for broad agents. They go beyond the typical laboratory experiments with robots and simulations in agent research.
For all scenarios (that would be difficult to program from scratch) we have discussed how the generic agent framework is to be customised by outlining the complete domain representations. Due to the declarative background of InteRRaP-R, the corresponding programming efforts are significantly smaller than for the predecessor InteRRaP. Moreover, the domain code is quite comprehensible which can be attributed to the adequate devices of abstract hierarchies and abstract resources. These are intuitive tools for programming broad agents in all three domains.
The experience with these scenarios confirms our claim that the InteRRaP-R implementation, under support by the efficient Mozart system, is able to install reactive, deliberative, and social facilities in non-trivial environments. It is also shown that the concept of layering and abstract resources improves the practicability of the predecessor InteRRaP and alternative agent designs.
Finally, we have identified possible improvements in InteRRaP-R that could ease the modelling of such demanding domains in the future. We will discuss their compliance to the existing model and the design methodology in the following conclusion.

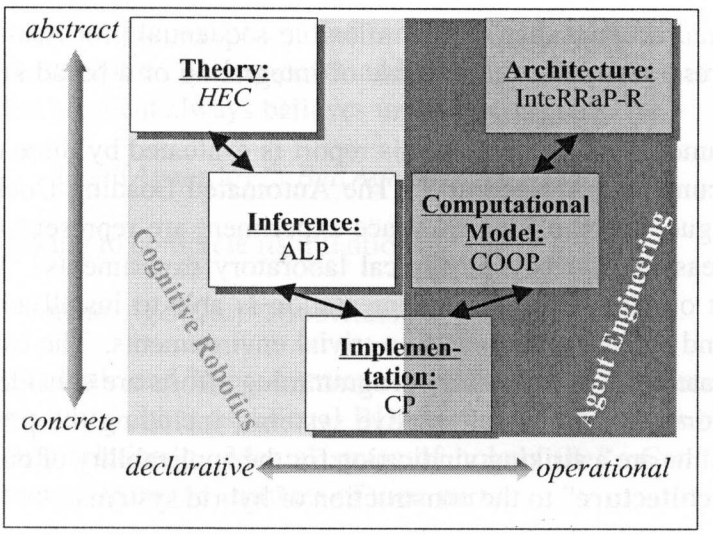

Figure 51: The Instantiated Design Space of Agents

8 Conclusion

Aiming at Shoham's notion of Agent-Oriented Programming [Sho90] for broad and industrial-strength systems, this report has presented a design methodology for hybrid agents, in particular for the layered InteRRaP model of [Mül96]. **Section 1** presented the Design Space of Agents as a way of structuring agent development process into five interconnected stages of specification organised along the two dimensions of declarativity and abstraction (Figure 51). Each specification stage — architecture, computational model, theory, inference, and implementation — contributes a characteristic set of methods, notations, design patterns, and tools all of which are equally important for obtaining models bridging theory and practice.

The Design Space of Agents reconciles and extends the isolated, but complementary approaches of Agent Engineering (architectural descriptions as initiated by Bratman et al. [BIP87] and promoted inter alia by Ferguson [Fer92] and Müller [Mül96]) and Cognitive Robotics (theoretical and inferential descriptions as initiated by McCarthy & Hayes [McC58] and promoted inter alia by Reiter [Rei99] and Kowalski et al. [KS96b]). For this purpose, computational models provide the crucial (formal) interface declarative and operational design issues as well as between conceptualisation and implementation. On the one hand, this is close to the DESIRE approach to prototyping arbitrary agent models [DKT94]. On the other hand, where DESIRE is a toolkit for organising modules using various types of logic, we aim at a coherent framework for building and actually implementing hybrid systems incorporating a single logic of time and action.

Throughout **Sections 2 to 6**, we have completely instantiated the Design Space of Agents with the reconstructed InteRRaP-R model (Figure 51). So far, this enterprise has a single match in the research on BDI theory [RG91], PRS architecture [GL87], AgentSpeak computational model [dL98], and the dMARS implementation. In contrast to the epistemic-logic foundation of BDI, our *HEC* theory builds on 'executable' logic programming. In contrast to the unified architecture of PRS, InteRRaP-R rep-

resents a layered account that avoids inflexible sequential processing and expensive consistency-preservation for the purpose of integrating of a broad spectrum of functionalities.
The design framework presented in this report is evaluated by three case studies that have been documented in **Section 7**. The Automated Loading Dock, the RoboCup simulation league, and the ROTEX space experiment are representative instances of application areas that go beyond typical laboratory experiments. The case studies confirmed that our InteRRaP-R implementation is able to install adaptive, reactive, deliberative, and social facilities in non-trivial environments. The case studies moreover demonstrated that the necessary programming efforts are considerably easier and more straightforward than for alternative systems, including the predecessor InteRRaP. These results are a striking vindication for the applicability of our slogan "**Agent = Logic + Architecture**" to the construction of hybrid systems.

8.1 Hybrid Agents and Holonic Multi-agent Systems

A DAI *application methodology* is a recipe for structuring MAS of up to thousands of agents according to a given domain. In an application methodology, the choice of the right agent design, such as a hybrid model for obtaining broad functionality, is just one stage of specification [Lin99]. Hence, it is important to define interfaces between application methodologies and design methodologies, such as the Design Space of Agents.
In many application areas, e.g., Telematics [BFV98], the identification of agents and their functions is not a trivial enterprise. For example, different shipping companies have to be modelled as competitive agents. Dispersed freight centres and transportation fleets of a single shipping company represent cooperative agents. It is also quite natural to model individual trucks of a fleet as autonomous agents equipped with transportation goals and route plans. And it can be appropriate to individuate the possibly different needs and different destinations of tractors, trailers, cargo containers, and the human driver of a single truck [Vie99].
The consequence is that a practical MAS should be organised as a society of agents which are again composed of sub-agents in a self-similar way. In other words, structures with a higher-level of autonomy (groups, agents, layers) should be decomposed into similar structures with a lower degree of autonomy (sub-groups, agents, layers, processes). For this *fractal* principle, Siekmann et al. [GSV99] have coined the term *holonic* MAS derived from the Greek *holos* meaning whole and the suffix *-on* meaning part. We have argued in [JG97, JG98] that this principle is likely to bridge the *micro-macro* gap of MAS design. Hence, a resource-adapting MAS architecture is presented in which *social layers* are built up by so-called *representative agents* that control the computation of subordinate agents by means of abstract resources quite similar to the way that control processes affect subordinate computations within a single layer of InteRRaP-R. In [Ger99], this architecture is extended to the ability of self-organisation, i.e., the social resources that are monitored and reconfigured are, e.g., the number of agents in a group, their design, and their communication facilities.
The holonic principle also serves as a theoretical means, e.g., to analyse InteRRaP-R in comparison with the well-known BDI approach (see [Jun99a]). In traditional BDI logic [RG91], the state of an agent is described by temporal and modal formulae

such as $Bel(Agent, \omega)$, $Goal(Agent, \omega)$, and $Intend(Agent, \omega)$ that are interpreted over a possible-worlds structure. BDI imposes additional semantical constraints onto that structure, e.g., that an agent always believes in having its goals

$$Goal(Agent, \omega) \supset Bel(Agent, Goal(Agent, \omega))$$

and is always working to complete its intentions

$$Intend(Agent, \omega) \supset \text{inevitable}\Diamond\neg Intend(Agent, \omega)$$

For representing holonic and layered structures, we can also introduce holons which are groups composed of either traditional BDI agents or sub-holons by the transitive and anti-symmetric relation $\lhd$. Mental states of holons (Bel^*, $Goal^*$, and $Intend^*$) transparently 'emerge' from the attitudes of their parts:

$$Bel^*(Holon, \omega) \equiv \tilde{\exists} Agent \lhd Holon \wedge Bel(Agent, \omega)$$

(Meta-)control hierarchies, such as present in InteRRaP-R, are defined by the partial order $\blacktriangleleft$:

$$Agent_1 \blacktriangleleft Agent_2 \supset (Bel(Agent_1, \omega) \supset Bel(Agent_2, \omega))$$

$$Agent_1 \blacktriangleleft Agent_2 \supset (\neg Bel(Agent_2, \omega) \supset \neg Goal(Agent_2, \text{optional}\Diamond Bel(Agent_1, \omega)))$$

The first of the above conditions states that any higher layer (here: agent) has introspection into the subordinate layer. The second condition (and analogue ones for goals and intentions) states that any higher layer will only reconfigure the subordinate layer consistently with its current mental attitudes. A formal result that we have obtained in this *Holonic and Layered BDI* (HLBDI) logic is that, provided each InteRRaP-R layer follows a suitable *commitment strategy* for realising persistent intentions, the complete InteRRaP-R holon, though incorporating inconsistent attitudes, exhibits a quite similar strategy [Jun99a]. So far, HLBDI is restricted to modelling layers as traditional BDI agents. Once we manage to connect HLBDI to the process level of COOP, it could provide a valuable verification tool for hybrid agents and their corresponding MAS in general.

8.2 Learning and Evaluating Resource Control

Abstract resources are an instrument for the real-time scheduling of concurrent computational activities. In this context, COOP relies on internal profiling, i.e., the scheduler uses a built-in performance measurement. As in other approaches to meta-control [BD94, RW91], this leads to short-term adaptivity provided that the designer supports (approximately) correct evaluation functions for processes. However, there is the question where to get these functions if the domain is not sufficiently known in advance. In the RoboCup domain, for example, we have experienced that the usefulness of dribbling and passing the ball depends on the skills of the opponent team.
In such cases, adaptive systems such as the cognitive architecture ACT-R [And93] envisage to learn long-term domain characteristics such as the quality of particular scheduling decisions from *external reinforcement* at runtime [KL96]. Algorithmic means are, for example, the memory-based reasoning procedure of [SW86] and the

bucket-brigade algorithm of [Hol85]. However, it is difficult to apply these algorithms to real-time scheduling. Feedback that results from, e.g., scoring a goal or loosing the ball, is too sporadic to drive required decision cycles of about 100 milliseconds.
In [JLG99a], we have thus presented an extended COOP control process that does not learn the quality of scheduling decisions in the first place, but rather learns *confidence* into the embedded internal profiling mechanisms. Hence, confidence measures are integrated as an additional scaling factor into the allocation algorithm presented in Section 3. They are gathered from external stimuli by a combination of memory-based learning (to store and retrieve exemplary situations, process evaluations, and the resulting feedback) and bucket-brigade learning (to distribute feedback among the responsible processes).
For the practical evaluation of such algorithms,the identification of benchmarking scenarios is quite difficult. In this report, we have relied on observational experience with the running systems in the RoboCup domain, for example. With a bit of systematics, RoboCup can also be used as an empirical testbed: To measure the impact of particular concepts, such as resource-control, it is possible to let various versions of the same agent model, e.g., one version with resource-control disabled, one version with slightly changed parameterisation, and one version including learning, play against fixed opponents in a statistically relevant number of games. The collected data for analysis should not only cover final scores, but also percentage of ball possession, successful passes, and other performance-related issues.
The 1999 version of the RoboCup simulation supports the gathering of such data: Teams are allowed to connect a special trainer agent to the server engine that is equipped with a god's eye view. Since its synthetic perception is close to ordinary players, we could already implement a preliminary situation recognition to the CosmOz trainer. Recently, an inductive learning algorithm has been applied in the ISAAC trainer [RTM99] that is able to generate rule-based team ratings from observing goal shoots across various log-files.

8.3 Ramification and Natural Events

Section 4 discussed the impact of adversary environments to the specification of reasoning principles for agents. Worst case assumptions, such as applied in *ECJ* and *HEC*, are one tool to address the agent's vulnerability. What these calculi do not yet cover is that other objects or agents in the environment are (pro-)active in themselves, i.e., they affect the world by simple reaction to and by complex deliberation on the agent's proper actions. For example, a robot that opens its grippers while its arm is raised triggers the carried box to move (fall out of the gripper). For example, a robot that pushes a single box triggers the movement of several boxes if located in a row. For a forklift's local planning process deciding about navigation, even the reflex process that is located in the same hybrid agent is a source of such 'natural' activities, e.g., by forcing the robot to dodge when meeting an obstacle.
These are all special cases of the general *ramification* problem stated in [GS88]: Rather than being representable as immediate effects of the agent's actions, such as of moving ahead or of opening the gripper, these natural activities are indirect and situation-dependent consequences of the world state. An advanced planner has to predict these consequences and either avoid them (by circumventing obstacles or by suppressing the

dodge reflex) or cope with their effects (by forcing the BBL to dodge only to the right or by re-approaching the partner for a box exchange).
For state-based calculi, [Thi97] has presented a formalisation of ramifications. It identifies particular, *instable* world states that violate given constraints, such as the impossibility of a forklift's heading an object if its dodging reflex is enabled. There are causal relationships inherent to these constraints: Active dodging causes a change in the robot's position immediately after heading an obstacle. By incrementally applying such causal rules for resolving the constraint violations, any state-based reasoner is thus enabled to arrive again at 'ordinary' stable states and to continue its regular inference.
To address ramifications in narrative-based formalisms, *natural events* [KM97a] have been proposed which are 'automatically' introduced by predefined trigger situations. For example, dodging always happens if an obstacle is ahead and the respective reflex is simultaneously enabled. This can be expressed in *HEC* as follows:

$$\begin{aligned}(DOMTRIG)\quad & \tilde{\forall} triggers(E,A,T_1,T_2,L) \equiv \tilde{\exists} L \doteq l_3 \wedge A \doteq dodge \wedge \\ & holds(ahead(Obstacle),T_1,T_2,nil,\dot{1},E,L) \wedge \\ & holds(saveNavigation,T_1,T_2,nil,\dot{1},E,L)\end{aligned}$$

$$(NAT1)\quad \tilde{\forall} triggers(E,A,T_1,T_2,L) \supset \tilde{\exists} happens(E,A,T_1,T_2,L)$$

We recognise that basic *HEC* features, such as worst-case assumptions and levels of abstraction, are compatible with and even useful for such extended reasoning principles which enable, e.g., the LPL planner to predict the BBL's dodging decision. Without the following constraint, however, the planner would predict the triggered reflex infinitely often as a consequence of the same situation:

$$\begin{aligned}(NAT2)\quad & \tilde{\forall} triggers(E_1,A,T_1,T_2,L) \wedge \\ & triggers(E_2,A,T_3,T_4,L) \supset \tilde{\exists} disjoint(T_1,T_2,T_3,T_4) \vee \tilde{\exists} E_1 \doteq E_2\end{aligned}$$

Modelling natural events in *HEC* via integrity, as done by *NAT*1, lets the planner cope with the effects of ramification and at the same time keeps the reasonable properties of well-definedness and termination upon finite residues. It is however not possible to infer counter-measures, such as the disablement of the dodge reflex, which avoid in advance the natural being triggered. For that purpose, we are currently experimenting with a version of *HEC* in which the *triggers* condition is closely integrated into the definitions of the core calculus.

8.4 The Inferential Frame Problem and Resources

In the light of ramification, a particular drawback of most narrative-based approaches including *HEC* becomes apparent: Since there is no central representation of state, checking the occurrence of situations in *NAT*1 and *NAT*2 over the complete narrative can be very expensive. When trying to demonstrate trigger conditions at various time-points, *HEC* computes prior occurrences again and again, because the intermediate results are not 'stored' in some shared data structure. Persistence caching by lemma generation (see Section 6) improves the situation, but is not a full replacement for a state representation.

A similar problem occurs when representing abstract resources as ordinary *HEC* fluents which is useful for reasoning about, e.g., stamina in determining soccer tactics. Any change of resources, such as by invoking a consuming process, must then be modelled in terms of terminating the old value and initiating the new value. Checking the (quantitative) persistence of resources is hence as expensive as the determination of the (qualitative) persistence of fluents.
This has been recognised as the so-called *inferential frame problem* [Thi99]. Unlike the traditional *representational frame problem*, the inferential frame problem is not about minimising the effort of axiomatisation, but about minimising the effort of making inferences with the chosen axiomatisation. These two issues do not necessarily correlate: Negation-as-failure, such as used in *HEC*, is a representationally elegant solution to formalise persistence, but inferentially quite expensive. An alternative proposed by [HS90] is to uniformly model change and persistence via an equational-logic approach [GHS+92]: In the *Fluent Calculus* (*FC*), situations are reified multi-sets of fluents using the associative and commutative multi-set constructor $\circ$ and a respective $AC1$-unification underlying $\doteq$:

$$(FC1) \quad \tilde{\forall} holds(F, S) \equiv \tilde{\exists} S \doteq F \circ Z$$

$$(FC2) \quad \begin{aligned} &\tilde{\forall} causes(S_1, P, S_2) \equiv \tilde{\exists} P \doteq nil \wedge S_1 \doteq S_2 \\ &\vee \tilde{\exists} P \doteq cons(A, P_2) \wedge terminates(A, S_1, S_3) \wedge initiates(A, S_1, S_4) \\ &\quad \wedge S_5 \circ S_3 \doteq S_1 \circ S_4 \wedge causes(S_5, P_2, S_2) \end{aligned}$$

Actually, $AC1$ treats fluents as resources that are consumed (the S_3 multi-set in $FC2$) by a terminating action and produced (the S_4 multi-set in $FC2$) by an initiating action. Else, they remain unaffected which requires no additional effort if employing modern, e.g., constraint-based unification algorithms. Regular fluents, i.e., the usual situation properties, are obtained as special cases of resources that only appear once in a multi-set (where $\dot{\varnothing}$ is the neutral to $\circ$):

$$\tilde{\forall} S \doteq F \circ F \circ Z \supset F \doteq \dot{\varnothing}$$

Recently, *FC* has been extended to represent narratives by introducing particular *happens* and *time* fluents into the state representations [Thi98]. We could also think of reifying states in *HEC*, such as in the following definition. Both are possible means for transferring ideas from narrative-based *HEC* to state-based *FC* and vice versa, possibly leading to an intermediate theory in the future.

$$\begin{aligned} &\tilde{\forall} holds(F \circ Z, T_1, T_2, C, B, E, L) \equiv \tilde{\exists} holds(F, T_1, T_2, C, B, E, L) \\ &\quad \wedge holds(Z, T_1, T_2, C, B, E, L) \end{aligned}$$

8.5 A Programming Language for InteRRaP-R Agents

In [Ros96], a programming language for the desig of InteRRaP agents ('ALaDIn') was presented based on the rationale given Müller [Mül96] which was a library of extendible Oz classes, but not necessarily a convenient AOP language in its own right. Due to the complex design, each InteRRaP module introduced classes of its own and

module interfaces required additional coding. This approach may be tolerable for customising uniform agents to particular domains, such as the programming of PRS agents in the dMARS language. It is certainly not feasible in the hybrid case.
The $\mathcal{A}$ language of [GL93] first promoted to develop domain specifications independently of underlying logics in special high-level languages.This idea has adopted for state-based Cognitive Robotics in GOLOG [LRL97] and generalised to narrative-based calculi by $\mathcal{E}$ [KM97a] and Reactive Pascal [Dav96]. However, these languages commonly lack constructs for the crucial control of the resulting (agent) inferences.
As a design that is based on the reconciliation of Agent Engineering with Cognitive Robotics, the domain representations of InteRRaP-R combine the concepts of action languages, such as $\mathcal{A}$, $\mathcal{E}$, GOLOG and Reactive Pascal, with additional control constructs, such as abstract resources and their parameterisation. These representations (currently written in Oz syntax) are moreover congruent across various modules and across the different layers of the agent model. Hence, it should be a straighforward next step to give them a neat surface syntax and possibly even a graphical notation which can be compiled into respective Oz code and which finally realise the Agent-Oriented Programming (AOP) of an advanced hybrid model.

A Auxiliary Definitions

The following are auxiliary definitions to the computational model of Section 3. They have been used with the standard `math0.zed`, `math1.zed`, and `mathoz.zed` files as the prelude for type-checking with the ZTC tool [Xia95]. They define booleans, declare several forward-referenced schemas, and introduce useful functions, such as the conversion of arbitrary sequences from sets (*anySequence*), various projection functions, and the summation of integers and floats (using the overloaded $\sum$ notation).

$\mathbb{B} == Bool$

$[Branch, Signal]$

$$toReal : \mathbb{Z} \to \mathbb{R}$$
$$max : \mathbb{P}\,\mathbb{R} \leftrightarrow \mathbb{R}$$

$[X]$

$$anySequence : \mathbb{P}\,X \leftrightarrow \text{seq}\ X \qquad \text{[build sequence from buffer]}$$

$$\forall\, y : \mathbb{P}\,X;\ x : X \bullet \exists\, s_1, s_2 : \text{seq}\,X \bullet x \in y \Leftrightarrow anySequence(y) = s_1 \frown \langle x \rangle \frown s_2$$

$[A, B]$

$$\pi_1^2 : A \times B \to A$$
$$\pi_2^2 : A \times B \to B$$

$[A, B, C]$

$$\pi_1^3 : A \times B \times C \to A$$
$$\pi_2^3 : A \times B \times C \to B$$
$$\pi_3^3 : A \times B \times C \to C$$

$[A, B, C, D]$

$$\pi_1^4 : A \times B \times C \times D \to A$$
$$\pi_2^4 : A \times B \times C \times D \to B$$
$$\pi_3^3 : A \times B \times C \times D \to C$$
$$\pi_4^4 : A \times B \times C \times D \to D$$

$$\sum : \mathbb{P}\,\mathbb{Z} \to \mathbb{Z} \qquad \text{[sum of set of integers]}$$
$$\sum : \mathbb{P}\,\mathbb{R} \to \mathbb{R} \qquad \text{[sum of set of floats]}$$

$$\forall\, pz : \mathbb{P}\,\mathbb{Z};\ z : \mathbb{Z} \bullet \sum(pz) = z \Leftrightarrow (z = 0 \land pz = \varnothing) \lor (\exists\, z_2 : pz \bullet z = (\sum(pz \setminus \{z_2\}) + z_2))$$
$$\forall\, pr : \mathbb{P}\,\mathbb{R};\ r : \mathbb{R} \bullet \sum(pr) = r \Leftrightarrow (r = 0.0 \land pr = \varnothing) \lor (\exists\, r_2 : pr \bullet r = (\sum(pr \setminus \{r_2\}) + r_2))$$

The following definition of *exhead*, *extail*, and *annotation* is to be inserted immediately before the $\Delta ProcessInf$ schema:

$$
\begin{array}{|l}
exhead, extail : Branch \rightarrow \text{seq}\, InferenceState \qquad \text{[access exceptions]} \\
annotation : Branch \leftrightarrow Signal \qquad \text{[access annotations]} \\
\hline
\forall\, b : Branch \bullet (b.exceptions = \langle\rangle \Rightarrow (exhead(b) = \langle\rangle \wedge extail(b) = \langle\rangle)) \wedge \\
\quad (b.exceptions \neq \langle\rangle) \Rightarrow (exhead(b) = \langle head(b.exceptions)\rangle \wedge \\
\qquad extail(b) = tail(b.exceptions)) \\
\forall\, b : Branch \bullet annotation(\!|\, \{b\}\, |\!) = \bigcup \{c : b.premises \cup b.conclusio \bullet c.annotation\}
\end{array}
$$

B Proofs of Some Propositions and Theorems

In the following, we present proofs and relevant lemmas for some of the propositions and theorems stated in Sections 4 and 5. It should be noted that we do not take a general theorem proving stance in which a possibly large semantical space is representatively captured by means of, e.g., abstract consistency properties [Fit90]. Rather, we look at *HEC* from a Logic Programming perspective that analyses the calculus in terms of logic procedures and finite data structures, i.e., the closed terms given by the Herbrand universe.

Proposition 4 (Existence of Infinite Sequences in *ECJ*) *Let M be a minimal three-valued (Herbrand) model for ECJ under I, Δ, DOM:*

$$M \models_3 Comp(ECJ1 \wedge \ldots \wedge ECJ4 \wedge ECS3 \wedge \Delta \wedge I \wedge DOM) \wedge CET \wedge ECK3 \wedge \ldots \wedge ECK5$$

For all C : Constant; $n : \mathbb{N}$, it holds $0.5 \in ran\, IP(C, n)$ if and only if there exists an infinite sequence $i : \mathbb{N}$; $U_{i,1}, \ldots, U_{i,6} : U$ such that $IP(holds, 5)(U_{i,1}, \ldots, U_{i,5}) = 0.5$, $IP(happens, 3)(U_{i,5}, U_{i,6}, U_{i,2}) = 1$, and $U_{i+1,3} = IF(cons, 2)(U_{i,5}, U_{i,3})$.

Proof. Because of the completion, it is assured that for any predicate not mentioned in axioms, facts, and constraints, the valuation under M must deliver 0. Furthermore, constraints are not part of the completed program in the theoremhood view of integrity, thus they cannot force undefined truth values provided the completed equivalence axioms by themselves are well-defined.
Thus, we first check that

$$0.5 \notin \text{ran}\, IP(\doteq, 2) \cup \text{ran}\, IP(member, 2) \cup \text{ran}\, IP(flip, 2) \cup \text{ran}\, IP(\dot{<}, 2) \cup \text{ran}\, IP(happens, 3) \cup \text{ran}\, IP(out, 3)$$

in order to concentrate the investigation of undefinedness upon *holds*, *clipped*, *initiates*, and *terminates*. The completed *happens* and $\dot{<}$ interpretations are either set to true in Δ or stay false because of their completion.
The well-definedness of $CET1 \wedge CET2 \wedge CET3$, thus $\doteq$, provided an infinite alphabet *Constant* follows from [Kun87, Mah88]. For *flip* now, the well-definedness is easy to see, as it can be only derived over the equivalence *ECJ*3 and *CET* does not allow for undefined $\doteq$ interpretations. Similarly, *out* is well-defined because of purely relying on $\dot{<}$ (*ECS*3).

For *member*, we have to argue by induction over the 'depth' of the first argument, that is, the number of recursive function applications which are needed to reach the entity from $IF(Constant, 0)$ — due to our model being isomorphic to the finite Herbrand trees, all entities can be reached this way in a bounded manner. For any depth of the argument U_1 such that $IP(member, 2)(U_1, U_2) = 0.5$, we could determine at least one equation $IP(\doteq, 2)(\tilde{U}_1, U_2) = 0.5$ where $\tilde{U}_1$ is an entity that is referred in the functional reconstruction of U_1. This contradicts with the well-definedness of $\doteq$.
Hence, we know that $IP(C, n)(U_1, \ldots, U_n) = 0.5$ implies that C is either *holds*, *initiates*, *terminates*, or *clipped*. Now suppose that either $IP(initiates, 6)(U_1, \ldots, U_6) = 0.5$ or $IP(terminates, 6)(U_1, \ldots, U_6) = 0.5$. By the completion, *DOM* is the only source to trace their definition by which there must be some $U_1' : U$ such that $IP(holds, 5)(U_1', U_3, \ldots, U_6) = 0.5$ (i).
Suppose now that $IP(clipped, 6)(U_1, \ldots, U_6) = 0.5$. Because of the *ECJ*2 definition, there must exist $U_1', \ldots, U_4' : U$ such that $IP(happens, 3)(U_1', U_2', U_3') = 1$, $U_4' = IF(cons, 2)(U_1', U_4)$, and $IP(terminates, 6)(U_2', U_1, U_3', U_4', U_5, U_1') = 0.5$. Because of (i), there must be $U_1'' : U$ such that $IP(holds, 5)(U_1'', U_3', U_4', U_5, U_1') = 0.5$. Thus, for each $IP(clipped, 6)(U_1, \ldots, U_6) = 0.5$, we find $U_1'', \ldots, U_6'' : U$ such that it holds $IP(holds, 5)(U_1'', \ldots, U_5'') = 0.5$, $IP(happens, 3)(U_5'', U_6'', U_2'') = 1$, and $U_3'' = IF(cons, 2)(U_5'', U_3)$ (ii).
Suppose now that $IP(holds, 5)(U_1, \ldots, U_5) = 0.5$. Because of the *ECJ*1 definition, there must exist $U_1', \ldots, U_5' : U$ such that $IP(happens, 3)(U_1', U_2', U_3') = 1$, $U_4' = IF(cons, 2)(U_1', U_4)$, and either $IP(initiates, 6)(U_2', U_1, U_3', U_4', U_4, U_1') = 0.5$ or $IP(clipped, 6)(U_1, U_3', U_2, U_3, U_5', U_1') = 0.5$.
Take the first case of $IP(initiates, 6)(U_2', U_1, U_3', U_4', U_4, U_1') = 0.5$. We know by (i) that there exists a $U_1'' : U$ such that $IP(holds, 5)(U_1'', U_3', U_4', U_4, U_1') = 0.5$. This means that we find $U_1'', \ldots, U_6'' : U$ such that it holds $IP(happens, 3)(U_5'', U_6'', U_2'') = 1$, $U_3'' = IF(cons, 2)(U_5'', U_3)$, and $IP(holds, 5)(U_1'', \ldots, U_5'') = 0.5$ (iii).
Take the second case of $IP(clipped, 6)(U_1, U_3', U_2, U_3, U_5', U_1') = 0.5$ and by (ii), there exists $U_1'', \ldots, U_6'' : U$ such that it holds $IP(holds, 5)(U_1'', \ldots, U_5'') = 0.5$, $IP(happens, 3)(U_5'', U_6'', U_2'') = 1$, and $U_3'' = IF(cons, 2)(U_5'', U_3)$ (iv).
(i), (ii), (iii), and (iv) demonstrate that for any $IP(C, n)(U_1, \ldots, U_n) = 0.5$, there exists some $U_1', \ldots, U_5' : U$ such that $IP(holds, 5)(U_1', \ldots, U_5') = 0.5$. These arguments combined show the proposition. □

Lemma 1 (Non-Members and Inequalities) *Let M be a first-order (Herbrand) model of ECJ4 $\wedge$ CET. Let $n : \mathbb{N}, U_{1,1}, U_{1,2}, \ldots, U_{n,1}, U_{n,2} : U$ such that for all $1 < i \leq n$, it holds $U_{i,2} = IF(cons, 2)(U_{i-1,1}, U_{i-1,2})$. For any $U' : U$ such that $IP(member, 2)(U, U_{n,2}) = 0$ it follows that $U \neq U_{i,1}$ for all $1 \leq i < n$.*

Proposition 5 (Three-Valued Minimal Models of *ECJ*) *Any minimal three-valued (Herbrand) model M of ECJ is already a two valued model of ECJ.*

Proof. By induction on the size $\#\Delta$, we show that M cannot assign 0.5 to any predicate. Suppose there exist $C : Constant, n : \mathbb{N}, U_1 \ldots, U_n : U$ such that $IP(C, n)(U_1, \ldots, U_n) = 0.5$, then by Proposition 4, there exists an infinite sequence $U_{i,1}, \ldots, U_{i,6} : U$ such that $IP(holds, 5)(U_{i,1}, \ldots, U_{i,5}) = 0.5$, $IF(cons, 2)(U_{i,5}, U_{i,3}) = U_{i+1,3}$, and $IP(happens, 3)(U_{i,5}, U_{i,6}, U_{i,2}) = 1.0$. Then, by the boundedness of Δ, there must be some $i \leq \#\Delta$ such that $U_{i,5} = U_{1,5}$.

Furthermore, since $U_{1,5}, U_{1,3}, \ldots, U_{i,5}, U_{i,3}$ satisfy the conditions of Lemma 1, it holds that $IP(member, 2)(U_{i,5}, U_{i,3}) = 1$, for otherwise $IP(\doteq, 2)(U_{i,5}, U_{1,5}) = 0$ contradicting with *CET*. Thus, by the second disjunct of the *ECJ*1 body evaluating to 0, we simplify $IP(holds, 5)(U_{i,1}, \ldots, U_{i,5}) = IP(\doteq, 2)(U_{i,4}, IF(\dot{1}, 0)) \in \{0, 1\}$ which contradicts the undefinedness of $IP(holds, 5)(U_{i,1}, \ldots, U_{i,5})$. □

Theorem 2 (Treatment of Worst Case in *ECJ*) *Let M be the minimal (Herbrand) model of ECJ:*

$$M \models \tilde{\forall} holds(F, T, C, \dot{0}, E) \supset holds(F, T, C, \dot{1}, E)$$

Proof. In the following, let $IF(\dot{0}, 0) = \tilde{0}$ and $IF(\dot{1}, 0) = \tilde{1}$.
We show below that provided there is a $IP(holds, 5)(U_1, U_2, U_3, \tilde{1}, U_4) = 0$ and $IP(holds, 5)(U_1, U_2, U_3, \tilde{0}, U_4) = 1$, then there would exist an infinite sequence $i : \mathbb{N}$; $U_{i,1}, \ldots, U_{i,5} : U$ such that $IP(cons, 2)(U_{i,4}, U_{i,3}) = U_{i+1,3}$, $IP(happens, 3)(U_{i,4}, U_{i,5}, U_{i,2}) = 1$, $IP(holds, 5)(U_{i,1}, U_{i,2}, U_{i,3}, \tilde{1}, U_{i,4}) = 0$ and $IP(holds, 5)(U_{i,1}, U_{i,2}, U_{i,3}, \tilde{0}, U_{i,4}) = 1$. Because of $\#\Delta$ being bounded, there must be some $k \leq \#\Delta$ such that $IP(Member, 2)(U_{i,4}, U_{i,3}) = 1$. Because of *ECJ*1, it holds then $IP(holds, 5)(U_{i,1}, U_{i,2}, U_{i,3}, \tilde{1}, U_{i,4}) = 1$ and $IP(holds, 5)(U_{i,1}, U_{i,2}, U_{i,3}, \tilde{0}, U_{i,4}) = 0$ which derives the desired contradiction.
Now suppose $IP(initiates, 6)(U_1, U_2, U_3, U_4, \tilde{1}, U_5) = 0$ and $IP(initiates, 6)(U_1, U_2, U_3, U_4, \tilde{0}, U_5) = 1$. By the latter expression and *DOMCAU*, there must me some $U_1' : U$ such that $IP(holds, 5)(U_1', U_3, U_4, \tilde{1}, U_5) = 0$. Because of the first expression, it must hold at the same time $IP(holds, 5)(U_1', U_3, U_4, \tilde{0}, U_5) = 1$ (i). The same can be stated for *terminates* (ii).
Now suppose $IP(clipped, 6)(U_1, U_2, U_3, U_4, \tilde{1}, U_5) = 0$ and $IP(clipped, 6)(U_1, U_2, U_3, U_4, \tilde{0}, U_5) = 1$. By the latter, we know that there exist $U_1', \ldots, U_4' : U$ such that $U_4' = IF(Cons, 2)(U_5, U_4)$, $IP(happens, 3)(U_1', U_2', U_3') = 1$, and $IP(terminates, 6)(U_2', U_1, U_3', U_4', \tilde{1}, U_1') = 0$. Furthermore, by the first statement, its must hold that $IP(terminates, 6)(U_2', U_1, U_3', U_4', \tilde{0}, U_1') = 1$.
By (ii), there is $U_1'' : U$ such that $IP(holds, 5)(U_1'', U_3', U_4', \tilde{1}, U_1') = 0$ and $IP(holds, 5)(U_1'', U_3', U_4', \tilde{0}, U_1') = 1$. Hence, we find $U_1''', \ldots, U_4''' : U$ such that it holds $U_3''' = IF(Cons, 2)(U_5, U_4)$, $IP(happens, 3)(U_4''', U_5''', U_2''') = 1$, $IP(holds, 5)(U_1''', U_2''', U_3''', \tilde{1}, U_4''') = 0$ and $IP(holds, 5)(U_1''', U_2''', U_3''', \tilde{0}, U_4''') = 1$ (iii).
Now suppose $IP(holds, 5)(U_1, U_2, U_3, \tilde{1}, U_4) = 0$ and $IP(holds, 5)(U_1, U_2, U_3, \tilde{0}, U_4) = 1$. By the latter, we know that there exist $U_1', \ldots, U_4' : U$ such that $U_4' = IF(Cons, 2)(U_3, U_4)$, $IP(happens, 3)(U_1', U_2', U_3') = 1$, $IP(initiates, 6)(U_2', U_1, U_3', U_4', \tilde{0}, U_1') = 1$, and furthermore $IP(clipped, 6)(U_1, U_3', U_2, U_3, \tilde{1}, U_4) = 0$. Because of the first, we know that either $IP(initiates, 6)(U_2', U_1, U_3', U_4', \tilde{1}, U_1') = 0$ or $IP(clipped, 6)(U_1, U_3', U_2, U_3, \tilde{0}, U_4) = 1$.
Take the first case, then by (i), there exists a $U_1'' : U$ such that $IP(holds, 5)(U_1'', U_3', U_4', \tilde{0}, U_1') = 1$ and $IP(holds, 5)(U_1'', U_3', U_4', \tilde{1}, U_1') = 0$. Comprising the arguments, we got $U_1''', \ldots, U_5''' : U$ such that $U_3''' = IF(Cons, 2)(U_3, U_4)$, $IP(happens, 3)(U_4', U_5', U_2') = 1$, $IP(holds, 5)(U_1''', U_2''', U_3''', \tilde{1}, U_4''') = 0$ and $IP(holds, 5)(U_1''', U_2''', U_3''', \tilde{0}, U_4''') = 1$ (iv).

Take the second case, then by (iii), there exist $U''_1, \ldots, U''_5 : U$ such that $U''_3 = IF(Cons, 2)(U_4, U_3)$, $IP(happens, 3)(U''_4, U''_5, U''_2) = 1$, $IP(holds, 5)(U''_1, U''_2, U''_3, \tilde{1}, U''_4) = 0$ and $IP(holds, 5)(U''_1, U''_2, U''_3, \tilde{0}, U''_4) = 1$.
Thus, we have $U'''_1, \ldots, U'''_5$ such that $U'''_3 = IF(Cons, 2)(U_4, U_3)$, $IP(happens, 3)(U'''_4, U'''_5, U'''_2) = 1$, $IP(holds, 5)(U'''_1, U'''_2, U'''_3, \tilde{1}, U'''_4) = 0$, and $IP(holds, 5)(U'''_1, U'''_2, U'''_3, \tilde{0}, U'''_4) = 1$ (v).
Combining the cases (i) – (v) derives the contradictory infinite sequence. □

Proposition 6 (Existence of Infinite Sequences in *HEC*) *Let M be a minimal three-valued (Herbrand) model for HEC under I, Δ, DOM:*

$$M \models_3 Comp(HEC1 \wedge \ldots \wedge HEC5 \wedge ECJ3 \wedge ECJ4 \wedge \Delta \wedge I \wedge DOM) \wedge CET \wedge ECK3 \wedge ECK4 \wedge HEC6 \wedge \ldots \wedge HEC12$$

For all C : Constant; $n : \mathbb{N}$, it holds $0.5 \in ran\, IP(C, n)$ if and only if there exists an infinite sequence $i : \mathbb{N}$; $U_{i,1}, \ldots, U_{i,8} : U$ such that $IP(holds, 7)(U_{i,1}, \ldots, U_{i,7}) = 0.5$, $U_{i+1,4} = IF(cons, 2)(U_{i,6}, U_{i,4})$, and $IP(happens, 5)(U_{i,6}, U_{i,8}, U_{i,2}, U_{i,3}, U_{i,7}) = 1.0$.

Proof. Similar to the proof of Proposition 4, we know that the predicates not mentioned in the calculus and the basic predicates $\doteq$, *member*, *happens*, $\dot{<}$ are well-defined. We extend this straightforwardly to *initially*, *dual*, *disjoint*, *decomposeInitially*, *decomposeHolds*, and *decomposeMacro* by checking I, *DOMSAB*, *DOMAAB*, *HEC*3 and *HEC*4. Thus, we can focus our analysis on *holds*, *clipped*, *iclipped*, and *causes*.
Suppose we are given $IP(causes, 8)(U_1, \ldots, U_8) = 0.5$, then by *DOMCAU* there must exist U'_1 such that $IP(holds, 7)(U'_1, U_3, \ldots, U_8) = 0.5$ (i).
Now suppose that $IP(clipped, 7)(U_1, \ldots, U_7) = 0.5$. Then we know by *HEC*2 that there exists $U'_1, \ldots, U'_6 : U$ such that $IF(cons, 2)(U_6, U_4) = U'_5$, $IP(happens, 5)(U'_1, \ldots, U'_4, U_7) = 1$ (if it was 0, the *clipped* expression would evaluate to 0) and such that $IP(causes, 8)(U'_2, U'_6, U'_3, U'_4, U'_5, U_5, U'_1, U_7) = 0.5$. By (i), there is $U''_1 : U$ such that $IP(holds, 7)(U''_1, U'_3, U'_4, U'_5, U_5, U'_1, U_7) = 0.5$. Hence we know that there exist $U'''_1, \ldots, U'''_6 : U$ such that $U'''_5 = IF(cons, 2)(U_6, U_4)$, $IP(happens, 5)(U'''_1, \ldots, U'''_4, U_7) = 1$ and $IP(holds, 7)(U'''_6, U'''_3, U'''_4, U'''_5, U_5, U''_1, U_7)$ (ii).
Now suppose that $IP(iclipped, 6)(U_1, \ldots, U_6) = 0.5$. Then we know by *HEC*5 that there exist $U'_1, \ldots, U'_6 : U$ such that $IF(cons, 2)(U_5, U_3) = U'_5$, $IP(happens, 5)(U'_1, \ldots, U'_4, U_6) = 1$, and $IP(causes, 8)(U'_2, U'_6, U'_3, U'_4, U'_5, U_4, U'_1, U_6) = 0.5$. By (i), there is a $U''_1 : U$ such that $IP(holds, 7)(U''_1, U'_3, U'_4, U'_5, U_4, U'_1, U_6) = 0.5$. Hence, there exist $U'''_1, \ldots, U'''_6 : U$ such that $U'''_5 = IF(cons, 2)(U_5, U_3)$, $IP(happens, 5)(U'''_1, \ldots, U'''_4, U_6) = 1$ and furthermore $IP(holds, 7)(U'''_6, U'''_3, U'''_4, U'''_5, U_4, U''_1, U_6)$ (iii).
Now suppose that we are given $IP(holds, 7)(U_1, \ldots, U_7) = 0.5$. Then we know that either the third or the second disjunct in *HEC*1 must be responsible for undefinedness. Suppose the second disjunct to be undefined, then there is a $U'_1 : U$ such that $IP(iclipped, 6)(U_1, U_3, U_4, U'_1, U_6, U_2) = 0.5$. By (iii), there are $U''_1, \ldots, U''_6 : U$ such that $U''_5 = IF(cons, 2)(U_6, U_4)$, $IP(happens, 5)(U''_1, \ldots, U''_4, U_2) = 1$, and furthermore $IP(holds, 7)(U''_6, U''_3, U''_4, U''_5, U'_1, U''_1, U_2)$. This means that for any $IP(holds, 7)(U_1, \ldots, U_7) = 0.5$, we find $U'''_1, \ldots, U'''_8 : U$ such that

$U'''_4 = IF(cons, 2)(U_6, U_4)$, $IP(holds, 7)(U'''_1, \ldots, U'''_7) = 0.5$, and
$IP(happens, 5)(U'''_6, U'''_8, U'''_2, U'''_3, U'''_7) = 1$ (iv).
Suppose the third disjunct to be undefined, then there is a $U'_1, U'_2 : U$ such that $IP(clipped, 7)(U_1, U'_1, U_3, U_4, U'_2, U_6, U_2) = 0.5$. We know by (ii) that there exist $U''_1, \ldots, U''_6 : U$ such that $IP(happens, 5)(U''_1, \ldots, U''_4, U_2) = 1$ and such that $IP(holds, 7)(U''_6, U''_3, U''_4, U''_5, U_5, U''_1, U_2)$. Hence, there are $U'''_1, \ldots, U'''_8 : U$ such that $U'''_4 = IF(cons, 2)(U_6, U_4)$, $IP(holds, 7)(U'''_1, \ldots, U'''_7) = 0.5$, and $IP(happens, 5)(U'''_6, U'''_8, U'''_2, U'''_3, U'''_7) = 1$. (v)
(i) – (v) furthermore show that for any $IP(C, n)(U_1, \ldots, U_n) = 0.5$, there exists some $U'_1, \ldots, U'_7 : U$ such that $IP(holds, 7)(U'_1, \ldots, U'_7) = 0.5$. These arguments combined show the proposition. □

Proposition 7 (Three-Valued Minimal Models of *HEC*) *Any minimal three-valued (Herbrand) model M of HEC is already a two valued model.*

Proof. By induction over $\#\Delta$ and $\#DOM$. Suppose $IP(C, n)(U_1, \ldots, U_n) = 0.5$, then by Proposition 6, there exists an infinite sequence $i : \mathbb{N}$; $U_{i,1}, \ldots, U_{i,8} : U$ such that $IP(holds, 7)(U_{i,1}, \ldots, U_{i,7}) = 0.5$, $U_{i+1,4} = IF(cons, 2)(U_{i,6}, U_{i,4})$, and either $IP(happens, 5)(U_{i,6}, U_{i,8}, U_{i,2}, U_{i,3}, U_{i,7}) = 1.0$.
Similar to the proof of Proposition 5 and using Proposition 1, there must be some place $k \leq \#\Delta$ such that the referred event $U_{k,6}$ is already member of the causal chain $U_{k,4}$, thus by *HEC*1, the *holds* expression must have been assigned a value from $\{0, 1\}$ from which we derive a contradiction. □

Theorem 5 (Treatment of Dual Fluents) *Let M be the minimal (Herbrand) model of HEC:*

$$M \models \tilde{\forall} holds(F, T_1, T_2, C, \dot{0}, E, L) \wedge Dual(F, F_-) \supset \neg holds(F_-, T_1, T_2, C, \dot{0}, E, L)$$

Proof. Suppose $IP(dual, 2)(U_1, U'_1) = 1$, $IP(holds, 7)(U_1, U_2, U_3, U_4, \tilde{0}, U_5, U_6) = 1$, and $IP(holds, 7)(U'_1, U_2, U_3, U_4, \tilde{0}, U_5, U_6) = 1$. By *HEC*3, it holds $IP(dual, 2)(U'_1, U_1) = 1$ such that we can exchange U_1 and U'_1 without loss of generalisation. By *HEC*1, we furthermore know that $IP(\dot{<}, 2)(U_2, U_3) = 1$ or $U_2 = U_3$.
By *HEC*1, there must be either $IP(initially, 2)(U_1, U_6) = 1$ and $IP(iclipped, 6)(U_1, U_3, U_4, \tilde{1}, U_5, U_6) = 0$ (a) or there exist $U_7, \ldots, U_{10} : U$ such that $IP(happens, 5)(U_7, U_8, U_9, U_{10}, U_6) = 1$, $IP(\dot{<}, 2)(U_{10}, U_2) = 1$, $IP(causes, 8)(U_8, U_1, U_9, U_{10}, IF(Cons, 2)(U_5, U_4), \tilde{0}, U_7, U_6) = 1$, and $IP(clipped, 7)(U_1, U_9, U_3, U_4, \tilde{1}, U_5, U_6) = 0$ (b). In case (b), U_7 must be furthermore different from U_5 because of *HEC*11. Because of Theorem 4, we furthermore know that $IP(causes, 8)(U_8, U_1, U_9, U_{10}, IF(Cons, 2)(U_5, U_4), \tilde{1}, U_7, U_6) = 1$.
By *HEC*1, there must be furthermore either $IP(initially, 2)(U'_1, U_6) = 1$ and $IP(iclipped, 6)(U'_1, U_3, U_4, \tilde{1}, U_5, U_6) = 0$ (a) or there exist $U'_7, \ldots, U'_{10} : U$ such that $IP(happens, 5)(U'_7, U'_8, U'_9, U'_{10}, U_6) = 1$, $IP(\dot{<}, 2)(U'_{10}, U_2) = 1$, $IP(causes, 8)(U'_8, U'_1, U'_9, U'_{10}, IF(Cons, 2)(U_5, U_4), \tilde{0}, U'_7, U_6) = 1$, and

$IP(clipped, 7)(U'_1, U'_9, U_3, U_4, \tilde{1}, U_5, U_6) = 0$ (b). In case (b), U'_7 must be furthermore different from U_5. Because of Theorem 4, we furthermore know that $IP(causes, 8)(U'_8, U'_1, U'_9, U'_{10}, IF(Cons, 2)(U_5, U_4), \tilde{1}, U'_7, U_6) = 1$.
Now let us consider case (a,a'). We then have $IP(initially, 2)(U'_1, U_6) = 1$, $IP(initially, 2)(U_1, U_6) = 1$ and $IP(dual, 2)(U_1, U'_1) = 1$ which is inconsistent ($HEC12$).
Now let us consider case (a,b'). Because of (b'), we can state that $IP(happens, 5)(U'_7, U'_8, U'_9, U'_{10}, U_6) = 1$, $IP(\doteq, 2)(U'_7, U_5) = 0$, $IP(dual, 2)(U_1, U'_1) = 1$, and $IP(causes, 8)(U'_8, U'_1, U'_9, U'_{10}, IF(Cons, 2)(U_5, U_4), \tilde{1}, U'_7, U_6) = 1$. Moreover, we know that $IP(\dot{<}, 2)(U_3, U'_9) = 0$, because otherwise we would get $IP(\dot{<}, 2)(U'_{10}, U_2) = 0$ (by $HEC10$, $ECK3$, and $ECK4$). Thus, the right hand-side of $HEC5$ is satisfied for which we derive $IP(iclipped, 6)(U_1, U_3, U_4, \tilde{1}, U_5, U_6) = 1$ (contradiction).
Now let us consider case (b,b') and let us first suppose that $IP(\dot{<}, 2)(U'_{10}, U_9) = 0$. Then, because of (b'), we state that it holds $IP(happens, 5)(U'_7, U'_8, U'_9, U'_{10}, U_6) = 1$, $IP(\doteq, 2)(U'_7, U_5) = 0$, $IP(dual, 2)(U_1, U'_1) = 1$, and $IP(causes, 8)(U'_8, U'_1, U'_9, U'_{10}, IF(Cons, 2)(U_5, U_4), \tilde{1}, U'_7, U_6) = 1$. Moreover, we know that $IP(\dot{<}, 2)(U_3, U'_9) = 0$, because otherwise we would get $IP(\dot{<}, 2)(U'_{10}, U_2) = 0$ (by $HEC10$, $ECK3$, and $ECK4$). Then we can conclude by $HEC4$ that $IP(disjoint, 4)(U_2, U_3, U'_9, U'_{10}) = 0$. Since the right-hand side of $HEC2$ is satisfied, we then derive $IP(clipped, 7)(U_1, U_9, U_3, U_4, \tilde{1}, U_5, U_6) = 0$ (contradiction).
Now let us finally consider case (b,b') and let us suppose $IP(\dot{<}, 2)(U'_{10}, U_9) = 1$. By $HEC10$, $ECK3$, and $ECK4$, we derive that it must hold $IP(\dot{<}, 2)(U'_9, U_{10}) = 0$. Moreover, because of (b), we can state that $IP(happens, 5)(U_7, U_8, U_9, U_{10}, U_6) = 1$, $IP(\doteq, 2)(U_7, U_5) = 0$, $IP(dual, 2)(U'_1, U_1) = 1$, and $IP(causes, 8)(U_8, U_1, U_9, U_{10}, IF(Cons, 2)(U_5, U_4), \tilde{1}, U_7, U_6) = 1$. Moreover, it holds that $IP(\dot{<}, 2)(U_3, U_9) = 0$, because otherwise we would get $IP(\dot{<}, 2)(U_{10}, U_2) = 0$ (by $HEC10$, $ECK3$, and $ECK4$). Then we can conclude by $HEC4$ that $IP(disjoint, 4)(U'_9, U_3, U_9, U_{10}) = 0$. Since the right-hand side of $HEC2$ is satisfied, we derive $IP(clipped, 7)(U'_1, U'_9, U_3, U_4, \tilde{1}, U_5, U_6) = 0$ (contradiction). □

Lemma 2 (König's Lemma) *Every infinite tree that is finitely branching must have an infinite branch.*

Proposition 10 (Termination) *Let $\mathcal{C} : \Delta \to \mathbb{N}^\infty$ be a strictly monotonic cost function on abducibles, i.e., $\Delta_1 \models \Delta_2$ only if $\mathcal{C}(\Delta_1) \geq \mathcal{C}(\Delta_2)$ and $\Delta_1 \models\!\!\models \Delta_2$ only if $\mathcal{C}(\Delta_1) = \mathcal{C}(\Delta_2)$. Then for any I; DOM; G, any search rule that never select disjuncts such that $\mathcal{C}(\mathcal{H}(F)) > k \in \mathbb{N}$ and any least-commitment computation rule that delays UNF until all other steps including the unfolding of member literals are done, $I \wedge DOM \vdash_{PIHEC} G$ terminates.*

Proof. We show that, supposed $I \wedge DOM \vdash_{PIHEC} G$ does not terminate, then there must be an infinite sequence of subsequently generated disjuncts each including a *member* literal such that the first argument is determined and the terms in both of its arguments are also referred to in the residue of the respective disjunct. The first arguments of

the *member* literals are building up an incremental causal chain and these literals must have been rewritten to $\bot$.

For any state of the proof, each non-failure disjunct must have a finite set of constraints which have been generated from the initially finite set of constraints in *PIHEC*. By the bound restriction in selection and the definition of $\mathcal{C}$, each non-failure disjunct must furthermore have a finite residue including a finite set of referred time-points, too. Because of Proposition 8, we know that the referred terms in such a residue must be also present (modulo substitution) in the successor residues.

Under these conditions, the combination of subsequent SMP, EQU, FCT, CAS, FIR, and PRP steps terminates. The crucial observation is that, due to the definition of *PIHEC*, 'firing' of constraints (releasing a disjunct in the conclusion with variables disjoint to the precondition), although possibly increasing the abducibles and propagating into other constraints, diminishes the overall number of constraints that are able to fire (where we reasonably expect PRP not to propagate the same information twice). This is especially due to the constraints derived from $\neg$*clipped* goals not being able to 'invent' new time-points — the referred time-points have to be propagated from the residue. Hence, in the worst case, a finite partial order is converted into a total order which must terminate.

Hence, G must contain the original sources of an infinite proof due to unfolding. Since the initial $C = nil$ arguments are determined and because of least-commitment, the respective C arguments of intermediate sub-goals will always refer to terms also appearing in the residue. In the following, we can regard the application of one UNF step followed by exhaustive applications (finitely many) of {SMP, EQU, FCT, CAS, FIR, and PRP} and of unfolding *member* (terminates because of the determinedness of C) as just a single 'least-commitment step' of IFF.

Since unfolding *disjoint*, *dual*, *initially*, *decomposeInitially*, and *decomposeFluent* does terminate, we can similarly to the proof of Proposition 6 trace the unfolding of the crucial predicates *holds*, *causes*, *clipped*, *iclipped* in a tree of subsequent sets of *holds* literals starting with the finite source goals in G. Each node of that tree corresponds to a disjunct appearing during the proof which contains all *holds* literals of that node. All successors to the node correspond to the successor disjuncts in the proof which have been generated from their father using one least-commitment step.

A successor node must replace one of the *holds* literals by a number of other *holds* predicates which enlarge the causal list of their father literal. Because of the third disjunct in *HEC*1 and the least-commitment strategy, there exists for each successor a corresponding *member* literal which has the causal list as argument and which must have been rewritten to $\bot$.

This tree is finitely branching due to the definition of *HEC* and the finiteness of the intermediate residues. Each branch of the tree builds several causal chains only referring to entities in the respective residue and at least one of the causal chains is incremented in each successor. Because of the non-termination of the proof procedure, the tree must be furthermore infinite.

Because of Proposition 2, we know that one branch in this tree must be infinite. Since each of the nodes in this branch has a successor, the associated disjunct must have been selected during the proof and therefore all the associated residues have a cost of less than k.

Within this infinite branch, we can construct again an infinite tree which describes the parallel extensions of the initial causal chains (*nil*) within that branch. This tree must be finitely branching, too, because of the finiteness of residues during the proof. Furthermore, this tree must be infinite, for otherwise we could show that there is a place in the selected infinite branch in which no causal chain would be extended.
Because of Proposition 2, again, there must be an infinite branch in this tree which builds up an infinite causal chain associated with *member* literals carrying the intermediate chains as their first argument and purely referring to entities from the residue. These *member* literals must have been rewritten to $\bot$.
By the overall bound of the intermediate residues, we know that there must be some place in this branch such that a referred entity in *member* is already part of the causal list in which case *member* must have been rewritten to $\top$, contradiction. □

References

[AC87] P. E. Agre and D. Chapman. Pengi: an Implementation of a Theory of Activity. In *Proc. of AAAI-87*, pages 268–272. Morgan Kaufmann, 1987.

[AG96] K. Arnold and J. Gosling. *The Java Programming Language.* The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

[All84] J. F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23:123–154, 1984.

[And93] J. R. Anderson. *Rules of the mind.* Lawrence Erlbaum Associates, Hillsdale, NJ, 1993.

[Asa99] M. Asada, editor. *RoboCup-98: Robot World Cup II*, volume 1604 of *Lecture Notes in Artificial Intelligence.* 1999. to appear.

[AZ87] S. T. Allworth and R. N. Zobel. *Introduction to Real-Time Software Design.* Macmillan, 1987.

[BD94] M. Boddy and T. L. Dean. Deliberation scheduling for problem solving in time-constrained environments. *Artificial Intelligence*, 1(67):245–285, 1994.

[BF95] A. L. Blum and M. L. Furst. Fast planning through planning graph analysis. In C. S. Mellish, editor, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 166–1642, Montreal, Canada, August 1995. Morgan Kaufmann.

[BFV98] H.-J. Bürckert, K. Fischer, and G. Vierke. Transportation scheduling with holonic mas – the teletruck approach. In *Proceedings of the Third International Conference on Practical Applications of Intelligent Agents and Multiagents (PAAM'98)*, 1998.

[BGLM92] A. Bossi, M. Gabbrielli, G. Levi, and M. C. Meo. Contributions to the semantics of open logic programs. In *Proc. of the International Conference on Fifth Generation Computer Systems*, pages 570–580. ICOT, 1992.

[BHL95] F. Bacchus, J. Y. Halpern, and H. J. Levesque. Reasoning about noisy sensors in the Situation Calculus. In *Proc. IJCAI'95*, pages 1933–1940, 1995.

[BHW98] H. D. Burkhard, M. Hannebauer, and J. Wendler. AT humboldt — development, practice and theory. In *RoboCup-97: Robot Soccer World Cup I*, volume 1395 of *Lecture Notes in Artificial Intelligence*, pages 357–372. Springer, 1998.

[BIP87] M. E. Bratman, D. J. Israel, and M. E. Pollack. Toward an architecture for resource-bounded agents. Technical report, Center for the Study of Language and Information, SRI and Stanford, 1987.

[BKMS95] R. P. Bonasso, D. Kortenkamp, D. Miller, and M. Slack. Experiments with an architecture for intelligent, reactive agents. In *Intelligent Agents II*, Lecture Notes in Artificial Intelligence. Springer, 1995.

[BLR92] J. Bates, A. B. Loyall, and W. S. Reilly. Broad agents. *SIGART Bulletin*, 2(4), August 1992.

[BLSH95] B. Brunner, K. Landzettel, B.-M. Steinmetz, and G. Hirzinger. Telesensor-programming: A task-directed programming approach for sensor-based space robots. In *Proc. of the International Conference on Advanced Robotics*, 1995.

[Boh97] T. Bohnenberger. Eine Deduktive Wissensbasis für die Agentenarchitektur InteRRaP. Master's thesis, Universität des Saarlandes, Saarbrücken, 1997.

[Boo91] G. Booch. *Object-Oriented Design with Applications.* Benjamin cummings, Menlo Park CA, 1991.

[Bow87] C. M. Bowling, editor. *Principles and Elements of Thought Construction, Artificial Intelligence, and Cognitive Robotics.* Csy Pub, 1987.

[Bro86] R. A. Brooks. A robust layered control system for a mobile robot. In *IEEE Journal of Robotics and Automation*, volume RA-2 (1), pages 14–23, April 1986.

[Bro91] R. A. Brooks. Intelligence without reason. Technical Report 1293, MIT AI Laboratory, April 1991.

[BS85] R. Brachman and J. Schmolze. An overview of the kl-one knowledge representation system. *Cognitive Science*, 9(2):171–216, 1985.

[BS95] R. Backofen and G. Smolka. A complete and recursive feature theory. *Theoretical Computer Science*, 146(1–2):243–268, July 1995.

[BT94] S. Bornscheuer and M. Thielscher. Representing concurrent actions and solving conflicts. In B. Nebel and L. Dreschler-Fischer, editors, *Proceedings of the German Annual Conference on Artificial Intelligence (KI)*, volume 861 of *LNAI*, pages 16–27, Saarbrücken, Germany, September 1994. Springer.

[BT97] S. Bornscheuer and M. Thielscher. Explicit and implicit indeterminism: Reasoning about uncertain and contradictory specifications of dynamic systems. *Journal of Logic Programming*, 31(1–3):119–155, 1997.

[BW96] A. Burns and A. Wellings. *Real-Time Systems and Their Programming Languages (2nd edition).* Addison-Wesley, 1996.

[CDT91] L. Console, D. Theseider Dupre, and P. Torasso. On the relationship between abduction and deduction. *Journal of Logic and Computation*, 2(5):661–690, 1991.

[CG86] K. L. Clark and S. Gregory. Parlog: Parallel programming in logic. *TOPLAS*, 8(1):1–49, 1986.

[CL89] L. Cavedon and J. W. Lloyd. A completeness theorem for sldnf resolution. *Journal of Logic Programming*, 7:177–191, 1989.

[CL90] P. R. Cohen and H. J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42(3):213–261, 1990.

[CL94] S. Costantini and G. A. Lanzarone. A meta-logic programming language. *International Journal of Experimental and Theoretical Artificial Intelligence*, 6:239–287, 1994.

[CL95] P. R. Cohen and H. J. Levesque. Communicative actions for artificial agents. In V. Lesser, editor, *Proc. of the 1st International Conference on Multiagent Systems*, pages 65–72. AAAI Press, 1995.

[Cla78] K. L. Clark. Negation as Failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum press, New York, 1978.

[CLL93] N. Carver, V. Lesser, and Q. Long. Resolving global inconsistency in distributed sensor interpretation: Modeling agent interpretations in DRESUN. In *Proceedings of the 12th International Workshop on Distributed Artificial Intelligence*, pages 19–33, Hidden Valley, Pennsylvania, May 1993.

[CM87] W. F. Clocksin and C. S. Mellish. *Programming in Prolog (3rd edition)*. Springer Verlag, 1987.

[Col85] A. Colmerauer. Prolog in 10 figures. *Communications of the ACM*, 28(12):1296–1310, 1985.

[CP79] P. R. Cohen and C. R. Perrault. Elements of a plan-based theory of speech acts. *Cognitive Science*, 3(3):177–212, 1979.

[CR79] E. Chang and R. Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Communications of the ACM*, 22:281–283, 1979.

[CY79] L. L. Constantin and E. Yourdon. *Structured Design.* Prentice-Hall, Englewood Cliffs, NY, 1979.

[Dab93] V. G. Dabija. *Deciding Whether to Plan to React.* PhD thesis, Stanford University, Department of Computer Science, December 1993.

[Dav96] J. Davila. Reactive Pascal and the Event Calculus. In U. Siegmund and M. Thielscher, editors, *Proc. of the FAPR'96 Workshop on Reasoning about Actions and Planning in Complex Environments*, volume 11 of *Technical Report AIDA*, 1996.

[DGKK98] P. Doherty, J. Gustafsson, L. Karlsson, and J. Kvarnström. TAL: Temporal action logics language specification and tutorial. *Linköping Electronic Articles in Computer and Information Science*, 3(15), 1998. URL: `http://www.ep.liu.se/ea/cis/1998/015/`.

[DHS+88] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language chip. In *International Conference on FGCS*, Tokyo, November 1988.

[dKLW98] M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A Formal Specification of dMars. In *Intelligent Agents IV*, volume 1365 of *Lecture Notes in Artificial Intelligence*, pages 155–174. Springer, 1998.

[DKT94] B. Dunin-Keplicz and J. Treur. Compositional formal specification of multi-agent systems. In *Intelligent Agents*, volume 890 of *Lecture Notes in Artificial Intelligence*, pages 102–117. Springer, 1994.

[DL89] E. H. Durfee and V. R. Lesser. Negotiating task decomposition and allocation using partial global planning. In *Distributed Artificial Intelligence, Volume II*, pages 229–244, San Mateo, CA, 1989. Morgan Kaufmann Publishers, Inc.

[dL98] M. d'Inverno and M. Luck. Engineering agentspeak(l): A formal computational model. *Journal of Logic and Computation*, 8(3), 1998.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[DS92] M. Denecker and D. De Schreye. Sldnfa: An abductive procedure for normal abductive programs. In *Proc. of the International Conference and Symposion on Logic Programming*, pages 686–700, 1992.

[DST99] P. Dell'Acqua, F. Sadri, and F. Toni. *Combining Introspection and Communication with Rationality and Reactivity in Agents*. Lecture Notes in Artificial Intelligence. Springer, 1999. to appear.

[DvLV96] L. Dorst, M. van Lambalgen, and F. Vorbraak, editors. *Reasoning with Uncertainty in Robotics*, volume 1093 of *Lecture Notes in Artificial Intelligence*. Springer, 1996.

[DW88] H. F. Durrant-Whyte. *Integration, Coordination, and Control of Multi-Sensor Robot Systems*. Kluwer Academic Publishers, 1988.

[EHN94] K. Erol, J. Hendler, and D. Nau. Htn planning: complexity and expressivity. In *Proc. of the 12th National Conference on Artifical Intelligence (AAAI-94)*, volume 2, Seattle, Washington, 1994. AAAI Press.

[EHT96] K. Eder, S. Hölldobler, and M. Thielscher. An abstract machine for reasoning about situations, actions, and causality. In R. Dyckhoff, H. Herre, and P. Schroeder-Heister, editors, *Proceedings of the International Workshop on Extensions of Logic Programming (ELP)*, volume 1050 of *LNAI*, pages 137–151, Leipzig, Germany, March 1996. Springer.

[EM88] R. Engelmore and T. Morgan, editors. *Blackboard Systems.* Addison-Wesly, 1988.

[EM91] C. Elsaesser and R. MacMillan. Representation and algorithms for multi-agent adversarial planning. Technical report, The MITRE Corporation, 1991.

[Esh88] K. Eshghi. Abductive planning with event calculus. In *Proc. of the Fifth International Conference on Logic Programming*, pages 562–578, 1988.

[Etz93] O. Etzioni. Intelligence without robots (a reply to brooks). *AI Magazine*, December, 1993.

[fAIS99] Institut für Autonome Intelligente Systeme. Kurt — an experimental robot platform for sewerage inspection, 1999. URL: `http://ais.gmd.de/BAR/KURTII.htm`

[FDC87] N. S. Flann, T. G. Dietterich, and D. R. Corpron. Forward-chaining logic programming with the atms. In *Proc. of AAAI'87*, pages 24–29, 1987.

[Fer92] I. A. Ferguson. *TouringMachines: An Architecture for Dynamic, Rational, Mobile Agents.* PhD thesis, Computer Laboratory, University of Cambridge, UK,, 1992.

[Fir92] R. J. Firby. Building symbolic primitives with continuous control routines. In *Proc. of the 1st International Conference on Artifical Intelligence Planning Systems*, 1992.

[Fis93a] K. Fischer. The rule-based multi-agent system magsy. In *Proceedings of the CKBS'92 Workshop.* DAKE Centre, Keele University, 1993.

[Fis93b] M. Fisher. Concurrent metatem — a language for modeling reactive systems. In *Parallel Architectures and Languages*, volume 694 of *Lecture Notes in Computer Science.* Springer-Verlag, 1993.

[Fit90] M. Fitting. *First Order Logic and Automated Theorem Proving.* Texts and Monographs in Computer Science. Springer Verlag, New York, September 1990.

[FK97] T. H. Fung and R. A. Kowalski. The IFF Proof Procedure for Abductive Logic Programming. *Journal of Logic Programming*, 33(2):151–165, 1997.

[FKM+95] K. Fischer, N. Kuhn, H. J. Müller, J. P. Müller, and M. Pischel. Sophisticated and distributed: The transportation domain. In C. Castelfranchi and J.-P. Müller, editors, *From Reaction to Cognition*, volume 957 of *Lecture Notes in Artificial Intelligence*, pages 122–138. Springer-Verlag, 1995.

[FMP95] K. Fischer, J. P. Müller, and M. Pischel. Unifying control in a layered agent architecture. In *Proceedings of the 1st Intl. Conference on Multiagent Systems*, San Francisco, 1995. AAAI Press/ The MIT Press.

[FN71] R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artifical Intelligence*, 2(3/4):189–208, 1971.

[For82] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[Fre96] E. C. Freuder, editor. *Principles and Practice of Constraint Programming - CP'96*, volume 1118 of *Lecture Notes in Computer Science*, Cambridge, MA, USA, 1996. Springer.

[fRuS99] Institut für Robotik und Systemdynamik. Space robotics. Deutsche Luft- und Raumfahrtgesellschaft, DLR, 1999. URL: `www.robotic.dlr.de`

[Fun96] T. H. Fung. *Abduction by Deduction.* PhD thesis, Imperial College, London, 1996.

[Gab96] D. M. Gabbay. *Labelled deductive systems : volume 1*, volume 33 of *Oxford logic guides.* Clarendon Press, Oxford, 1996.

[Ger99] C. Gerber. *Self-Adaptation and Scalability in Multi-Agent Societies.* PhD thesis, Universität des Saarlandes, Saarbrücken, 1999. to appear.

[GHS+92] G. Große, S. Hölldobler, J. Schneeberger, U. Sigmund, and M. Thielscher. Equational Logic Programming, Actions, and Change. Technical Report AIDA–92–14, FG Intellektik, TH Darmstadt, 1992. Appeared in *Proc. Joint International Conference and Symposium on Logic Programming JICSLP'92.*

[GL87] M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning. In *Proc. of the 6th National Conference on Artificial Intelligence*, 1987.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. A. Kowalski and K. Bowen, editors, *Proceedings 5th International Conference on Logic Programming*, pages 1070–1080, Cambridge, Massachusetts, 1988. MIT Press.

[GL93] M. Gelfond and V. Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming*, 17(2,3, and 1):301–322, 1993.

[GLR91] M. Gelfond, V. Lifschitz, and A. Rabinov. What are the limitations of the situation calculus? In S. Boyer, editor, *Automated Reasoning, Essays in Honor of Woody Bledsoe*, pages 167–181. Kluwer Academic, 1991.

[Gol98] M. P. Golombek. The mars pathfinder mission and science results. In *Proc. of the 29th Lunar and Planetary Science Conference*, 1998.

[Goo76] I. J. Good. *Good Thinking*. University of Minnesota Press, Minneapolis, 1976.

[Gre69] C. Green. Applications of Theorem Proving to Problem Solving. In *Proceedings of IJCAI'69*, 1969.

[GRS88] A. van Gelder, K. Ross, and J.S. Schlipf. Unfounded sets and well–founded semantics for general logic programs. In *Proceedings of the 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 221–230, 1988.

[GS86] D. M. Gabbay and M. J. Sergot. Negation as inconsistency. *Journal of Logic Programming*, 3(1):1–35, 1986.

[GS88] M. Ginsberg and D. E. Smith. Reasoning about actions ii: The qualification problem. *Artificial Intelligence Journal*, 35:311–342, 1988.

[GS90] B. J. Grosz and C. L. Sidner. Plans for discourse. In P. R. Cohen, J. Morgan, and M. E. Pollack, editors, *Intentions in Communication.* MIT Press, 1990.

[GS96] A. Greveni and L. Schubert. Accelerating partial-order planners:some techniques for effective search control and pruning. *Journal of Artificial Intelligence Research*, 5:95–137, 1996.

[GSV99] C. Gerber, J. Siekmann, and G. Vierke. Flexible autonomy in holonic multi-agent systems. In *AAAI Spring Symposium on Agents with Adjustable Autonomy*, 1999.

[Gui95] C. Guilfoyle. Vendors of intelligent agent technologies. In *Agent Software*, pages 92–98. Unicom Seminars, Uxbridge, Middlesex, 1995.

[Hal90] A. Hall. Sevent myths of formal methods. *IEEE Software*, 7(5):11–20, 1990.

[Hay87] I. Hayes, editor. *Specification Case Studies.* Prentice-Hall, London, 1987.

[HdBvdHCM98] K. V. Hindricks, P. S. de Boer, W. van der Hoek, and J. J. Ch. Meyer. A Formal Semantics for an Abstract Agent Programming Language. In *Intelligent Agents IV*, volume 1365 of *Lecture Notes in Artificial Intelligence.* Springer, 1998.

[Hen97] M. Henz. *Objects for Concurrent Constraint Programming*, volume 426 of *The Kluwer International Series in Engineering and Computer Science.* Kluwer Academic Publishers, Boston, November 1997.

[Her67] J. Herbrand. Researches in the theory of demonstration. In J. van Heijenoort, editor, *From Frege to Gödel: A Source Book in Mathematical Logic*, pages 525–581. Harvard University Press, 1967.

[HJ90] S. Haridi and S. Janson. Kernel andorra prolog and its computational model. In *Proc. of the 7th International Conference on Logic Programming*, 1990.

[HM90] J. Y. Halpern and Y. O. Moses. Knowledge and Common Knowledge in a Distributed Environment. *Journal of the the ACM*, 37(3):549–587, 1990.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580 and 583, 1969.

[Hol85] J. H. Holland. Properties of the bucket brigade algorithm. In J. J. Grefenstette, editor, *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, 1985.

[Hor86] E. J. Horvitz. Reasoning about beliefs and actions under computational resource constraints. In *Proc. of the 3rd Workshop on Uncertainty in Artifical Intelligence*, Philadelphia, PA, 1986.

[HS90] S. Hölldobler and J. Schneeberger. A new deductive approach to planning. *New Generation Computing*, 8:225–244, 1990.

[HT96] C. S. Herrmann and M. Thielscher. Reasoning about continuous processes. In B. Clancey and D. Weld, editors, *Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 639–644, Portland, OR, August 1996. MIT Press.

[Hum96] R. Hummel. Uncertainty reasoning in object recognition by image processing. In L. Dorst, M. van Lambalgen, and F. Vorbraak, editors, *Reasoning with Uncertainty in Robotics*, volume 1093 of *Lecture Notes in Artificial Intelligence*, pages 131–145, 1996.

[HW96] M. Henz and J. Würtz. Using Oz for college time tabling. In E.K.Burke and P.Ross, editors, *The Practice and Theory of Automated Time Tabling: The Selected Proceedings of the 1st International Conference on the Practice and Theory of Automated Time Tabling, Edinburgh 1995*, Lecture Notes in Computer Science, vol. 1153, pages 162–177, 1996.

[JC97] H. Jäger and T. Christaller. Dual dynamics: Designing behavior systems for autonomous robots. In *Proceedings of AROB-97 (Artificial Life and Robotics)*, pages 76–79, Beppu, Japan, 1997.

[JF98a] C. G. Jung and K. Fischer. A Layered Agent Calculus with Concurrent, Continuous Processes. In *Intelligent Agents IV*, volume 1365 of *Lecture Notes in Artificial Intelligence*, pages 245–258. Springer, 1998.

[JF98b] C. G. Jung and K. Fischer. Methodological comparison of agent models. Technical Report RR-98-1, DFKI GmbH, Saarbrücken, Germany, 1998.

[JFB96a] C. G. Jung, K. Fischer, and A. Burt. Multi-agent planning using an abductive event calculus. Technical Report RR-96-4, DFKI GmbH, Saarbrücken, Germany, 1996.

[JFB96b] C. G. Jung, K. Fischer, and A. Burt. Resolution, Constructive Negation, and Abduction over Finite Domains in Higher Order Constraint Programming. In *Proceedings of the 1st DFKI Workshop on Constraint-Based Problem Solving*, Saarbrücken, Germany, 1996. DFKI GmbH.

[JFH+99] C. G. Jung, S. Franke, S. Hess, M. Kohlhase, and V. Sorge. Agent-based integration of mathematical services. *Journal of Universal Computer Science*, 5(3):156–187, 1999.

[JFS97] C. G. Jung, K. Fischer, and S. Schacht. Distributed Cognitive Systems. Technical Report D-97-8, Saarbrücken, 1997.

[JG97] C. G. Jung and C. Gerber. Towards the bounded optimal agent society. In C. G. Jung, K. Fischer, and S. Schacht, editors, *Distributed Cognitive Systems*, number D-97-8 in DFKI Document, Saarbrücken, 1997. DFKI GmbH.

[JG98] C. G. Jung and C. Gerber. Resource management for boundedly optimal agent societies. In *Proceedings of the ECAI'98 Workshop on Monitoring and Control of Real-Time Intelligent Systems*, pages 23–28, 1998.

[JHKS98] C. G. Jung, S. Hess, M. Kohlhase, and V. Sorge. An implementation of distributed mathematical services. In *6th CALCULEMUS and TYPES Workshop*, Eindhoven, Netherlands, July 13–15 1998. Electronic Proceedings http://www.win.tue.nl/math/dw/pp/calc/proceedings.html.

[JL87] J. Jaffar and J. L. Lassez. Constraint logic programming. *ACM Principles of Programming Languages*, pages 111–119, 1987.

[JLG99a] C. G. Jung, J. Lind, and C. Gerber. Learning and adaptivity in intelligent real-time systems (extended abstract). In *Proceedings of the International Conference on Autonomous Agents (Agents'99)*, 1999.

[JLG+99b] C. G. Jung, J. Lind, C. Gerber, M. Schillo, P. Funk, and A. Burt. An architecture for co-habited virtual worlds. In C. Landauer and K. L. Bellman, editors, *Virtual Worlds and and Simulation Conference (VWSIM'99)*, Simulation Series. The Society for Computer Simulation International, 1999.

[Jon80] C. B. Jones. *Software Development — A Rigorous Approach.* Prentice-Hall, London, 1980.

[Jon86] C. B. Jones. *Systematic Software Development using VDM.* Prentice-Hall, London, 1986.

[Jun98a] C. G. Jung. On the Role of Computational Models for Specifying Hybrid Agents. In *Cybernetics And Systems'98 — Proceedings of the 14th European Meeting on Cybernetics and System Research*, pages 749–754, Vienna, 1998. Austrian Society for Cybernetic Studies.

[Jun98b] C. G. Jung. Situated abstraction planning by abductive temporal reasoning. In H. Prade, editor, *Proc. of the 13th European Conference on Artificial Intelligence ECAI'98*, pages 383–387. Wiley, 1998.

[Jun99a] C. G. Jung. Emergent mental attitudes in layered agents. In J. P. Müller, M. P. Singh, and A. S. Rao, editors, *Intelligent Agents V*, volume 1555 of *Lecture Notes in Artificial Intelligence*, pages 195–211. Springer, 1999.

[Jun99b] C. G. Jung. Layered, resource-adapting agents in the robocup simulation. In M. Asada and H. Kitano, editors, *RoboCup-99: Robot Soccer World Cup II*, volume 1604 of *Lecture Notes in Artificial Intelligence*. Springer, 1999. to appear.

[KAK+97] H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, and E. Osawa. Robocup: The robot world cup initiative. In *Proc. of The First International Conference on Autonomous Agent (Agents-97)*, Marina del Ray, 1997. The ACM Press.

[KG91] D. Kinny and M. P. Georgeff. Commitment and effectiveness of situated agents. In *Proc. of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 82–88, Sydney, Australia, 1991.

[Kha83] O. Khatib. Dynamic control of manipulators in operational space. In *Sixth IFTOMM Congress on the Theory of Machines and Mechanisms*, December 1983.

[Kit99] H. Kitano. Robocup-rescue: Search and rescue for large scale disasters as a domain for multi-agent research. In *Proceedings of IEEE Conference on Man, Systems, and Cybernetics (SMC-99)*, 1999.

[KK71] R. A. Kowalski and D. Kuehner. Linear resolution with selection function. *Artificial Intelligence*, 2:227–260, 1971.

[KKT93] A. C. Kakas, R. A. Kowalski, and F. Toni. Abductive Logic Programming. *Journal of Logic and Computation*, 2(6):719–770, 1993.

[KKT98] A. C. Kakas, R. A. Kowalski, and F. Toni. *The Role of Abduction in Logic Programming*, volume 5, pages 235—324. Oxford University Press, 1998.

[KL96] L. P. Kaelbling and M. L. Littman. Reinforcement learning: A survey. *Journal of Artificial Intelligence*, 4:237 – 285, 1996.

[KM97a] A. C. Kakas and R. Miller. Reasoning about actions, narratives, and ramifications. *Electronic Transactions on Artificial Intelligence*, 1(4):39–72, 1997.

[KM97b] A. C. Kakas and C. Mourlas. Aclp: Flexible solutions to complex problems. In *Proceedings of Logic Programming and Nonmonotonic Reasoning*, 1997.

[Kow79] R. A. Kowalski. *Logic for Problem Solving*, volume 7 of *Artificial Intelligence Series*. Elsevier Science Publisher B.V. (North-Holland), 1979.

[Kow95] R. A. Kowalski. Logic without model theory. In D. Gabbay, editor, *What is a logical system?* Oxford University Press, 1995.

[KP88] H. A. Kautz and E. P. Pednault. Planning and plan recognition. *AT&T Technical Journal*, 67(1):25–40, 1988.

[KR87] R. Krickhahn and B. Radig. *Die Wissensrepräsentationssprache OPS5*. Vieweg und Sohn, Braunschweig/Wiesbaden, 1987.

[KR90] L. P. Kaelbling and S. J. Rosenschein. Action and planning in embedded agents. In P. Maes, editor, *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*, pages 35–48. MIT/Elsevier, 1990.

[KS86] R. A. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1):67–95, 1986.

[KS94] R. A. Kowalski and F. Sadri. The situation calculus and event calculus compared. In M. Bruynooghe, editor, *Proceedings of the International Logic Programming Symposium*, pages 539—553, Ithaca, New York, 1994. The MIT Press.

[KS96a] H. A. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In B. Clancey and D. Weld, editors, *Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 1194–1201, Portland, OR, August 1996. MIT Press.

[KS96b] R. A. Kowalski and F. Sadri. Towards a unified agent architecture that combines rationality with reactivity. In D. Pedreschi and C. Zaniolo, editors, *Logic in Databases*, volume 1154 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[KT93] K-Team. *Khepera Users Manual*. Lausanne, 1993.

[KTS+98] H. Kitano, M. Tambe, P. Stone, M. Veloso, S. Coradeschi, E. Osawa, H. Matsubara, I. Noda, and M. Asada. The robocup synthetic agent challenge. In *RoboCup-97: Robot Soccer World Cup I*, volume 1395 of *Lecture Notes in Artificial Intelligence*, pages 62–73. Springer, 1998.

[KTW94] R. A. Kowalski, F. Toni, and G. Wetzel. Towards a declarative and efficient glass-box clp language. In N. Fuchs and G. Gottlob, editors, *Proc. of Logic Programming Workshop WLP'94*, Zurich, 1994.

[Kun87] K. Kunen. Negation in Logic Programming. *Journal of Logic Programming*, 4:231 – 245, 1987.

[KWH+93] H. Kitano, B. Wah, L. Hunter, R. Oka, T. Yokoi, and W. Hahn. Grand challenge ai applications. In *Proc. of IJCAI'93*, 1993.

[Leh96] G. Lehmann. Basisalgorithmen zur Situationserkennung in InteRRaP. Master's thesis, Universität des Saarlandes, Saarbrücken, 1996.

[LF97] Y. Labrou and T. Finin. Semantics and conversations for an agent communication language. In M. N. Huhns and M. P. Singh, editors, *Readings in Agents*, pages 234–242. Morgan Kaufmann, 1997.

[LH92] D. M. Lyons and A. J. Hendricks. A Practical Approach to Integrating Reaction and Deliberation. In *Proceedings of the 1st International Conference on Artifical Intelligence Planning Systems*, 1992.

[Lin95] F. Lin. Embracing causality in specifying the indirect effects of actions. In C. S. Mellish, editor, *Proceedings of the International Conference on Artifical Intelligence*, pages 1985–1991, Montreal, Canada, August 1995. Morgan Kaufmann.

[Lin99] J. Lind. *A Software Engineering Development Model for Multiagent Systems*. PhD thesis, Universität des Saarlandes, Saarbrücken, 1999. to appear.

[LLL+94] Y. Lespérance, H. J. Levesque, F. Lin, D. Marcu, R. Reiter, and R. B. Scherl. A Logical Approach to High-Level Robot Programming: A Progress Report. In B. Kuipers, editor, *Control of the Physical World by Intelligent Systems: Papers from the '94 AAAI Fall Symposium*, pages 79–85, New Orleans, 1994.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming. 2nd ext. Edition.* Symbolic Computation. Springer, Berlin - Heidelberg - New York, 1987.

[LMM88] J.-L. Lassez, M. J. Maher, and K. Marriott. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, 1988.

[LRL97] H. Levesque, R. Reiter, and Y. Lespérance. Golog: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31:59–84, 1997.

[LS95] F. Lin and Y. Shoham. Provably correct theories of action. *Journal of ACM*, 42(2):293–320, 1995.

[Mac88] A. Mackworth. *Encyclopedia of AI.* John Wiley & Sons, 1988.

[Mae90] P. Maes, editor. *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back.* MIT/Elsevier, 1990.

[Mah87] M. J. Maher. Logic semantics for a class of committed choice programs. In *Proc. of the 4th International Conference on Logic Programming*. MIT Press, 1987.

[Mah88] M. J. Maher. Complete axiomatizations of the algebras of finite, rational, and infinite trees. In *Proc. of the 3rd Symposium on Logic in Computer Science*, pages 348–357, Edingburgh, 1988.

[Mar82] D. Marr. *Vision.* W. H. Freeman, San Francisco, California, 1982.

[MC91] T. W. Malone and K. Crowston. Toward an interdisciplinary theory of coordination. Technical report, 1991.

[McC58] J. McCarthy. Programs with Common Sense. In *Proceedings of the Symposium on the Mechanization of Thought Processes*, volume 1, pages 77–84, London, November 1958.

[McC63] J. McCarthy. *Situations and Actions and Causal Laws.* Stanford Artificial Intelligence Project, Memo 2, 1963.

[McC85] J. McCarthy. Formalization of STRIPS in situation calculus. Technical report, Formal Reasoning Group, Department of Computer Science, Stanford University, 1985.

[Meh99] M. Mehl. *The Oz Virtual Machine - Records, Transients, and Deep Guards*. PhD thesis, Technische Fakultät der Universität des Saarlandes, 1999. submitted.

[Mey88] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs NY, 1988.

[MH69] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[Mil92] R. Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[Mis91] L. Missiaen. *Localized Abductive Planning with the Event Calculus*. PhD Dissertation, K.U. Leuven, Leuven, September 1991.

[MM82] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM TOPLAS*, 4:258–282, 1982.

[Mor98] W. Morell. Steuerung kontinuierlichen verhaltens in einer hybriden agentenarchitektur. Master's thesis, Universität des Saarlandes, 1998.

[MP98] M. Maher and J.-F. Puget, editors. *Principles and Practice of Constraint Programming - CP'98*, volume 1520 of *Lecture Notes in Computer Science*, Pisa, Italy, 1998. Springer.

[MR95] U. Montanari and F. Rossi, editors. *Principles and Practice of Constraint Programming CP'95*, volume 976 of *Lecture Notes in Computer Science*, Cassis, France, 1995. Springer.

[MS92] M. Meier and J. Schimpf. An architecture for prolog extensions. In *Proceedings of the 3rd International Workshop on Extensions of Logic Programming*, Bologna, 1992.

[Mül96] J. P. Müller. *The Design of Intelligent Agents: A Layered Approach*, volume 1177 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, December 1996.

[Mül99a] J. P. Müller. The right agent (architecture) to do the right thing. In J. P. Müller, M. P. Singh, and A. S. Rao, editors, *Intelligent Agents V — Proceedings of the Fifth International Workshop on Agent Theories, Architectures, and Languages (ATAL-98)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Heidelberg, 1999.

[Mül99b] T. Müller. *The Mozart Constraint Extension Manual*, 1999.

[New90] A. Newell. *Unified theories of cognition.* Harvard University Press, Cambridge, London, 1990.

[Nil84] N. J. Nilsson. Shakey the robot. Technical report, SRI AI Center, April 1984.

[Nod95] I. Noda. Soccer server: a simulator for robocup. In *JSAI AI-Symposium 95: Special Session on RoboCup*, December 1995.

[NS76] A. Newell and H. A. Simon. Computer science as empirical enquiry: Symbols and search. *Communications of the ACM*, 19(3):113–126, 1976.

[Oes97] B. Oestereich. *Objekt-Orientierte Softwareentwicklung mit der Unified Modeling Language.* Number ISBN 3–486–24319–5. R. Oldenbourg Verlag, München, 1997.

[OI92] Y. Ohta and K. Inoue. A forward-chaining hypothetical reasoner based on upside-down meta-interpretation. In *Proc. of the International Conference on Fifth Generation Computer Systems*, pages 522–529. ICOT, 1992.

[PA80] C. R. Perrault and J. F. Allen. A plan-based analysis of indirect speech acts. *American Journal of Computational Linguistics*, 6(3–4):167–182, December 80.

[Par72] D. Parnas. On the criteria to be used in decomposing systems in modules. *Communications of the ACM*, 6(33):636–651, 1972.

[Pet92] A. Pettorossi. *Meta-programming in Logic*, volume 649 of *Lecture Notes in Computer Science.* Springer, 1992.

[Pia99] M. Piaggio. HEIR — a non-hierarchical hybrid architecture for intelligent robots. In J. P. Müller, M. P. Singh, and A. S. Rao, editors, *Intelligent Agents V — Proceedings of the Fifth International Workshop on Agent Theories, Architectures, and Languages (ATAL-98)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Heidelberg, 1999.

[PK94] D. Poole and K. Kanazawa. A decision-theoretic abductive basis for planning. In *Proc. AAAI Spring Symposium on Decision-Theoretic Planning*, 1994.

[Pro99] The NASA New Millenium Program. Deep space 1. NASA, 1999. URL: `http://nmp.jpl.nasa.gov/ds1/`.

[Rei78] J. Reichardt. *Robots: Fact, Fiction, and Prediction.* Penguin Books, New York, 1978.

[Rei80] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[Rei91] R. Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation*, pages 359–380. Academic Press, 1991.

[Rei99] R. Reiter. Knowledge in action: Logical foundations for describing and implementing dynamical systems. URL: `http://www.cs.toronto.edu/cogrobo/`, 1999.

[RG91] A. S. Rao and M. P. Georgeff. Modeling Agents Within a BDI-Architecture. In R. Fikes and E. Sandewall, editors, *Proc. of the 2rd International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 473–484, Cambridge, Mass., April 1991. Morgan Kaufmann.

[RG95] A. S. Rao and M. P. Georgeff. BDI-agents: from theory to practice. In *Proceedings of the First Intl. Conference on Multiagent Systems*, San Francisco, 1995.

[RN95] S. J. Russell and P. Norvig. *Artificial Intelligence, A Modern Approach*. Prentice Hall, 1995.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[Ros96] M. Rosinus. Aladin: A language for designing interrap agents. Master's thesis, Universität des Saarlandes, 1996.

[RS95] S. J. Russell and D. Subramanian. Provably Bounded Optimal Agents. *Journal of Artificial Intelligence Research*, 2, 1995.

[RTM99] T. Raines, M. Tambe, and S. Marsella. Towards automated team analysis: A machine learning approach. In *Proc. of RoboCup'99*, 1999. to appear.

[RW91] S. J. Russell and E. Wefald. *Do the Right Thing*. MIT Press, Cambridge Mass, 1991.

[RZ94] J. S. Rosenschein and G. Zlotkin. *Rules of Encounter: Designing Conventions for Automated Negotiation among Computers*. MIT Press, 1994.

[Sac74] E. D. Sacerdoti. Planning in a Hierarchy of Abstraction Spaces. *Artificial Intelligence*, 5:115–135, 1974.

[San94] E. Sandewall. *Features and Fluents. The Representation of Knowledge about Dynamical Systems*. Oxford University Press, 1994.

[San97] E. Sandewall. Logic-based modelling of goal-directed behaviour. Technical report, Linköping University, 1997. URL: `http://www.ep.liu.se/ea/cis/1997/`.

[Sar93] V. A. Saraswat. *Concurrent Constraint Programming.* The MIT Press, 1993.

[Sch98] R. Scheidhauer. *Design, Implementierung und Evaluierung einer virtuellen Maschine für Oz.* PhD thesis, Universität des Saarlandes, Fachbereich Informatik, Saarbrücken, Germany, December 1998.

[Sea69] J. R. Searle. *Speech Acts.* Cambridge University Press, 1969.

[Sha87] E. Shapiro, editor. *Concurrent Prolog, Volume 1 and 2.* MIT Press, 1987.

[Sha89] M. Shanahan. Prediction is deduction but explanation is abduction. In *Proceedings of the IJCAI 89*, page 1055, 1989.

[Sha90] M. Shanahan. Representing continuous change in the event calculus. In *Proceedings of the ECAI 90*, pages 589–603, August 1990.

[Sha95] M. Shanahan. A circumscriptive calculus of events. *Artificial Intelligence Journal*, 77:249–284, 1995.

[Sha96] M. Shanahan. Robotics and the Common Sense Informatic Situation. In W. Wahlster, editor, *Proceedings of the European Conference on Artificial Intelligence (ECAI'96)*, pages 684–688, 1996.

[Sha97a] M. Shanahan. Event calculus planning revisited. In *Proc. of the Fourth European Conference on Planning*, 1997.

[Sha97b] M. Shanahan. Noise and the Common Sense Informatic Situation for a Mobile Robot. In *Proc. AAAI'96*, pages 1098–1103, 1997.

[Sha97c] M. Shanahan. *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia.* MIT Press, 1997.

[Sho90] Y. Shoham. Agent-oriented programming. Technical report, Stanford University, 1990.

[Sie89] J. H. Siekmann. Unification theory. *Journal of Symbolic Computation*, 7:207–274, 1989.

[Sim82] H. A. Simon. *Models of Bounded Rationality.* MIT Press, Cambridge, 1982.

[Sin91] M. P. Singh. Towards a formal theory of communication for multiagent systems. In *Proc. of the 12th International Conference on Artificial Intelligence*, pages 69–74. Morgan Kaufmann, 1991.

[Smi80] R. G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. In *IEEE Transaction on Computers*, number 12 in C-29, pages 1104–1113, 1980.

[Smo95] G. 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.

[Smo97] G. Smolka, editor. *Principles and Practice of Constraint Programming - CP'97*, volume 1330 of *Lecture Notes in Computer Science*, Linz, Austria, 1997. Springer.

[Som92] I. Sommerville. *Software Engineering*. International Computer Science Series. Addison-Wesley Publishing Company, 1992.

[SP96] A. Sloman and R. Poli. SIM_AGENT: A toolkit for exploring agent designs. In M. Wooldridge, J. P. Müller, and M. Tambe, editors, *Intelligent Agents — Proceedings of the 1995 Workshop on Agent Theories, Architectures, and Languages (ATAL-95)*, volume 1037 of *Lecture Notes in Artificial Intelligence*, pages 392–407. Springer-Verlag, 1996.

[Spi92] M. Spivey. *The Z notation (second edition)*. Prentice Hall International, Hempel Hempstead, England, 1992.

[Sub94] V. S. Subrahmanian. Amalgamating knowledge bases. *ACM Transactions on Database Systems*, 19(2):291–331, 1994.

[SW86] Stanfill and Waltz. Towards memory-based reasoning. *Communications of the ACM*, 29(12), 1986.

[SW98] K. Schild and J. Würtz. Off-line scheduling of a real-time system. In K. M. George, editor, *Proceedings of the 1998 ACM Symposium on Applied Computing, SAC98*, pages 29–38, Atlanta, Georgia, 1998. ACM Press.

[Tel94] G. Tel. *An Introduction to Distributed Algorithms*. Cambridge University Press, 1994.

[Thi97] M. Thielscher. Ramification and causality. *Artificial Intelligence*, 89(1–2):317–364, 1997.

[Thi98] M. Thielscher. How (not) to minimize events. In A. G. Cohn, L. K. Schubert, and S. C. Shapiro, editors, *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, Trento, Italy, June 1998. Morgan Kaufmann.

[Thi99] M. Thielscher. From Situation Calculus to Fluent Calculus: State update axioms as a solution to the inferential frame problem. *Artificial Intelligence Journal*, 1999. (To appear).

[TK95] F. Toni and R. A. Kowalski. Reduction of abductive logic programs to normal logic programs. In L. Sterling, editor, *Proc. International Conference on Logic Programming*, pages 367–381. MIT Press, 1995.

[tTvH98] A. ten Teije and F. van Harmelen. Characterising approximate problem-solving: From partially fulfilled preconditions to partially achieved functionality. In H. Prade, editor, *Proc. of the 13th Biennial European Conference on Artificial Intelligence (ECAI'98)*, pages 78–82, 1998.

[Van89] A. VanGelder. The alternating fixpoint semantics of logic programs with negation. In *ACM Symposium of Principles of Database Systems*, pages 1–10. Association for Computing Machinery, 1989.

[vEK76] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[Vie99] G. Vierke. *Cooperative and Competitive Resource and Task Allocation in the Haulage Domain with a Holonic Multi-Agent System*. PhD thesis, Universität des Saarlandes, Saarbrücken, 1999. to appear.

[vM92] F. v. Martial. *Coordinating Plans of Autonomous Agents*, volume 610 of *Lecture Notes in Artificial Intelligence*. Springer, 1992.

[vNM44] J. von Neumann and O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, Princeton, 1944.

[Wal97] D. L. Waltz. *Understanding Line Drawings of Scenes with Shadows*, pages 19–91. McGraw Hill, 197.

[War83] D. H. D. Warren. An abstract prolog instruction set. Technical report, October 1983.

[Wel94] D. Weld. An introduction to least-commitment planning. *AI Magazine*, 15(4):27–62, 1994.

[Wen99] A. Wenner. Hybride agenten und entscheidungsfindung in der telerobotik. Master's thesis, Universität des Saarlandes, Saarbrücken, 1999.

[WHR96] R. Washington and B. Hayes Roth. Incremental Abstraction Planning for Limited-Time Situations. In *New Directions in AI Planning*, pages 91–102. IOS press, 1996.

[Wie99] M. A. Wieczorek. The lunokhod memorial, 1999.

[Wir71] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 4(14):221–227, 1971.

[Wir76] N. Wirth. *Systematic Programming: An Introduction.* Prentice-Hall, Englewood Cliffs, NY, 1976.

[WJ95] M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2), 1995.

[Woo95] M. Wooldridge. This is myworld: The logic of an agent-oriented testbed for dai. In M. Wooldridge and N. Jennings, editors, *Intelligent Agents: Theories, Architectures, and Languages*, volume 890 of *Lecture Notes in Artificial Intelligence*, pages 160–178. Springer-Verlag, 1995.

[Woo96] M. Wooldridge. Practical Reasoning with Procedural Knowledge: A Logic of BDI Agents with Know-How. In *Proceedings of the International Conference on Formal and Applied Practical Reasoning.* Springer-Verlag, 1996.

[Wür98] J. Würtz. *Lösen kombinatorischer Probleme mit Constraintprogrammierung in Oz.* PhD thesis, Universität des Saarlandes, Fachbereich Informatik, Saarbrücken, Germany, January 1998.

[Xia95] J. Xiaoping. ZTC: A Type Checker for Z Notation — user's guide. 1995. ftp://ise.cs.depaul.edu/pub/ZTC.

[Yan90] Q. Yang. Formalizing planning knowledge for hierarchical planning. *Computational Intelligence*, 6:12–24, 1990.

[Zil95] S. Zilberstein. Models of Bounded Rationality. In *AAAI Fall Symposium on Rational Agency*, Cambridge, Massachusetts, November 1995.

Theory and Practice of Hybrid Agents

Christoph G. Jung and Klaus Fischer

RR-01-01
Research Report