Masterarbeit am Institut für Informatik der Freien Universität Berlin,

Arbeitsgruppe Künstliche Intelligenz

Agent-based Blackboard Architecture for a Higher-Order Theorem Prover

Max Wisniewski
max.wisniewski@fu-berlin.de

Betreuer: PD. Dr. Christoph Benzmüller

Berlin, den 17. Oktober 2014

Abstract

The automated theorem prover Leo was one of the first systems able to prove theorems in higher-order logic. Since the first version of Leo many other systems emerged and outperformed Leo and its successor Leo-II. The Leo-III project's aim is to develop a new prover reclaiming the lead in the area of higher-order theorem proving.

Current competitive theorem provers sequentially manipulate sets of formulas in a global loop to obtain a proof. Nowadays in almost every area in computer science, concurrent and parallel approaches are increasingly used. Although some research towards parallel theorem proving has been done and even some systems were implemented, most modern theorem provers do not use any form of parallelism.

In this thesis we present an architecture for Leo-III that use parallelism in its very core. To this end an agent-based blackboard architecture is employed. Agents denote independent programs which can act on their own. In comparison to classical theorem prover architectures, the global loop is broken down to a set of tasks that can be computed in parallel. The results produced by all agents will be stored in a blackboard, a globally shared datastructure, thus visible to all other agents.

For a proof of concept example agents are given demonstrating an agent-based approach can be used to implemented a higher-order theorem prover.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Berlin, den _____ _____

(Unterschrift)

Contents

1 Introduction

The Leo-III Project's aim is to write a new Higher-Order Theorem Prover that will be the successor to Leo-II. Leo-III will make use of massive parallelism through an agent-based approach and a blackboard architecture.

Classical Theorem Provers like Leo-II [BPTF07], Satallax [Bro12], or the first order theorem prover E [Sch13] are primarily using a single loop in which they manipulate a set of formulas to obtain a proof for a conjecture or disprove it. If in a classical theorem prover a formula is to be manipulated and there are two alternative options, one is taken and the other is discarded. The concept of parallelism aspired in Leo-III is to compute both alternatives. This way we can search for a solution in both alternatives. Even better both branches can learn from each other, such that same computations are not performed twice.

Although parallelization in the context of theorem provers is well studied and documented by for example Bonacina [Bon99], they have not been implemented in a competitive theorem prover. In general in Bonacina's taxonomy on parallelization can be divided in three major categories. Parallelization on the search level, where the alternatives on how to continue the proof are done concurrently and parallelization on the clause level, where mid level computations are done in parallel, for example a resolution. The last category is on the term level, where tasks as simplification are performed parallel.

In Leo-III the parallelization is done through agents operating on a blackboard. An agent, as described by Weiss [Wei13], is an independent process that only performs simple tasks. Simple in the sense that an agent cannot solve the complete problem on his own. In Leo-III an agent could for example traverse a set of formulas and delete redundancies. Although this agent will never solve the problem of finding a proof, he will enhance the performance of other agents by reducing the search space.

The blackboard is a global shared datastructure. Every formula and any result an agent produces is stored in the blackboard. The desired effect is that even unsuccessful alternatives that are explored during the proof search can contribute to the final result.

The architecture that will be designed through this thesis will not be limited to one of the categories of Bonacina. Nevertheless the agent-based approach will give a general method to parallelize on any of the search, clause and term level.

The main source of inspiration is Volker Sorge's Ω-Ants [Sor01] blackboard architecture. He utilized a blackboard approach and agents to query specialized provers. The results of these provers are suggestions to the user, on which he can make decisions in his interactive proof search.

Orthogonal to this approach Leo-III will use the agents to perform the actions directly in the blackboard and produce the proof by themselves this way.

The goal of this thesis is to show that an agent-based approach can be used to implement a competitive automated theorem prover.

The outline of the thesis is the following. In section 2 the concepts of theorem proving and automation are presented. In section 3 the blackboard is introduced and some synchronization techniques, investigated through the course of this thesis, are presented and concluded by the decision for the synchronization of Leo-III 's blackboard. In section 4 agents and multiagent systems are introduced and the (multi-)agent architecture for Leo-III is explained. Section 5 deals with the scheduler for the agents and introduces game theory as the foundations for the scheduling algorithm. In section 6 and 7 example agents and datastructures for Leo-III are presented and evaluated.

2 Theorem Proving

The art of proving a statement based on a set of assumptions is a millennia old tradition in the western world. In ancient Greek philosophers as Plato, Socrates and especially Aristoteles in his *Organon* introduced an uniform way of deriving new information from given ones. Thus the first logic was born.

These early attempts of logic struggled with problems due to interpretation on either the facts in their proofs and on the other hand the derivation style. The human language was a big problem in this case, because the interpretation of words can change in different context. A fact formulated in human language, even if logical consistent derived, can be wrong if interpreted in another way. Without a formalization in which way to read certain facts, this poses a great problem as can be seen in table 2.1.

A	Nothing is better than money	
B	A sandwich is better than nothing	
C	A sandwich is better than money	from A, B

Table 2.1: Simple example of two different interpretations of the term *nothing*.

Without question we would syntactically agree on the derivation of C, but even if most people would agree on A and B, they would most certainly not agree on C. The problem here is the interpretation of *nothing*. In A *nothing* can be seen as a quantification of the whole sentence – *for all objects that exist, money is better than each of it*. In B a reference to the ranking itself is made, in this case some kind of measure. *Nothing* has the measure 0 where a set containing the sandwich has a measure strictly greater. In this sentence there is no quantification, but *nothing* is the description of the empty set.

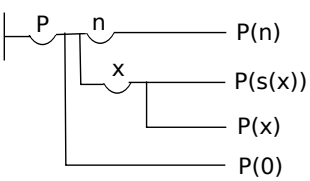
Not only is A a implicit quantified sentence and B a propositional, but we even cannot be sure the comparison *better* means the same relation.

Nonetheless the first formalization of the proof emphasizes the derivation is correct. In a sense this derivation is still correct, if we take *nothing, sandwich*

and *money* as simple objects. But then we would have not modeled what our intended interpretation was.

The formalization of logic is a rather recent development and goes back to Frege in his *Begriffsschrift* [Fre79] in 1879. He developed a style to write down a proof mathematically, without uncertainty. The logic addressed in his article was *classical predicate logic*.

In figure 2.1 the induction principle is shown. It states in both cases the same. If P holds for some 0 and we can derive from $P(x)$ that it holds for the successor $s(x)$, then we can conclude it holds for all natural numbers. We can already see a resemblance in both calculi. Most natural deduction calculi are written down as a tree, exactly as Frege introduced his calculus. The difference is just in the set of rules that can be applied and a the notation of the trees.



$$\frac{P(0) \qquad \begin{array}{c} P(x) \\ \vdots \\ P(s(x)) \end{array}}{\forall n \,.\, P(n)}$$

Figure 2.1: Induction principle in Frege's proof style compared to natural deduction.

With the beginning of the formalization of proofs there was suddenly the possibility to analyze proof systems formally with their own methods. Today, especially in philosophy, there exists a variety of different proof systems and logics.

In the remainder of this section we firstly introduce the logic Leo-III is situated in. After this we will look at the automation of the proofs and in the end the key aspect to Leo-III, the parallelization, is illuminated.

2.1 Higher Order Logic

One of the key features of natural language is the abstraction from specific things to talk about. For humans it is natural to formulate sentences in the style of *every student in the course sits in the lecture hall* or *every (something with predicate abstraction*.

In the first example we see an abstraction over students, i.e. quantification over each student and in the second an abstraction over all predicates. Early work on this subject is again done in Frege's *Begriffsschrift* [Fre79]. He does not explicitly mention the fact that he allows quantification over predicates, but as he encodes the induction principle (see figure 2.1), we can assume that this is the case.

Bertrand Russel [Rus08] first pointed out that quantification, as introduced by Frege, is inconsistent, by introducing his famous *Russel Paradox*.

He has proven that in Frege's logic it is possible to formulate the following set – *the set of all sets, that do not contain itself.* It is not provable if this set contains itself or the contradiction.

To get rid of this paradox Russel introduced his *ramified theory of types.* But as Goedel has shown in his first incompleteness theorem [Gö31] there cannot exist a consistent, complete proof calculi when quantifying unrestricted over predicates. The *classical higher order logic* or *simple type theory* used nowadays refers to the work of Alonso Church [And14] with henkin models to ensure the existence of such a proof calculi.

2.1.1 Church's simple theory of types

The input language of Leo-III is similar to the language used in Church's simple type theory. This section shortly introduces his logic. For most parts, we can see *classical higher order logic* as a typed λ-calculus with special symbols for the connectives to define the logic.

Definition 2.1. (type)
We call o and ι base types, and $\alpha \to \beta$ a function type from a type α to a type β.

The set of all types is denoted by $\mathcal{T}$. ⌟

The type o will be the Boolean type, whereas ι is the type of individuals, the simplest objects in the logic. Based on these the typed terms of the simply typed λ-calculus are introduced.

Definition 2.2. (typed term)
A term t in the simply typed λ-calculus annotated with a type α, meaning t_α is the term t with type α. The Baccus Nauer form for well-typed terms is as follows.

$$\begin{array}{rcl} t_\alpha & ::= & c_\alpha,\, X_\alpha \\ & | & t^1_{\beta\to\alpha}\; t^2_\beta \\ & | & (\lambda X_\beta \,.\, t^1_\gamma)_{\beta\to\gamma}, \text{ with } \alpha = \beta \to \gamma \end{array}$$

Where c_α is a constant symbol of type α from a set of symbols $\mathcal{C}$, X_α is a variable of type α from a set of variable symbols σ, $t^1_{\beta\to\alpha} t^2_\beta$ is an application and $(\lambda X_\beta \,.\, t^1_\gamma)$ is a function. As in the untyped λ-calculus we can introduce the set of *free* terms. A variable X_α is *free* in t, if it is contained in t and no abstraction $(\lambda X.t')$ exists where this X is contained in t'. A variable is *bound* otherwise.

Observe that it is not possible to pass a wrong typed argument to a function, as there cannot be a well-formed application with wrong type. ⌟

Higher-order logic is a term describing a quantification over higher ordered variables. We will look at this concept later in this chapter. The next definition introduces the order for the term.

Definition 2.3. (Type / Term order)
The order function *ord* is defined inductively through

$$\begin{aligned} ord(o) = ord(\iota) &= 0 \\ ord(\alpha \to \beta) &= \max\{ord(\alpha)+1, ord(\beta)\}. \end{aligned}$$

The order of a term t_α is defined by $ord(t_\alpha) = ord(\alpha)$. □

This definition is enough to introduce higher-order logic. The logical connectives and operators can be introduced as constant symbols with a fixed interpretation that has to be available in the semantics later on. To evaluate these terms a applicative structure [BBK04] can be used to carry the information to constant and variable symbols. For simplicity reasons the approach here is to define the set of symbols and their semantics explicitly on top of the typed λ-calculus, inspired by Andrews [And14].

Definition 2.4. (higher order-logic syntax)
The classical higher-order logic (HOL) is a *typed* λ *calculus* with the connectives $\forall_{(\alpha\to o)\to o}$, $\neg_{o\to o}$, $\wedge_{o\to o\to o}$. ⌟

The other logical connectives can be defined the usual way. Observe that the quantifier $\forall$ does not introduce a binding mechanism. Whereas usually a quantification is written by $\forall X.PX$, it is written as $\forall(\lambda X.PX)$ in HOL. This way quantifiers do not have to be part of the syntax definition, but can be defined as constant symbols with a fixed meaning.

A term t_o is called a *sentence* if all variables are *bound*.

Using this we can express the difference between higher-order and first-order logic. First order quantifiers allow only quantification over predicates P with $ord(P) = 1$, where the 1 is the relevant term. This logic is called *first-order logic* (FOL). In higher-order logic quantification is allowed over any kind of predicate P, in particular $ord(P) > 1$.

2.1.2 Semantics

The semantic of HOL can be introduced in various ways. Since this thesis does not tackle the logical side, we will restrict ourselves to a simple introduction to standard and henkin semantics.

We start by introducing the models for the evaluation of the formulas.

Definition 2.5. (Frame & Interpretation)
Let $\{D_\alpha\}_{\alpha\in\mathcal{T}}$ be a collection of sets, where a D_α is a set of elements of type α called *domain*. The *domain* $D_o = \{T, F\}$ contains two elements denoting truth and falsehood. The *domain* $D_\iota \neq \emptyset$ can be chosen arbitrarily and $D_{\alpha\to\beta}$ is a set of functions from D_α to D_β.

$\{D_\alpha\}_{\alpha\in\mathcal{T}}$ is called a *frame*.

An *interpretation* $\langle\{D_\alpha\}_{\alpha\in\mathcal{T}}, \mathcal{I}\rangle$ is a *frame* paired with a function $\mathcal{I}$, that maps each constant symbol c_α to an element in D_α. The function I has to be total, such that each constant symbol has a element assigned. ⌟

The interpretation of the HOL connectives $\wedge_{o\to o\to o}, \neg_{o\to o} \prod^{\alpha}_{(\alpha\to o)\to o}$ is the usually assumed, s.t. $\mathcal{I}(\wedge)$ returns true, if and only if both arguments are true, and $\mathcal{I}(\neg)$ returns the other element in D_o. Lastly $\mathcal{I}(\prod^{\alpha})$ returns true for a given function $p_{\alpha\to o}$, if the predicate is T for all elements in D_α. We need the *interpretation* to contain at least these functions.

Definition 2.6. (Valuation)
Let $\langle \{D_\alpha\}_{\alpha\in\mathcal{T}}, \mathcal{I}\rangle$ and σ a variable mapping that assigns a variable X_α an element of the domain D_α. The substitution $\sigma[a/X_\alpha]$ denotes the assignment that maps all variables to the same element, except X_α, that maps now to a.

The *valuation* $\mathcal{V}$ maps a term t_α with a variable mapping σ to an element of the *domain* D_α with

$$\begin{array}{lcll}
\mathcal{V}(X_\alpha, \sigma) & = & \sigma(X_\alpha) & \text{, for } X_\alpha \text{ variable} \\
\mathcal{V}(c_\alpha, \sigma) & = & \mathcal{I}(c_\alpha) & \text{, for } c_\alpha \text{ constant} \\
\mathcal{V}((s_{\alpha\to\beta} r_\alpha), \sigma) & = & (\mathcal{V}(s_{\alpha\to\beta}, \sigma)\mathcal{V}(r_\alpha, \sigma)) & \\
\mathcal{V}((\lambda X_\alpha \,.\, s_\beta), \sigma) & = & f_{\alpha\to\beta} \in D_{\alpha\to\beta}, \text{ s.t. each } z \in D_\alpha \text{ is mapped to } \mathcal{V}(s_\beta, \sigma[z/X_\alpha]) &
\end{array}$$

⌟

An *interpretation* $\langle \{D_\alpha\}_{\alpha\to\beta}, \mathcal{I}\rangle$ is a *henkin* or *standard* model if a valuation $\mathcal{V}$ exists. If it is a henkin/standard model, then the valuation is unique.

The above definitions exactly define *henkin model* and hence for the valuation *henkin semantic*. The difference between standard and henkin models, is that a domain in a henkin model the domain $D_{\alpha\to\beta} \subseteq D_\beta^{D_\alpha}$ is a subset of all functions. In a standard model the domain $D_{\alpha\to\beta} = D_\beta^{D_\alpha}$ is always full. Lastly we can define validity for a model.

Definition 2.7. (Validity)
A term t_o called *formula* is valid in a henkin/standard model H if and only if for all variable assignments σ $\mathcal{V}(t_o, \sigma) = T$, for $\mathcal{V}$ the unique valuation for H.

We write $H \models t_o$.

A formula t_o is valid in henkin(standard) semantic, if for all henkin(standard) models H $H \models t_o$, then we write $\models t_o$. ⌟

To show the model validity of a formula, human cannot go the way of the valuation function, since the domains D_α can be infinite. Hence the notion of a proof system is necessary to show validity.

2.1.3 Proofs

The task of a proof calculus is to generate from a set of valid formulas new valid formulas. To form a proof calculus an initial set of valid and rules about how to generate new valid formulas have to be given.

The next definition introduces the notion of a proof in one of the simplest proof calculi first formulated by Hilbert [vH02].

Definition 2.8. (Hilbert Proof System)
Let Γ be a set of formulas called *axioms* and $\mathcal{R}$ a set of inference rules.

1. A sequence $\phi_0, \phi_1, \ldots, \phi_n$ is a proof sequence if for all i
   - $\phi_i \in \Gamma$
   - ϕ_i follows from a rule in $\mathcal{R}$ by some $\phi_{j_1}, \ldots, \phi_{j_k}$, whereby $j_1, \ldots, j_k \leq i$.
2. A proof sequence $\phi_0, \ldots, \phi_n$ is a proof for the hypotheses Ψ if $\phi_n = \Psi$.

   We write $\Gamma \vdash_{\mathcal{R}} \Psi$.

The proof calculus was introduced to proof formulas without the modal validity. The next definition introduces two properties for a proof calculus that makes a connection to the model validity.

Definition 2.9. (Soundness & Completeness)
Let $(\mathcal{R}, \Gamma)$ be a proof calculus. The proof calculus is called

1. *sound* if for all Φ_o terms of type o holds
$$\Gamma \vdash_{\mathcal{R}} \Phi_o \Rightarrow \models \Phi_o.$$
2. *correct* if for all Φ_o terms of type o holds
$$\models \Phi_o \Rightarrow \Gamma \vdash_{\mathcal{R}} \Phi_o$$

The result of Goedel's incompleteness result [Gö31] has shown that for *standard models* there is no *sound* and *complete* proof calculus. Therefor we already introduced *henkin models* for which *sound* and *complete* calculi exist. On the other hand this result is due to the fact that HOL with *henkin models* is exactly as expressive as first-order logic.

2.2 Automated Theorem Proving

From the development of the first to modern computers, the computation power as well es the applications had a great boost. Their first role as mathematical computation engine was soon applied in many non mathematical fields. Whereas computer were on the rise through the century in many application, in mathematics their application stagnated except for some discipline that relied on heavy computation.

Using computers in the proof search, which is a key aspect of mathematics, has been and in some parts is still a proscribed topic.

The first automated theorem prover, getting known and used for its stability, was Otter [McC90] which was released 1989. Otter is a first-order logic theorem prover using resolution and paramodulation. Since Otter has been stable a long time, many fields started applying theorem proving. Some of the

more popular applications are the seL4 kernel [KEH+09], where a complete kernel written in C was proven correctly. The DVD codec's correctness proof in Coq [YB04] could be automatically translated to executable ML code that was firstly done by Ralph Loader by using the *Curry-Howard isomorphism* to generate executable code. A last application is in philosophy and formal logics, where recently Gödels ontological argument could be verified [BP14]. Although not widely used in mathematics, there are examples of proofs that were only found through theorem provers. Firstly there was the four color theorem proven in *Coq* and most recently the Kepler conjecture, that was proven with help of *Isabelle/HOL* [NPW02].

In computer aided proving there are two major styles to distinguish. One tries to find a proof directly by themselves, completely liberating the person from the work process. The result is either a proof object, which can be verified by another program or the human himself, or simpler the answer *yes/no*. These kind of provers are called *automated theorem prover* (ATP). The other approach is a more interactive variant. The computer just supports the human in the proof search by showing him possible applications of rules, applying the rules and verifying every step. The verification support ensures, that no slip is in the proof. These provers are called *interactive theorem prover* (ITP).

2.2.1 Automated Proof Calculi

We have already seen the syntax and semantics of higher-order logics that we want to use and have seen a derivation system that can formulate a proof. But for a computer these simple derivation systems as presented in definition 2.8 are poorly efficient, since a computer would not know which axioms to take and initiate. In the worst case the proof will never be found.

For this reason more suitable proof calculi have been implemented. Some examples are

Tableaux: Tableaux style proofs split *conjunctive* clauses in the formula set and branches them into a tree structure. A proof is found if every branch is closed, namely *false* could be derived.

Resolution: Resolution proofs work by translating the formula set step-wise into *conjunctive clause normal form* (CNF). By combining to CNF clauses, if they contain a formula ϕ and its negated formula $\neg\phi$, a new clause can be generated, with all formulas of both combined clauses without the ϕ.

The proof can be closed if the empty clause could be derived.

Both tableaux and resolution are well described by Fitting [Fit90].

DP: The proof style of Davis and Putnam [DP60] works by instantiating the literals of a CNF with values *true* or *false*.

Through an intelligent backtracking in the case that the empty clause could be derived in one instantiation, the proof can either find a valid instantiation, or show, that now instantiation can be found.

This proof style is a satisfiability proof style that can return a model in the case that the initial formula set is not valid.

All the above proof calculi have been proven sound and complete.

Modern Theorem Provers Since Otter many new provers came up with enhancements in different fields. On the one hand, there are Spass, E, and Prover9, which is the direct successor to Otter, in the field of first order logic, on the other hand there are Leo-II and Satallax.

Today's performance is mainly due to the yearly competitions in which the provers can compete in different categories. They differ in more advanced calculi, better selection and filtering for the rules to apply or partial rewriting of formulas to a more suitable form to produce each year faster provers.

2.2.2 The TPTP World

In the early days every new prover came up with its own interface, its own input language and its own proof output. This has proven to be a challenging task for people trying to use these theorem provers since often multiple provers were needed for a single problem due to their individual strengths.

The need for some kind of standard was immanent, since both users and developers do not have to be burdened with syntax for each new prover, but concentrate on the real problem they want to solve. In the pure quantified theorem proving context the most established framework is the *Thousand Problems for Theorem Provers* (TPTP) [Sut09] developed and maintained by Goeff Sutcliffe. The TPTP is on the one hand a library for test problems for theorem provers and on the other hand TPTP defines many categories of input languages, such as *cnf* for input in clause normal form, *fof* for untyped first order problems and – important for Leo-III – *thf* the typed higher order form.

As it already happened in logics by Frege it is necessary to standardize the languages of theorem provers. If there is a standard theorem proving can propagate to many possible application areas, e.g. philosophy, mathematics and computer linguists. The problem library allows a transparent testing of theorem provers. This way a developer of a new prover does not have to think of possible problems and on the other hand he cannot cheat in their evaluation of the prover. Each prover has some problems on which he runs faster than others. A standardized testing framework prohibits a new prover to profile himself by only running on suitable test for him, but allows a non biased evaluation of the qualities the prover owns.

Since many ATPs began to use TPTP as their input language and used the problem library, a web interface called *SystemOnTPTP* allows to run many provers remotely on servers in Miami. *SystemOnTPTP* workflow is described in figure 2.2. A problem in one of the supported syntaxes is send to the server. On the server there are sever tools for syntax checking and optimize formulation of the input syntax. For provers that do not support TPTP's syntax as their input format translation tools are available. The processed input is then send

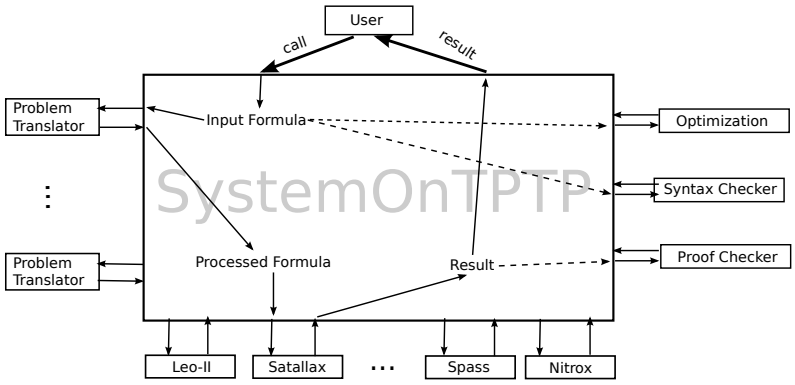


Figure 2.2: Abstract communication in the SystemOnTPTP server architecture.

to one ore more provers. The result can be processed again by other tools, e.g. a proof checker. After all tools that were selected by the user run the result will be returned to the caller.

Competitions In the process of developing performant theorem provers, some sort of competition has proven to be very successful. On the one hand the developers of the provers can compare their provers, which is itself a good motivation to build stronger, more performant systems. But on the other hand people from communities only interested in proving certain kind of problems can formulate these in TPTP syntax and send them to the competitions. As long as they are part of the competition, the provers will enhance just to perform better in the competition and by that enhance their capabilities in the field the community needs.

The predecessor to Leo-III, Leo-II, performed quite well in the past, for he won the *CASC-J5* competition [Sut11] in the *thf* division. The *CASC* is one the bigger competitions for theorem provers. It is based on TPTP input syntaxes and split in multiple divisions, depending on the expressiveness of the prover.

2.3 Parallelization of Theorem Proving

Whereas sequential theorem proving has been successful under development for several decades, this is not the case for the task of parallelizing theorem provers. Maria Paola Bonacina [Bon99] investigated many of the used possibilities on how to parallelize theorem provers, but very little of the parallelizations are used in state-of-the-art theorem provers.

Firstly we have to define some term regarding the definition of parallel programs.

Definition 2.10. (Parallel Programs)
Let $\mathcal{P}$ be a program. We call $\mathcal{P}$

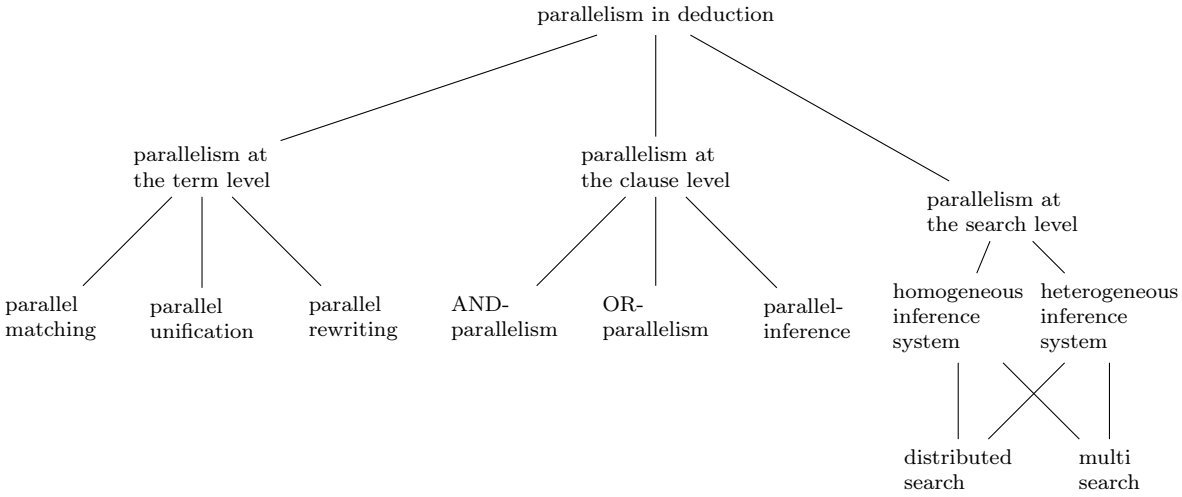


Figure 2.3: Types of parallelism in proof calculi

(i) non-sequential if there is more than one control state.

(ii) concurrent if the processes share memory.

(iii) parallel if the processes run physically independent and at the same time.

⌟

In this definition it is notable that a non-sequential or concurrent program does not need to run parallel. Early multi-threading indeed simulated this behavior by scheduling the processes while they run. They could interfere with each other, but because there was but one processor, there was no real parallelism.

Through this work parallel and concurrent will often be used identically, since we assume that the designed systems will run on systems with more than one processor or even distributed over many computers, and concurrent, because most of the designed systems will write on the same formula set directly, hence it is concurrent.

As described in the previous section, most calculi work by translating the formula set into some kind of clause form. Hence we can look at the three types of parallelism shown in figure 2.3 in the second column.

Term level parallelism The first subtree *parallelism at term level* uses the parallelism to manipulate the smallest units in the context. Dealing with terms there can be simplifications, rewritings or unifications computed in parallel. Parallelizing at the term level results in almost no interference between the different processes, since the changes done, are mostly local so two processes manipulating different terms will not affect each other.

	α-rule	β-rule
$A \wedge B$	A, B	-
$\neg(A \wedge B)$	-	$\neg A$, $\neg B$
$A \vee B$	-	A, B
$\neg(A \vee B)$	$\neg A$, $\neg B$	-
$A \supset B$	-	$\neg A$, B
$\neg(A \supset B)$	A, $\neg B$	-
	...	

Table 2.2: α - β - nature of connectives

Clause level parallelism The second subtree *parallelism at clause level* is one step up from the term level parallelism. On this level inference rules can be applied in parallel. This can be done either direct, as emphasized with the last part, by inserting inferred clauses into the search space. More commonly used are AND-/OR-parallelism.

AND-/OR-parallelism Propositional logical formulas can be divided in three categories. First they can be literals that are constants or negation of constants. Secondly are fundamental OR(β) connectives and lastly fundamental AND(α) connectives, written as $\alpha(\Phi_1, \Phi_2)$ or $\beta(\Phi_1, \Phi_2)$. This categorization, together with a δ rule for quantifier, was introduced by Melvin Fitting [Fit96]. It is long known that $\wedge, \vee, \neg$ is enough to define propositional logic and with the distribution law for $\wedge, \vee$ there can be always one of both on top.

In table 2.2 we see each formula is part of one category. The idea behind this partition is that α connectives can be proven if both generated formulas can be proven. In β connectives only on of the two has to be proven.

The method can still be applied in higher-order logic as long as the formula has at top level propositional character.

This fundamental structure can be used to split the search context. In a refutation proof, the goal is divided in two parts.

In figure 2.4 the α- splitting rule can be seen. The advantage is that both subgoals can be proven independently [1].

Thus two processes can work independently on both goals. In the case of α connectives, the proof needs to wait for both subgoals to be closed. If the connective is of type β only one has to find a proof and the common goal can be closed.

In comparison to computing other inference rules in parallel, the splitting in these cases has the advantage of lesser inferences if two rules are applied at the same time because the independence of the proof context.

It remains to say that most theorem provers limit themselves to either of the splitting rules.

[1] This is true except for still shared context, such as unification of Skolem variables that has to be done in both parts equally.

$$\frac{\Phi, \alpha(\varphi_1, \varphi_2) \vdash \Psi}{\Phi, \varphi_1 \vdash \Psi \qquad \Phi, \varphi_2 \vdash \Psi}$$

Figure 2.4: α-split of the proof context

Search level parallelism At the highest level of abstraction the parallelism is at the stage to decide on a rule, that should be applied to the proof context.

This task on the opposite site of the task to compute these rules on terms or clauses. Since the system won't benefit from two equal processes, searching the same set of clauses for possible rules to apply, some kind of difference has to be made between the processes.

The kinds of parallelism can be seen in figure 2.3 and are described in the following.

Homogeneous inference system. The first distinction is made in the capabilities of the processes. In a homogeneous inference system all processes have the same rules they check for applicability.

Heterogeneous inference system. The other case is that the processes have different sets of inference rules, that check for applicability.

Distributed Search. The second distinction is made in the available search space. In the case of the distributed search the search space is to some degree divided, such that the processes have only a partial view on the formula set.

Multi Search. In multi search parallelism all processes see the complete search space and try to find applicable inference rules here.

The figure emphasizes that the two categories can be combined freely. Whereby one has to be careful with the combination of homogeneous inference systems with multi search and heterogeneous inference systems with distributed search.

In the first example the system might do the exact same work many times. Hence at least the search strategies of the processes have to differ. In the second case the distribution has to be made carefully. Assume a proof contains necessarily an application of rule R_1 on some formulas $\Phi_1, \ldots \Phi_n$. The process tasked to watch over R_1 never sees all these formulas and hence the proof can not be found. Such an implementation would not be complete.

As already talked about the two major properties proof calculi and their implementations should always have are *soundness* and *completeness*.

Talking about non-sequential programs, there are two kinds of properties the programs should be able to fulfill. Without defining the underlying modal logic or the models behind them, we will look at a colloquial definition.

Definition 2.11. (Safety & Liveness Properties)
Let P be a property and $\mathcal{M}$ a program.

1. A safety property save(P) is valid in the program $\mathcal{M}$ if never during the execution of $\mathcal{M}$ stating in a valid start state the formula P is true.

2. A liveness property live(P) is valid in the program $\mathcal{M}$ if the formula P is evaluated infinitely often to true during every possible computation of $\mathcal{M}$ starting in a valid start state.

In designing parallel theorem provers both these kinds of properties are closely connected.

Soundness / Safety. To proof soundness of a parallel theorem prover, almost always some kind of safety property has to be proven.

One imagine a theorem prover where the underlying calculus is correct, but in the parallel execution an error happens. Two processes attempt to write at the same time and in the result they add a formula to the context that should not be derivable, for example false.

Another example dealt with is in the parallelized version of Otter called ROO [LMS92a]. An invariant from Otter that in the set of *processed formulas* every pair of formulas was mutually normalized is established had to be maintained This was some work, that must not be dealt with in Otter, because there only one formula, was processed at a time. In ROO, many formulas could be processed in parallel, such that it might occur, that at the same time processed formulas, where not mutually normalized. The invariant of Otter became a safety property in ROO, that had to be maintained.

Most of the time the safety properties are invariants made on the members of the datastructures.

Completeness / Liveness. The completeness of most calculi relies heavily on the fact that every inference rule can be applied infinitely many times.

If for example the α rule mentioned above could no longer be applied, but there are still *and* connectives in the context that have to be split, the proof might not be found.

Hence the completeness of the calculus relies on the fact, that each process can execute infinitely often.

The agent-based blackboard architecture presented in this thesis is not limited to one of the kinds of parallelization. The architecture is only a tool that can be used to implement agents on all of these levels of parallelism.

3 Blackboard

One of the key aspects for Leo-III is the massive use of parallelism. Therefor the aim is to write all datastructures ready for concurrent access and modification. Aside from the concurrent access the general idea in Leo-III is information sharing.

The approach in Leo-III is to share the information by making almost every data used for the computation by the agents public. An architecture for these requirements is the *Blackboard Architecture*.

All agents write their intermediate results onto the *blackboard*, where the other agents can see partial results. Hence a computation has probably never to be done twice in different contexts, if it has already been done, and even a failed proof attempt can leave conclusions in the context of the proof search, such that the next attempt can learn from previous failures.

In this section we will firstly describe what a blackboard is and as a second step look at some synchronization mechanisms possible for the datastructures in it.

3.1 Blackboard Systems

The *Blackboard Architecture* was introduced and mostly used in *artificial intelligence* as a knowledge representation architecture. The first program that used this architecture was the Hearsay-II speech-understanding system by Erman et al. [EL80].

The way blackboard systems work is often described [Wei13] by the following metaphor.

> Imagine a group of human specialists seated next to a large blackboard. The specialists are working cooperatively to solve a problem, using the blackboard as the workplace for developing the solution.
>
> Problem solving begins when the problem and initial data are written onto the blackboard. The specialists watch the blackboard, looking for an opportunity to apply their expertise to the developing solution. When a specialist finds sufficient information to make a contribution, he records the contribution on the blackboard, hopefully enabling other specialists to apply their expertise. This process of adding contributions to the blackboard continues until the problem has been solved.

This metaphor emphasizes the key aspects a blackboard system offers.

Modular Solutions Each specialist looks only at the blackboard and decides to write on it if he sees something interesting for him. Hence we can add and remove specialists as we like without dealing with interactions between the specialists.

Therefor we are also independent of the specialized knowledge of one of them, as long as there is a combination of other specialists that subsumes the behavior of that one.

As the word "specialist" suggests one of them is never enough to find a solution. The solution will always be a combination of work done by the specialists.

Multiple Solutions In a complex problem and with many specialists there can be multiple ways to find a solution, depending on the interaction of the specialists. In fact a blackboard system can explore many approaches simultaneously. In this case the fastest solution wins.

Of course a communication between the different solution approaches can happen, because every data is stored visible at the blackboard.

Free Canvas & Readability The blackboard architecture itself does not determine the kind of data stored in it. As the material blackboard it is just a canvas where everyone can write everything.

But of course for the interaction of the specialist one has a priory define a common language, i.e. datastructures and access methods if a specific blackboard is implemented.

As in real life there can be many specialists write something on the blackboard, but if no one understands the other one, they will never find a solution as long as no one can find it by himself.

Information Retrieval The data stored on the blackboard can grow very large. A specialist that wants to provide additional data has to look in this whole set of data for the small set he needs to conclude additional data.

Hence a blackboard always has access methods to efficiently search for many aspects of the data. Often the data is stored many times, each time sorted after a different property of the information.

This way an expert must not perform his own search, but ask the blackboard to give him the most interesting data.

Event-based The specialists work in a trigger mechanism. They look at the blackboard until something is written with which they can work. This event then triggers the expert to update the blackboard.

Therefor a blackboard architecture can be seen as purely reactive, only depending on the change on the blackboard.

Control Mechanism Although the experts work independently of each other, they cannot roam freely on the blackboard.

One image the case that two specialists want to modify the same formula at the same time. In the physical world there is only room for one, but in the computer both might interfere with each other.

For this case some mechanism has to be designed to assign the control to one of the specialists to write.

All these aspects are typical for a blackboard and conclude in three facets, that have to be implemented.

A – The data storage together with its interface, such that we have the common language every specialist has to use while writing on the blackboard and a storage that allows multiple access, such that many experts might work separately as long as there are no conflicts.

B – The specialists itself have to be implemented.

C – The control mechanism has to be designed which determines who can write onto the blackboard.

For the rest of the section we will stick to point A. Point B will be dealt with in section 4 and the control mechanism in section 5.

3.2 Blackboard Synchronization

The blackboard as described before is a big canvas accessible for many specialists at once. All these specialists share the blackboard as their single place to store data. The problem arises how to handle multiple write actions to the blackboard at ones.

If two specialists work on disjoint objects this should be no problem. But as stated in section 3.1 the blackboard consists of multiple search datastructures and the concurrent access to a datastructure always needs to be taken care of especially. Another problem has to be addressed when both write the same data.

The specialists on the other hand do not know of the datastructures stored in the blackboard, hence they are not the ones that can avoid clashes.

In this section some possibilities for the datastructure synchronization are presented and advantages and disadvantages are compared.

3.2.1 Monitor (Blackboard Layer)

A monitor is a high abstraction for the concurrency problem. A monitor supplies a simple lock in which only one process at a time can execute code. The synchronization can be written with a high abstraction of the interleaving of the processes. The points at which the processes can interleave is defined by the user through the operations `wait` and `signal`.

A monitor would hold a simple solution to the problem and it would be easy to write an access that allows multiple access to the data structure at once, as long they are reading or accessing different areas in the blackboard.

Since Java 1.5 a fair implementation of monitor is supported in the form of `Condition` variables and a `ReentrantLock` that mainly orients on C monitors. This monitor mechanism would support strong fairness (in fact it supports first-come-first-serve) instead of no guarantee at all, but lacks the syntactic sugar of the Java's monitors.

This approach would use unsynchronized data structures within the blackboard and synchronize in the blackboard.

Pro:

- The solution is easy to implement. For the blackboard it would be a simple readers writers solution.
- The datastructures itself do not have to be synchronized. Adding new datastructures to the blackboard is no different then in the sequential case.
- It is easy to verify safety and liveness properties, i.e. correctness of the algorithm, for we can use the guaranteed liveness and safety properties of the monitor.
- Monitors are supported in the standard Java library, hence are already included within Scala. No external program has to be used and there are fewer error sources.
- Computation time to verify which process enters the critical section would be small.
- Java `Conditions` are fair and therefor provide the possibility to prove completeness of the system.

Con:

- The blackboard will be locked extensively, i.e. a write to a data structure locks it for any other read or write, but in reality it could be possible for other read operations to access other parts of the data structure with no conflict or even write is possible without conflict. A top level approach would therefor loose valuable computation time with suspended processes.
- Most processes will rewrite a huge part of the formula set the whole time we will get a high amount of write locks which will result in a nearly sequential behavior. The benefit of the concurrent approach would degenerate to a random execution order approach. This is also a nice approach, because we can use machine learning approaches trying to find the optimal order, but with a less strict locking we can still do this and compute in parallel nonetheless.

3.2.2 Monitor (Datastructure Layer)

In comparison to the blackboard layer not all datastructures have to be locked completely. In the top level approach we have one lock for every datastructure,

hence can not update more than one at the same time. A first step optimization would be to move the monitor inside the datastructure. This would allow all datastructures to be manipulated concurrently. If an update to the blackboard always happens in the same order we can then see this update as a pipeline as used in processors.

But the implementation of the synchronization may now differ from datastructure to datastructure and even be more efficient.

Example – Branch locking As an example we will look on a simple solution to synchronize a tree structure for editing and querying.

Suppose we have a binary search tree $\mathcal{T}$. Each node $n \in \mathcal{T}$ has a read and write variable r_n, w_n assigned in the monitor. If a query is executed on $\mathcal{T}$ then it happens as in the sequential case. The difference is that the executing process will enter the monitor for each node and check if no process attempts to write the node, meaning w_n is false. If this is the case he increases r_n and then executes the sequential code in the node. Since every query process works this way many of these can execute in parallel. If they exit the node, they decrease r_n.

Editing works in the same manner. Except the process sets w_n to true and waits until no process is reading the node, meaning $w_n = 0$. Then he makes the changes to the node and releases the lock, advancing his work to the next node. If an action has to take multiple locks at once, he does so by taking them all at once inside the monitor. This way there cannot be circular dependencies that will lead to *deadlocks*.

This simple locking already allows multiple reading and also writing, as long as the writing happens on independent nodes of $\mathcal{T}$.

The data structure can handle access more freely, but on the other hand the blackboard has another problem with this kind of locking. Faster searching for various properties of the formula set uses not one search datastructure but many. If each data structure locks for itself a modify for the same set of data can leave an inconsistent state of the whole blackboard.

Problem – Inconsistency Let P_1, P_2 be two processes that want to modify some data x at the same time. The data x is stored in T_1, T_2 for two different properties.

If P_1 and P_2 now race for the update of x to either y or z they may outpace each other.

In a specific scenario P_1 might be first, updating x to y in T_1. The slower one P_2 updates x to z, such that z is in T_1 the new value. The next time P_2 gets faster and passes P_1, leading to a situation where y was inserted last in T_1.

We end up in a situation where in T_1 z is saved and in T_2 it is y. The set of data is inconsistent, since either one or both should be in T_1 and T_2.

Pro:

- Local solutions have a higher throughput than top level locking.
- Multiple read or write operations can accesses at the blackboard at the same time.
- It is still easy to prove completeness due to fairness properties of the monitor.

Con:

- Extra synchronization for modification of the same element is necessary.
- Standard datastructures have to be reimplemented with locking for most implementations do only top level locking.
- Soundness is an issue, since a race condition as mentioned above could violate invariants.

3.2.3 Software Transactional Memory

Software Transactional Memory (STM) is a mechanism analogous to transactions on databases. A sequence of read, write and modification of shared variables can be put inside a transaction. In the semantics we can view a transaction as a sequence of code that can not be interrupted, i.e. we can view all our code inside transactions as single threaded. This is a great feature because correctness proofs will be not much harder than for sequential programs.

For a transactional system there can various implementations. They reach from code reordering (in the case of Scala and Java byte code reordering), locking (for example the infamous two-phase-locking) to lock-free algorithms.

The implementation reflected in more detail in this work is CCSTM [BCO10] which implements lock-free algorithms for STM. In the case of a lock-free algorithm one process may fail if the set of read variables has been manipulated before the commit.

As in section 3.2.2 this approach has a higher throughput than toplevel locking, because two processes with disjoint write and read sets can operate without conflict.

A major downside is that a failing process may fail infinite times, such that we are not guaranteed that an agent will finally perform an action. This way we can not satisfy completeness. At last because we are only interested in transactions on the blackboard it is of no interest that I/O access can not be rolled back.

One key feature of CCSTM is its guaranteed strong isolation by the type system of Scala. Hence no extra computation time is needed but isolation is checked at compile time. For this each variable used has to be wrapped in a `Ref` class and we have to rewrite any data structure we want to use.

Pro:

- Higher throughput than top level locking.
- Hardly any extra coordination for synchronization.
- Easy proofs for soundness.

Cons:

- No guaranteed completeness.
- Processes may starve in failing.

3.2.4 Lock-free Datastructures

Lock-free data structures can be viewed as transactional memory. The Lock-free data structures meant here are CTries implemented by Aleksander Prokopec, Nathan Bronson, et al. [PBBO12]. The datastructure handles multiple access by generating snapshots of a trie's branch for each accessing process. The results are stored as a DAG (directed acyclic graph) at the node the processes split. The information of the trie can always be reconstructed using the DAG. For the trie not to grow to big, the DAG Snapshots are consolidated from time to time.

This approach has much in common with the internal blocking datastructures. Where the blocking datastructures lose time in the blocking state, the lock-free data structures lose time in the growing size, recapitulation in the following steps and the consolidation. In neither the lock-free case nor the blocking case there occurs a problem if two processes work in different areas of the trie.

But as the internal blocking datastructure this one also poses problems in the multiple search instances of the formula set and needs extra handling.

Pro:

- CTries are already implemented and part of Scala standard library.
- CTries have a high throughput and are at least on par with the other possibilities.
- Since the implementation is already proven to be correct we can use it to proof soudness and completeness.

Cons:

- Needs extra locking for modification of the same element.
- Parallel writings create versions of the nodes that have to be merged at some time, consuming ressources on this end.

3.3 Chosen Synchronization Mechanism

The decision for the synchronization is based on some the following properties we want from the blackboard.

Extendability: Through the development and the enhancement later on, it is necessary to add or change datastructures in the blackboard.

Independent access: If different areas are accessed, two specialists should be able to compute in parallel if they do not conflict each other.

Consistency: With many datastructures the blackboard should be able to keep all data stored consistendly.

The second point prohibits us to put all datastructures behind a global lock or inside a monitor. The search could not be performed concurrently but only sequentially. This would be a big downside, because search is by far the greatest factor for the runtime in ATPs. If this would be the bottleneck of the system, we would get virtually no improvement compared to a sequential ATP.

The first desired property forces us to step back from a global synchronization solution for the datastructures. If Leo-III would use a global synchronization, not necessarily one big lock, the complete mechanism has to be changed each time a datastructure is added.

Since we cannot use a global synchronization, the burden of synchronization is shifted to the programmers of the datastructures. Looking at the last section, this is not that bad, since for a specific datastructure a faster synchronization can be found, compared to a general approach.

Using local synchronization leaves us with the problem of consistency of the datastructures. As we explored, the consistency can be violated if two specialists try to modify the same data at the same time.

The means of consistency will be burdened on the communication between the specialists that are the agents in Leo-III. Thus we will explore this problem in more detail in the next section.

As already mentioned, this thesis will mostly cover the agents for Leo-III. We see that for the blackboard virtually nothing is done or implemented, but a great amount of decisions concerning the blackboard have been done.

Since these decision have a huge impact on the agent system of Leo-III, they are covered at all and first.

4 Agents

The agent-based approach goes back to Thomas Schelling [Sch71] who first thought of the agent-based approach. From the time Schelling simulated the agents on paper and with coins agent-based systems have come a long way

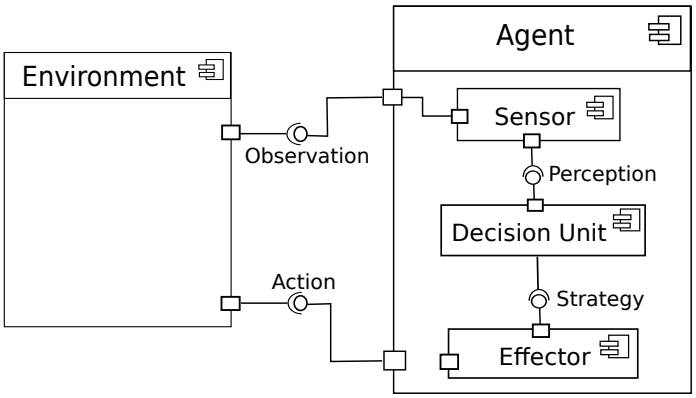


Figure 4.1: Abstract component diagram of an agent and its connection to the environment.

with research combined from game theory, sociology, complex systems and evolutionary programming.

Schelling and many others' work concentrated mostly on *agent-based models*, which focus on the interaction and understanding of the emergent properties of many agents working together. Opposed to this more philosophical approach *multiagent systems* only use an architecture of agents to solve a problem.

Both areas are closely connected, but their goals are different. For we are interested in theorem proving and not understanding the agents themselves, we will first and foremost concentrate on the *multiagent systems*. Therefor we will from now on stick to the term of *multiagent systems*.

This section will firstly motivate the agent-based approach and define some terminology. Then the design for the *multiagent system* of Leo-III will be described and lastly some implementation specific details about the agent execution is shown.

4.1 Multiagent Systems

An agent is an autonomous subsystem or routine situated in an actual environment and works by observing the environment and changing its surroundings [Wei13]. As we aim for an multiagent system (MAS) the agents are characterized through a partial view on the environment or a limited ability to modify it, such that a solution can only be reached by cooperation of many agents.

In theory in a multiagent system all agents have to cooperate to achieve the goal, hence we do not want one agent that can solve the problem singlehandedly. Because we use a blackboard architecture the agents for Leo-III always see the complete environment and therefor the agents should have by definition only limited abilities.

Definition 4.1. (Agent)
An *agent* is a computer system that is *situated* in some environment and that

is capable of *autonomous action* in this environment in order to achieve its delegated objectives. ⌟

We note that an agent consists of at least three components (see figure 4.1)

(1) A *sensor*, allowing the agent to perceive its surroundings.

(2) A *decision unit*, allowing the agent to decide on the strategy based on its perception of the environment.

(3) An *effector*, allowing the agent to manipulate the environment based on the strategy chosen.

This concept of a agent is fairly vague and can be seen as the common ancestor of all agent types. Before looking at the global architecture consisting of many agents, one has to look at the different agent-architectures that are based on the way the agent deals with the environment and the way the agent itself makes decisions.

We can roughly distinguish between three different kinds of agent-architectures [Woo02].

Reactive Agents make decisions only trigger basedly. They act on their perception alone, do not save knowledge about the world and have no decision procedure. In their behavior they work as human reflexes. If a human perceives a pain in a limb, he will instinctively move the limb to the center of the body. There is no thinking about the action. The perception triggers the action spontaneously.

Adaptive Agents gain knowledge about the world through their perception and save it. This way an adaptive agent is able to make decisions based on their current perception and their gained knowledge.

The knowledge is saved in suitable datastructures and can be changed depending on the perception.

For example a human can glean information on the current weather. He can remember the weather of the last days, he knows the current season and on the perception of the sky he can make a prediction on the weather today and e.g. take an umbrella with him.

Cognitive Agents are on the other end of the spectrum. They do not act on their perception, but first interpret their perception into their own saved model of the world.

Based on the model a cognitive agent has of the world he than starts to take action. Many autonomous robotic systems used this approach, where they first build up a map of the surroundings and then start to plan a route to take, based purely on the virtual map.

A human uses such a behavior, just as the robot, if he walks through his house in the dark. He cannot see, but in his mind he can remember

where everything should be. He acts purely on his internal map of the house. Of course this approach has difficulties if the model of the world is wrong. In the case of the human for example if someone has moved a chair. The internal map has the chair not in it, such that he will most likely trip over it.

There are different and finer classification, for example by Russel and Norvig [RN03]. Since the inside of our agents are not known and made up by the developers of the specific agents we stick to this simple classification. In this case Leo-III agents can be reactive, adaptive or something in between.

Because the functions of one agent are mostly limited, many agents are needed for a system, that can actively solve tasks. This leads to the definition of a *multiagent system*.

Definition 4.2. (Multiagent system)
A *multiagent system* is a system of multiple agents in which no single agent can solve the global objective by himself. ⌟

The definition for MAS is fairly vague. The only information we can obtain from the definition is that a MAS is not a system in which we plug in some individual systems and let them compete.

There are many possible architecture to employ a MAS and one of them is the *blackboard architecture* described above. But the MAS is not limited to this architecture.

Belief-Desire-Intention Architecture The most famous architecture in MAS is the belief-desire-intention (BDI) architecture [DLG+04, Woo02]. In BDIs the agents declare a goal they want to achieve. Based on their believes of their perception they will act in an intended way. The architecture is slightly different from the blackboard architecture since all agents have their own belief database that will be manipulated. Except from the beliefs the desired actions build a database on their own to generate an action. Lastly the generated action is filtered and checked if they met the intention.

Layered Agent Architectures As the name conveys the agent are arranged in layers visualized in figure 4.2. In layer architectures [Woo02] the agents do necessarily rely on the sensors, but the agents are chained. One output of an agent is the input for another agent.

Agents in this context work similar to a *filter* in a *pipes-and-filter* architecture as visualized on the right side. In the left side only the first agent is situated in the real world. The agent pipe the information first upwards and then the result is written back down to the first agent which can communicate with effectors.

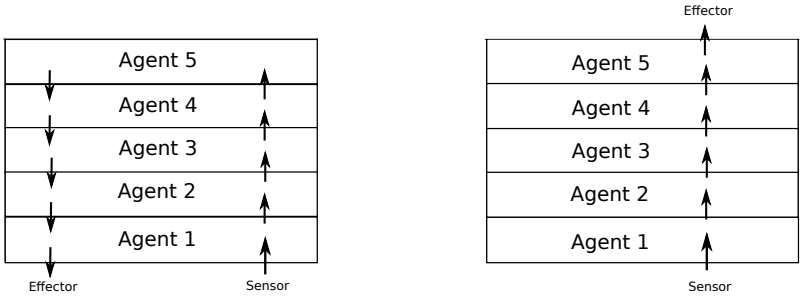


Figure 4.2: Vertical layered agent architecture

4.1.1 Agent Communication

When agents interact with their environment it often happens that two agents want to perform contradicting actions.

For example we are in a robot *rovie* searching life on the mars. Two agents in *rovie* view the surface through a sensor. Agent A is responsible to search for water-like objects. Agent B searches for intelligent life and structural indicators for it.

Both have a problem if A sees a block of ice to the left and B sees house to the right. Both action the agents want to perform contradict each other, since A wants to go left and B wants to go right.

For humans it would be natural to start arguing where to drive. This is also a choice made in many MAS.

For an argument in this context three things are needed. First and foremost the agents have to be able to communicate. We do not have to specify if they communicate directly or through a negotiator, there only has to exist a channel. If A cannot talk to B, their only option would remain to fight over *rovie* by speaking at the same time to the engine. This would result in particular no behavior, because one action would negate the other or even worse destroy the engine.

Next the agents have to define a form of currency. To decide on one of the possible executions, both have to emphasize the value of their action. Without currency it is hard for an external negotiator or the other party to understand the value. Aside from the currency A and B must posses resources in this currency.

The last thing is to decide on a mechanism that determines the decision made, if both declared their values and commitment to pay.

In section 5.1 a theorem is given, stating that scheduling processes without currency and payment, cannot be optimized.

For *rovie* we can think of a direct communication. A declares, that the block of ice could contain biological life and he is therefor willingly to pay 100

units. B declares that the house looks like a rock and is therefor only willing to pay 50 units. They both decide that A can drive *rovie* to the ice block by paying 50 units.

The communication and decision finding has been heavily explored through *game theory*. Since the communication of the agents decides on the access to shared objects, in *rovie*'s case the actuators of the tires and the engine, this communication will act as the scheduler for the blackboard. In the remainder of this section, the internal structure of the agents for Leo-III will be discussed. The communication and game theoretic background will be tackled in section 5.

4.2 Leo-III Agent

Before describing the design for the MAS of Leo-III it remains to describe the general concept of an agent for Leo-III. In section 4.1 we introduced three types of agent-architectures.

Since Leo-III is designed to be an extendable platform through the agents, we cannot determine every structure the agents might take, but there is a general concept. The *blackboard* approach introduced in section 3 is designed to share each and every information. Agents should therefor have no internal datastructures to keep track of knowledge.

If this approach is used, we can see the agents as *reactive agents*, because they make decision purely on their perception of the world, in this case the blackboard. On the other hand the agent can keep track of their knowledge by saving it in the blackboard. Hence their behavior resembles that of an *adaptive agent*.

Since both these architectures are represented the desired Leo-III agent has an architecture in between these two. This keeps us with the task to define in which way his *perception, decision and action* unit should work.

Through this work, three major approaches were tested of which the last one was taken for Leo-III.

4.2.1 Simple Processes

The first attempt was to design an agent simply as a process. The action the process takes is to perform the loop in figure 4.3.

The agent is active in the way that he himself waits for a condition to be fulfilled. In the very first approach this was in fact an actual locking algorithm, which iterated over the blackboard every time some data was inserted as can be seen in the left hand side of figure 4.3. After an update of data the blackboard informs through a signal the change. Every agent was either waiting for the change or is currently executing. If he waits he wakes up and searches for the update. If he was still working he will see the new data if he finishes the work.

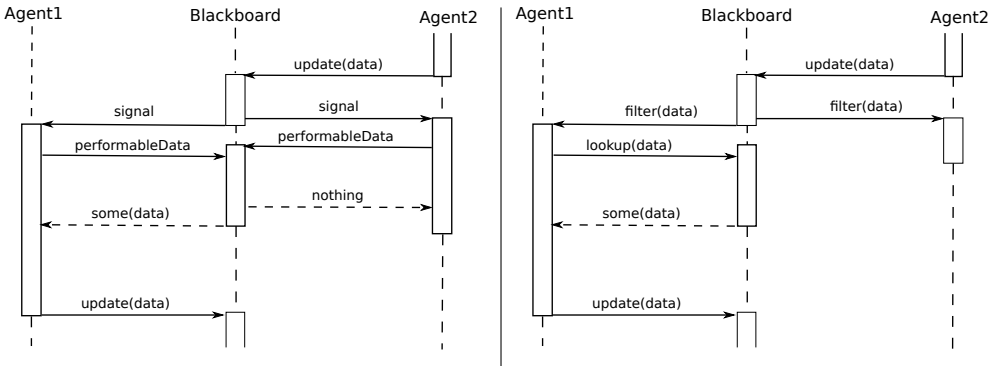


Figure 4.3: Workflow for an agent designed as a simple process

In our example only Agent1 can perform, but Agent2 cannot. Then both make a lookup in the blackboard, whether there is something to operate on. Only Agent1 finds something and updates the result of its computation in the end.

This approach was quite slow, since the agent did not keep track of the changes that where made, such that he iterated over every data for each change. A solution that survived into the final version, is to define a filter for an agent. Each agent can thereby be informed more explicitly, what has changed and with this as a hint perform a faster lookup in the blackboard as can be seen in the right hand side of figure 4.3. Here Agent2 can already see on the updated data that he has nothing to and stops. Agent1 still performs his lookup, but this time he has more information, namely that only something compatible with the updated data should lead to a new result.

This approach is the most direct one. It requires virtually no architecture in this case MAS and was a good design for first tests. The biggest problem arising, using this kind of agents, is the lack of synchronization support. In section 3.2 some of the blackboard possibilities are explored and the decision was to make the datastructures themselves save for multiple access and not for consistency, as described in that section.

4.2.2 Software Transactional Memory

As described in section 3.3 the datastructures in the blackboard are synchronized each one for themselves. This burdens the task of conflict resolution to the MAS and the agent implementation. On the other hand Leo-III should provide a general architecture to extend the prover. Hence we cannot burden the implementation with the synchronization.

A Leo-III agent should be of the kind that he can act as if no other agent exist in the world, i.e. the agent is isolated. His action should be atomic in the sense that either all his changes are written to the blackboard or none. If two processes update formulas at the same time the resulting state has to be consistent as we already explored in 3.2. Lastly if a process is done writing its

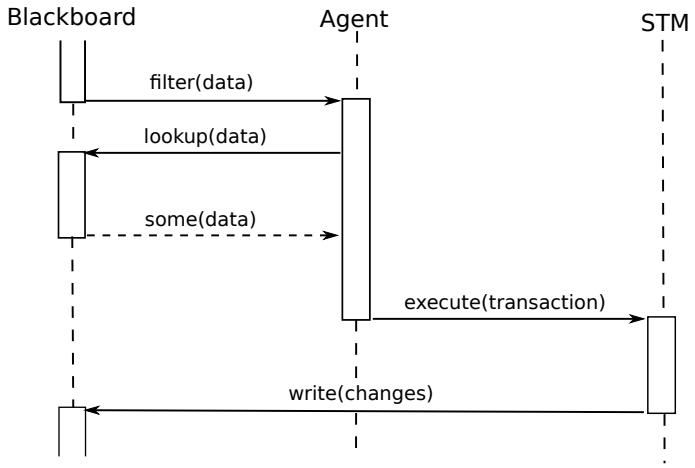


Figure 4.4: Workflow for an agent designed with software transactional memory.

action to the blackboard the change cannot be undone so the action has to be durable.

All these properties for the actions of an agent are the same as for a transaction in database management systems. This is evident since the *blackboard* itself is a specialized database.

Since we need all the features of a transactional system, the first attempt evaluated in this context was a system of *software transactional memory*.

As can be seen in figure 4.4 the overall workflow is still the same as on the right of figure 4.3, but the execution and write back into the blackboard are now capsuled into a transaction and executed through the software transactional system.

This approach was not taken due to two reasons. Firstly to write an agent with transactions has proven itself to be difficult, since the syntax is a burden to the programmer. The desired agent for Leo-III should make it only necessary to write the behavior of the agent. The programmer should not bother with synchronization mechanisms or possible other agents in the systems.

The second reason is that the *transactional memory* allows to execute the actions in parallel, but we have no control which actions should be taken preferably. Then again conflicting transactions should maybe never executed if in the new context the action could no longer be applied.

One thing we learned from this design was that a transactional design solves mostly all problems we burdened on us by using a local synchronization in the blackboard. On the other hand we see that the use of transactions gives us the opportunity to separate the agents generation of an action from its execution.

In the last approach we will hence concentrate on two things. How to hide the transactional behavior, a way to gain more control over the transaction

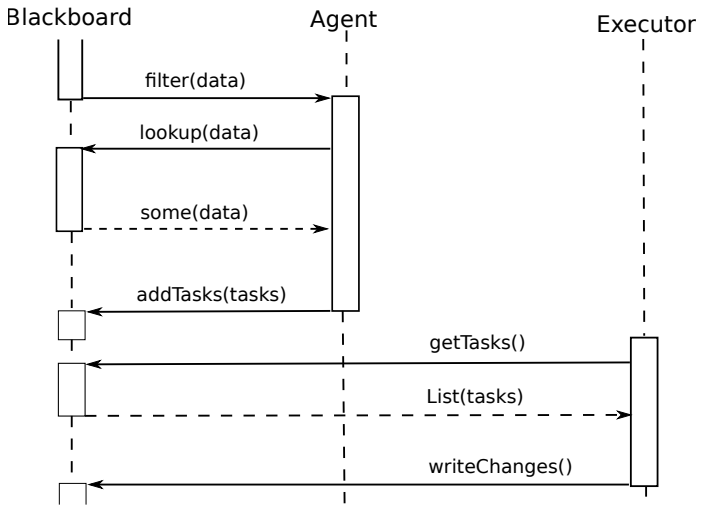


Figure 4.5: Workflow for the transaction agent.

execution and, lastly an approach to implement an agent without an static bound process.

4.2.3 Transactions

The last explored approach uses an own transaction system for the agents. The workflow is designed as seen in figure 4.5. Agent1 does not create a software transaction, but an object called *task* that capsules a deferred action. A *task* is send to a scheduler which is in fact the same as the *software transactional memory*, with the difference that the scheduler can be customized specific to Leo-III.

The designed concept of the execution was explored by Guessoum and Briot [GB99] as *active agents*. The concept behind this idea is to model the execution of an agent as an *active object* [Agh86]. Active Objects are a way to work asynchronously with concurrency. In this context a function does not return the value as a result, but as a promise that sometime in the future the result will be there. The object, often called *future*, has a blocking method to get the result, but the caller does not have to call this method immediately.

In the background a *task* containing the implementation of the function and a reference to the *future* is inserted in a queue. If the *task* is considered for execution by a *scheduler*, the *task* is executed and the result is inserted into the *future*, where henceforth the result can be accessed non-blockingly.

This is the general concept behind this last idea, but with a major difference. In the context of *active agents / active objects* a task is inserted into a queue, where this task has an execution method. Leo-III calls the *run* method

on the agent, with the *task* as an input. This way an agent has more control over their tasks and can, if really necessary, handle internal datastructures.

Implementing a *multiagent system* with this *agent architecture* requires three things.

1. A transactional behavior that allows only tasks to execute that do not conflict each other.
2. An execution engine, that performs the execution of the task.
3. A scheduler, that decides on the next tasks to be executed.

All three are major parts of the *mulitagent architecture* as they decide on the interaction of the agents with their environment.

4.3 Leo-III MAS Architecture

In this section the general concept and design of the MAS for Leo-III is presented. Since we already decided on a *blackboard* based architecture that was introduced in section 3 we will concentrate in more detail on the specifics of the chosen implementation in this section.

In the context of Leo-III an agent is situated in the blackboard as the only environment. Compared to the two other possible architectures presented, there is virtually no structure between the agents. The attention of an agent is directed at the blackboard.

Mainly the agents operate in a global loop like structure, which can be seen in figure 4.6

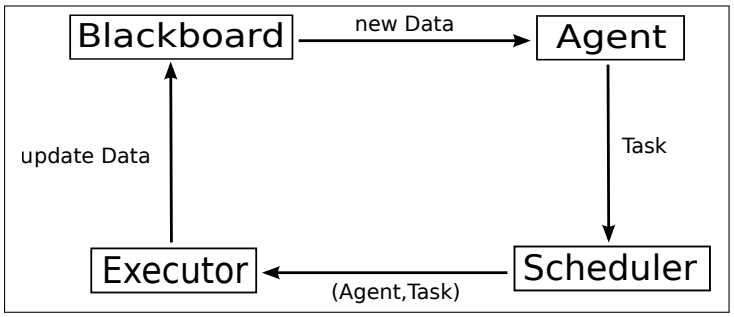


Figure 4.6: General design for the Leo-III MAS

Each agent observes the blackboard. On the arrival of new data, he decides whether he could work with the data. If this is the case, he generates one or more tasks. All these tasks of each agent are send to the scheduler. The scheduler decides which task should be executed such that two tasks never collide, as we decided in section 3.3. If he has decided on a set of tasks, he sends these tasks paired with a reference to their agent to an executor. The executor then runs the agent on the task and is entrusted with the task, so

that the result is written back to the blackboard. This new data closes the circle, since a next iteration of task generation is initiated.

The workflow matches the abstract agent design in figure 4.1. The sensor unit is here directly coupled with the blackboard. This one knows, which data have been added and therefor the sensor unit can be omitted. The newly added data can be directly perceived by the agent. The decision unit is represented by the task generation, whereby the strategy is called *task* in the Leo-III context. The effector is the result of the agent. The action is performed in the executor by writing the data to the blackboard.

With the blackboard structure already given in section 3, we have to deal with the remaining questions. These are *task generation* of the agents, *scheduling* of the task, *execution* of the task and lastly *writing* of the result back to the blackboard.

Task Generation The chosen agent architecture offers a filter method for updated formulas to generate new tasks. The multiagent architecture now has to use this function. Since agents react purely passive, there has to be some part in the architecture to call this method.

In figure 4.4 the blackboard called the filter method after the formulas were updated. The architecture pattern used here is the *observer pattern*.

In the *observer pattern* an object can register itself to a certain type of events. The pattern is often used in interactive programs, e.g. graphical applications. In there an event could be a click on a button. In this program a variety of objects have to be updated, not all explicitly known to the button. In the graphical application we could imagine a button that switches between two modes of operation. First of all, the interface has to be switched to the new mode. Then the new mode might require a reset of a number of datastructures.

The click now notifies all registered objects, in this case the interface and the datastructures, about its activation. The button does not know, what changes have to be applied, but each of the objects does and the button has references to these objects.

The pattern is easy to update actively by runtime. For example if an undoing tool is added, the tool needs to know of the mode change to make an undo step available. For the button nothing changes, except another observer registers and will be called if the activation happens.

In the case of Leo-III all agents are *observers* on the blackboard, interested in a data update. Thus the architecture is allowed to add agents without changing the implementation of the blackboard. The *observer pattern* even allows to remove or add agents during the execution of the program. This way specialized provers can be added if the analyzed context emphasizes that they could be beneficial.

The one thing that requires more thought than in the button example, is the notification of the observers. The button had the advantage of a singular indivisible action that happened. As soon as the click occurred, the graphical interface and all other datastructures could be informed.

In the context of the *blackboard* the *agents* are interested in an update of data. Since the update of data always happens inside of a transaction, the notification cannot be send each time data is updated in the blackboard. This is due to the *atomicity* of the transactions, since a transaction will update sequentially in the blackboard. If the notification happens after each update, the filter would be applied to an unobservable state, according to *atomicity*, since one formula of the transaction is updated but another is not.

Hence the notification has to be done coupled with the writing to the blackboard. This problem will be tackled below in the *writing* to the blackboard.

Lastly the tasks which were generated after a notification has been done have to be saved. The agreement in the beginning was that all data is saved in the blackboard. In this case the tasks are saved in the agents and the blackboard has a method to access the tasks through the agents. The explanation will be given in section 5.3.

Scheduler This is the second time the scheduler is mentioned. The first time in section 3 the scheduler was an abstract routine that selects which specialist may write to the blackboard. After deciding on an (multi)agent architecture, there are more requirements for the scheduler.

The chosen agent architecture demands that the scheduler should be a transactional system. Hence he has to take care that no two tasks executions may collide with each other. The property of collision between tasks, if their read and write sets intersect, was first formulated by Bernstein [Ber66]. For transactional systems there are many well known possibilities to avoid collision, such as two-phase locking, time-stamp ordering or serialization graph checking[BHG86].

The scheduler will implement a form of strict two-phase locking. Since Leo-III uses a scheduler which makes it easier to implement the two-phase locking, because *one* central instance, the scheduler, acquires all locks and releases all locks at once.

In comparison to the above mentioned methods to implement transaction a scheduler performs slower, since the scheduler itself is not parallelized. Two-phase locking itself requires serial behavior only if two processes attempt to work at the same time, but their tasks are colliding. Otherwise both will run completely independent, hence there is no communication overhead.

In section 3 we already observed that in Leo-III coordination of agents is necessary. The scheduler is thus not only the place where the transaction system is implemented, but also the part of the system that decides which tasks are to prioritize.

The prioritization is an important part in Leo-III. Depending on the freedom of the implemented agents, a vast space of possibilities could be explored. Returning to the mars rover from section 4.1.1, a bad prioritization might lead to a behavior where the agents A and B drive *rovie* to the water and to the stone structure, but never remain long enough at one side for the real research. The rover would be working the whole time, but accomplish nothing.

Execution To this point, the agent can generate transactions, called tasks, submit them to the scheduler and have them selected. The next step is the execution of the tasks. Since the next section 4.4 is dedicated to this topic and the point is self explaining, we will not further explain the functionality here.

Since the blackboard and agents are purely passive, the execution unit is designed as an active process. The execution unit is itself designed as an agent that observes the blackboard for changes in the scheduling. If new tasks are available and the execution agent has capacity left, he will call the scheduling method in the blackboard and execute the tasks.

The general design of the execution agent resembles the *simple process* architecture we first analyzed for the Leo-III agents. Since this agent follows only this simple pattern and is stimulated only if tasks are created, there is no need for a more complex structure. Also he cannot be part of the structure as the other agents, since there would be no *active* execution agent to run his tasks.

Writing After the task was created, chosen and executed, the result still has to be written back into the blackboard.

As all other processes in Leo-III, the processes to write the results to the blackboard are designed as an agent. The data he is interested in are the results created by the other agents. The task he creates applies the updates in the result to the blackboard.

In comparison to the normal agents in Leo-III, his observing behavior is not fed by the blackboard. The execution agent inserts the results directly as *write back tasks* into his internal queue. Since he should be the only one interested in these result, this makes the communication easy.

The design as tasks allows the writing to the blackboard again to be made in parallel. Also depending on the implementation of the execution agent, there are no more resources needed for the writing. All execution resources remain with the execution agent.

Since the writing agent is the only one with a direct access to the blackboard, he himself does not return a result.

There are two things to mention for the *writing agent*. First, aside from the blackboard itself, the *writing agent* is the only part in Leo-III that has to be changed, when datastructures are changed or added to the blackboard. Since he has to apply every update in the results to the datastructures, it is necessary to link the corresponding fields in the result to the datastructures.

This can be done through a map of the fields to the datastructures, but since there is no possibility to see, if some datastructures need special care, we distanced ourselves from this approach. But it can be written at any time if it seems promising to be used on datastructures.

The other part for the writeback agent is to release the locks after the update happened. As mentioned before, the filtering cannot be done in the update methods of the blackboard, since atomicity is endangered otherwise. Since there is no *active* agent in the system, except for the scheduler, the only place the filtering can happen is after the writing was completely done, thus in the writing agent. Hence the writing agent will first write the changes, then release the locks and lastly begin the task generation.

An exemplary writing agent has been implemented in this manner and was used in the tests.

4.4 Execution of Agents

As described in section 4.2 the architecture regarding the execution is modeled as active objects. For uniformity the executor, as well as the process writing the results back to the blackboard, are modeled as agents. Our idea with the blackboard is that every agent stores its data in the blackboard. Hence the executor agent complies this rule. Since most other agents are not interested in this side of the blackboard, the interfaces are divided, such that each agent sees only relevant information for him.

The workflow is as described in figure 4.5. Every time a data is updated in the *blackboard*, it is iterated over all registered agents and a filter is called. The filter internally saves a set of enabled tasks. The *execution agent* is registered to the changes of the task sets. If he has capacity left, he asks the blackboard for a new set of tasks. Internally the *blackboard* collects the tasks of the registered *agents*.

This part of the *blackboard* is called the *SchedulingBlackboard*, since the returned set of tasks should be collision-free and should maximize the revenue of the execution. The implementation of this scheduling method is shown in the next section.

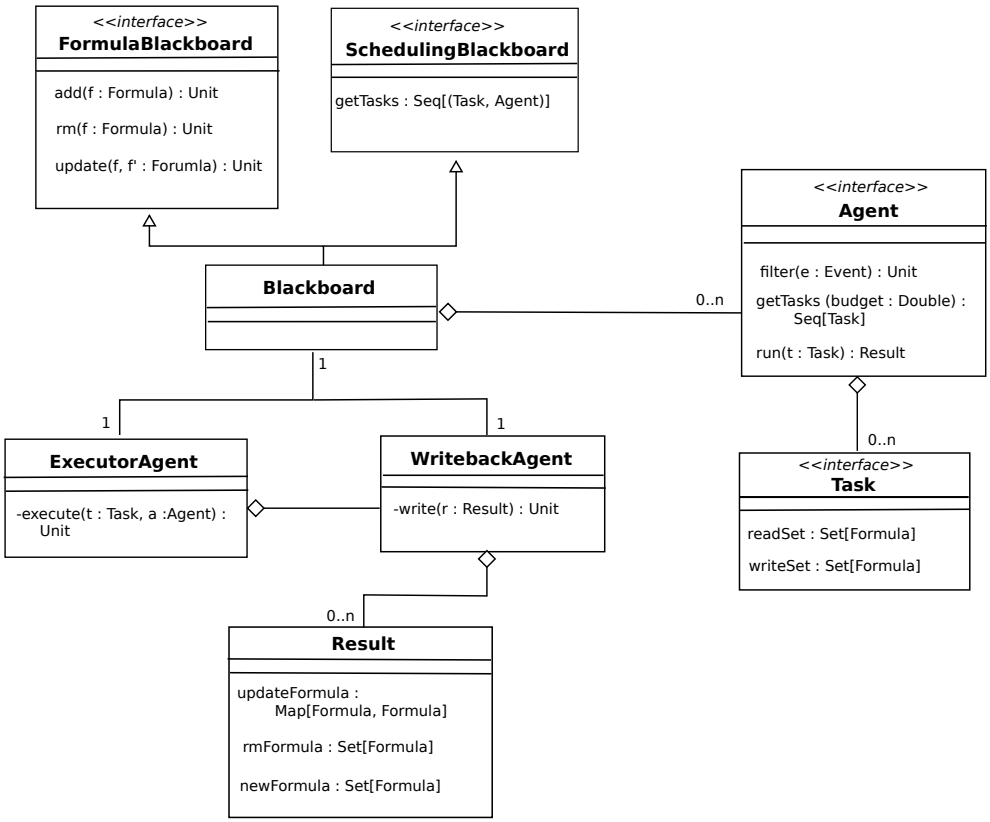


Figure 4.7: Class diagram for the execution structure of Leo-III

If the *execution agent* receives the tasks to be executed, he will execute them in parallel. An *agent*, upon executing a *task*, creates a *result* object. These result objects contain the changes to the *blackboard* that are not yet executed.

These *results* are taken by an *writeback agent* which simple task is to commit the changes to the blackboard.

Executor Implementation The executor internally has to run the agent on the task. Since we already modeled the architecture as an active object [GB99, Agh86] like computation model, it would not make sense to keep a process for each agent and pass them the argument.

This would be inefficient, especially for a big number of agents. First of all, the agents could not really perform in parallel, since the number of cores does not increase with the number of agents we use. Hence in a system with many agents, quite a number of agents lay dormant.

This is even more problematic as they hinder other tasks from execution, due to design transactional system. Secondly, this approach would hinder the active object structure of tasks, since only one task per agent can be executed in parallel, even if these tasks do not interfere.

Luckily in Java there is already a structure of processes implemented that are not limited to one task. The object is an `ExecutorService` that allows the user to define a priori a number of threads which should run in parallel.

After initialization a `Runnable` can be generated, encapsulating the execution of the `Task`. The `Runnable` can be given to the `ExecutorService`, which will assign a thread for execution as soon as there is one available.

The advantage of this implementation is that we can limit the number of threads, hence to not overburden the system that runs Leo-III.

Properties of the Implementation This chosen design and implementation have proven to be very independent of the agent implementation until now.

It is a desirable feature to implement the execution of tasks, rather than that of agents. This way an agent can run in parallel to itself and work that can be done independently, will automatically be parallelized.

If agents were designed as processes, for a parallelization of the agents work, the agent has to be inserted multiple times into the system.

This would contribute nothing new to the system, but another object responsible for handling the tasks.

The use of an executor service is beneficial to the resource consumption, since we can limit the number of threads running without limiting the number of agents.

By implementing the transactional system on top of the agents, a developer of a new agent does not have to concern with side effects or collisions with other agents. He only specifies when the agent is interested to operate, the action of his operation and the returned result.

The interest and the action are always part of an agent implementation. The result objects are the only extra task burdened on the developer. For this he is guaranteed that his execution is always written consistently to the blackboard.

If the *blackboard* is enriched with more datastructures through the development of Leo-III the single changes made to the execution system are the *writeback agent* and the `Result` interface [2].

If new kinds of data are written, they must be tracked in the result. The *writeback agent* must apply all changes to the blackboard, but old *agents* do not have to concern themselves with the new data and act as if nothing has changed.

After describing the execution and design of the agents and describing the general design of the blackboard, the only thing remaining for the execution is scheduling the tasks.

[2]The blackboard interface changes as well, but as long as only additions are made, the agent and execution structure is not concerned.

5 Games & Auctions

In section 4 the general design for the agents to interact with the blackboard and each other was discussed.

The design of the scheduler was to execute only not interfering tasks, but the method to choose these tasks is still open. In the proving context we cannot simply implement a method that has a good response time or a high throughput, as seen in standard scheduling algorithms. We neither can proof hard guarantees as in real-time systems.

The agent approach heavily depends on the right choice at the right time. Each evaluated task has a certain value for the agents in any given context. We assume the agents to be social, i.e. they do not cheat the system. The scheduler should now work in order to always to optimize the welfare of the whole system.

The mechanism that chooses tasks has to satisfy certain conditions. Firstly only non interfering tasks should be chosen. Next, each tasks is rated through a profit function, how much does an agent gain if the tasks are executed. The mechanism should then choose a set of tasks that maximizes the total profit. Lastly, because this is the method the scheduler uses to assign tasks for execution, the mechanism should satisfy certain liveness properties. One of the central ones is that each agent may eventually run a task on a set of formulas.

In this section we will take a look at the general game theoretic foundations for such a mechanism and then give a fast approximating algorithm for the problem.

5.1 Game Theory

This section is mainly based on Nisan et al. [NRTV07].

Game theory deals with the interaction of players with each other in the presence of individual goals each player wants to achieve.

The applications were firstly seen in economics, politics and in law. Later computer science applied game theory as well as biology in their domains.

In this section some foundations are explained which are needed for the algorithm the scheduler uses and will be explained at the end of the section.

Games

Definition 5.1. (Game)
Let $p_1, ..., p_n = \mathcal{P}$ be a set players and $\mathcal{S}_\mathcal{P}$ a set of individual strategies for each player.

A vector $s = (s_i \in \mathcal{S}_\mathcal{P})_{i \in \mathcal{P}}$ is called (pure) *strategy* and $(s \mid s_k)$ for $s_k \in \mathcal{S}_\mathcal{P}$ is the strategy s except the player k plays the individual strategy s_k.

A function $g : S_{p_1} \times \ldots \times S_{p_n} \to \mathbb{R}^n$ that assigns a strategy the payoff for each player is called a *game*. And g_{p_i} is the projection to the payoff of the player p_i. ⌟

Intuitively a game is a function that tells, dependent on the other players, each player what his payoff is.

Example: (Prisoner-Dilemma)
The prisoner dilemma is the standard example introducing games. In the setting two criminals are interrogated, each one in another room. Hence they cannot hear or communicate.

Since the police has virtually no evidence, neither is convicted for the full sentence if no one talks. If one confesses, he sentenced to one year in prison, the other one to four. If both confess, they are sentenced to three years both.

The *game* can be represented in a matrix as seen in table 5.1

	Criminal 1 confesses	Criminal 1 remain silent
Criminal 2 confesses	-3,-3	-4,-1
Criminal 2 remains silent	-1,-4	-2,-2

Table 5.1: Normalform of the prisoner dilemma game.

⌟

In the example a game is defined, but no mechanism on choosing a solution for the prisoners. In the context both prisoners do not know of the others choice, hence they act selfish, which motivates the next definition.

Definition 5.2. (Nash Equilibrium)
Let g be a game for players $p_1, \dots p_n$ and strategies $\mathcal{S}_{p_i}$.

A (pure) strategy s is a *Nash equilibrium*, iff for each player p_i holds

$$\forall s' \in \mathcal{S}_{p_i} \,.\, g_{p_i}(s) \geq g_{p_i}(s \mid s')$$

⌟

Intuitively a strategy is a *Nash equilibrium* if each player individual cannot find a better solution.

Applied to the prisoner dilemma the only *Nash equilibrium* is the case, when both confess, since they get four years in prison if they decide to talk in this situation. The other possibilities are not *Nash equilibria*, since each one gains more profit individually by talking.

Aside from pure strategies games can be made over mixed strategies, where a player does not decide on one strategy but a probabilistic distribution over all his strategies. For this case *Nash equilibria* are defined over the expected profit.

For the two prisoners, they could decide, how much they trust the other one. If they can quantify their trust, they give a probability with which they confess and talk.

Aside from these simple equilibria, there are a variety of equilibria for a great amount of specialized game that can be read in Nisan et al. [NRTV07].

Mechanism Design Mechanism design is a discipline in game theory where some procedure is developed to make a *optimal* decision based on the strategy of a set of players.

Classical examples in for mechanisms are

Elections: The items are the candidates for the election. A mechanism is a function that orders the candidates, such that it corresponds to a single choice of all voters.

Markets: In economics usually a *perfect market* is assumed. That is a reallocation of all resources and money, s.t. the profit of the market is maximal.

Auctions: Generally speaking an auction consists of buyers and sellers. An *auction* assigns the goods from the sellers to the buyers, s.t. the buyers get the most value for their money and the sellers make the most possible profit.

Government: A government has to make decision representing all of the nations inhabitants. This way a government can be seen as a mechanism and the policy of the government as the *single social choice*.

Definition 5.3. (Mechanism)
Let $i_1, \dots, i_n \in \mathcal{I}$ be items and $r_1, \dots r_m \in \mathcal{L}$ individual preferences, where $\mathcal{L}$ is the set of linear orders.

A mechanism is a function $\mathcal{F} : \mathcal{L}^n \to \mathcal{L}$ that decides depending on all individual preferences a *social welfare function*.

A mechanism $\mathcal{F} : \mathcal{L}^n \to I$ that decides on one element depending on all preferences is a *social choice function*. ⌟

This definition still lacks a property of what a *good social welfare function* is. To motivate following definition we will firstly look at a famous paradox.

Problem: (Condorcet's Paradox)
Assume we have three players and three items a, b, c to choose from. The players have the following preferences

1. $a \succ b \succ c$

2. $b \succ c \succ a$

3. $c \succ a \succ b$

The problem in this example is that no matter what order the mechanism chooses at least two players will be dissatisfied with the choice. ⌟

Keeping this in mind, we can work towards a definition of a good choice.

Definition 5.4. (Good social choice)
Let $\mathcal{F}$ be a *social welfare function*.

1. F satisfies *unanimity* if for all $\prec \in L, F(\prec, \dots, \prec) = \prec$.

   Intuitively this means that if all agree on an order the mechanism will choose this order.

2. A player p_i is a *dictator* if for all $\prec_{p_1}, \dots \prec_{p_m} \in L$ $F(\prec_{p_1}, \dots, \prec_{p_m}) = \prec_{p_i}$.

   Intuitively a *dictator* says what to do, independent of the choice of preferences.

3. F is *independent of irrelevant alternatives* if the preference between two items $a, b \in I$ is only dependent on the players.

   Formally this property means, if for all $\prec_1, \dots, \prec_m, \prec'_1, \dots, \prec'_m \in L$ with $\prec = F(\prec_1, \dots, \prec_m)$ and $\prec' = F(\prec'_1, \dots, \prec'_m)$, then $a \prec_i b \Leftrightarrow a \prec'_i b$ for all i implies $asuccb \Leftrightarrow a \prec' b$.

   Intuitively this means that the ordering of a and b in the social welfare does not depend on a third element c.

⌟

The next is the famous impossibility result by Arrow [Arr51], which can be stated as follows.

Theorem 5.5. *(Arrow's Theorem) Every social welfare function over a set of more than two candidates that satisfies unanimity and independence of irrelevant alternatives is a dictatorship.*

A simple proof was given by Ning Neil Yu [Yu12].

Arrow's theorem is famous in economics. Colloquially speaking it states that in a system without money, no mechanism for fair solutions can be found.

If money is involved there is an escape from dictatorship. Each player assigns a value to the items, thereby ordering them. The mechanism can now make a decision by charging the players a few, if their choice was taken.

In this setting each player wants to maximize his value. The mechanisms almost always optimize a global revenue function that ensures to some extend that all players are pleased.

Auctions One type of *social welfare function* that can be optimized with money is an *auction*. We are interested in these specific games, because the scheduler will be based on them.

Definition 5.6. (Auction)
Let $\mathcal{P} = \{p_1, \dots, p_n\}$ be bidders (players) and $\mathcal{I} = \{i_1, \dots, i_m\}$ Items. Each player has a value function $v_{p_i} : \mathcal{I} \to \mathbb{R}^+$ and an budget $b_{p_1}, \dots, b_{p_n}$.

An *auction* (*auction mechanism*) is a mapping of items to bidders

$$\sigma : \mathcal{I} \to \mathcal{P}$$

and a prizing function

$$\rho : \mathcal{I} \to \mathbb{R}^+$$

such that no one buys more than he can afford

$$\forall 1 \leq p \leq n : \sum_{i \in \sigma^{-1}(p)} \rho(i) \leq b_p$$

and the total value

$$\sum_{p=1}^{n} \sum_{i \in \sigma^{-1}(p)} v_p(i)$$

is maximized. ⌟

This is the basic definition of a market with no dividable resources and no sellers. The intuition behind this simple market is that the auctioneer is selfless and wants to maximize the social revenue.

We will go with this definition, rather than a auctioneer that is interested in his own profit or with many sellers, because this fits the market we will define the scheduler on.

One last alteration is that in reality the value function v for the players is not known to the auctioneer. Hence the bidders can start to manipulate the auctioneer by telling false values.

Definition 5.7. (Incentive Compatible)
An auction mechanism is called *incentive compatible* if the bidders refer telling the truth over a lie. □

Note that only allowing the bidders to send the bids according to their own, makes an auction a game. Otherwise there would be no room for strategies.

5.2 Combinatorical Auctions

After we looked at simple auctions and mechanisms this section is dedicated to the problem we want to solve for the scheduling problem.

The problem is already applied to the setting of tasks, whereas the standard combinatorical auction speaks about sets and intersections.

Preliminaries To talk about the solutions, firstly we have to fix some notation.

Definition 5.8. (Tasks)
Let $\mathcal{F}$ be a set of items.

A task t over $\mathcal{F}$ is a three tupel (R, W, V), where $R \subseteq \mathcal{F}$ is the read set of the task, $W \subseteq \mathcal{F}$ is the write set of the task and $V \in \mathbb{R}^+$ is its value.

We define $t^r = R$, $t^w = W$ and $t^b = V$ as shorthands for the elements of the tupel. ⌟

With this we can start define what an "intersection" for tasks means.

Definition 5.9. (Collision)
Let t_1, t_2 be two tasks.

We call t_1 and t_2 colliding if and only if

$$t_1^w \cap t_2^w = \emptyset \wedge t_1^r \cap t_2^w = \emptyset \wedge t_1^w \cap t_2^r = \emptyset.$$

We call t_1 and t_2 non-colliding if and only if both are not colliding.

We introduce the the two predicates $col(t_1, t_2)$ and $ncol(t_1, t_2)$ for both these properties. ⌟

Definition 5.10. (Collision-free)
Let $t_1, \ldots, t_n$ be a set of tasks and $\mathcal{T} \subseteq [n]$ be an index sets.

The subset of tasks defined by $\mathcal{T}$ are called *collision-free*, if and only if

$$\forall_{i \in \mathcal{T}} \forall_{j \in \mathcal{T} \setminus \{i\}} ncol(t_i, t_j).$$

For abbreviation $colfree_{t_1,\ldots,t_n}\mathcal{T}$ or we can omit the tasks if they are clear in the context.

⌟

Combinatorical Auction A combinatorical auction takes place in a setting, where every player wants to obtain a subset of items to achieve a goal. E.g. a mechanic would like to buy a broken car, but only in combination with an engine and new wheels. In this context the mechanic has no benefit in buying only one of the items, because neither works on its own. He would only pay a price if he can get the car, the engine and the wheels together.

In the auction there are many of these players and each one wants a different subset of items. The auctioneer on the other hand wants to maximize his income. That is the total amount of money he gets from each player.

The task for the mechanism for this auction is to assign the items distinctly to all players, such that the income for the auctioneer is maximized, but no item is given to two different persons.

This is the case in the standard combinatorical auction setting. But for our scheduler we have items that can be shared through many players if they do not wish to modify the item. If they want to change the items they have to state that they want it for their own.

Hence we exchange the property that the subsets do not intersect, i.e. no item is given to two different players, with the property of collision-freeness.

Definition 5.11. (Combinatorical Auction)
Let $t_1, \ldots, t_n$ be a set of tasks over a set of items R.

A *combinatorical auction* is a mechanism for the following optimization problem.

$$\underset{\mathcal{T}\subseteq[n]}{\operatorname{argmax}} \left\{ \sum_{i\in\mathcal{T}} t_i^p \;\middle|\; colfree\mathcal{T} \wedge \forall_{\mathcal{T}'\subseteq[n]} \left(colfree\mathcal{T}' \wedge \sum_{i\in\mathcal{T}'} t_i^p \leq \sum_{j\in\mathcal{T}} t_j^p \right) \right\}$$

⌟

Complexity To classify this problem we firstly have to obtain a decision version of this problem. We choose the canonical transformation of a optimization problem.

Definition 5.12. (Combinatorical Auction - Decision Version)

Let $t_1, \ldots, t_n$ be a set of tasks over a set of items R and $k \in \mathbb{R}^+$.
Then $((t_1, \ldots, t_n), k)$ is in the language COMBAUCT, if and only if

$$\exists \mathcal{T} \subseteq [n].colfree_{t_1,\ldots,t_n}\mathcal{T} \;\wedge\; \sum_{i\in\mathcal{T}} t_i^p \geq k.$$

The input for this problem $t_1, \ldots, t_n$, R and k is in the input. We will omit the R, because every item necessary is encoded in the tasks. ⌟

The formulation to this problem suggests a connection to the backpack problem BACKPACK. And in the following it is proven that indeed the problem itself is NP-complete.

Theorem 5.13.
The language COMBAUCT is NP-complete.

Proof 5.13.
In standard manner it is shown that COMBAUCT is in NP and then a classical NP-complete problem is reduced to COMBAUCT.

COMBAUCT $\in$ NP :
The following algorithm is a verifier for COMPAUCT.

Listing 5.1: Verifier for the COMBAUCT problem

```
verifyAuction(t Tasks[n], k : Rational, a : {1...
    n}[r])
  sum = 0
  for(i <- a)
     sum += t[i]
     for(j <- a)
        if(i != j && (intersects(t[i].write,t[i].
           write) || intersects(t[i].write,t[i].
           read)))
           return false
  if (sum >= k) return true
  else return false
```

The method `intersects` can be implemented naively in $\mathcal{O}(n^2)$. Therefor this method runs lies in $\mathcal{O}(n^3)$ and the problem COMBAUCT is in NP.

SETPACK $\prec_p$ COMBAUCT :

For the polynomial reduction we want to show that there exists a function f that runs in polynomial time such that

$$x \in \text{SETPACK} \Leftrightarrow f(x) \in \text{COMBAUCT}.$$

Set Packing is one of *Karp's 21 NP-complete problems* and has the following formulation.

Definition 5.14. (Set Packing)
Let $\mathcal{U}$ be a universe and $\mathcal{S} \subset \{S \subset \mathcal{U}\}$ a set of subsets of the universe.

A *packing* of $\mathcal{S}$ is a subset $\mathcal{C} \subset \mathcal{S}$, s.t. all elements of $\mathcal{C}$ are pairwise disjoint.

A *maximum set packing* is a packing $\mathcal{C}$, s.t. there exist no packing $\mathcal{C}'$ with $|\mathcal{C}| < |\mathcal{C}'|$. ⌟

For the NP formulation of the problem a number k is given and it is checked whether there is a packing $\mathcal{C}$ with $|\mathcal{C}| \geq k$.

The function f now takes every $s \in \mathcal{S}$ and makes a task t_s of it with $t_s^w = s$, $t_s^r = \emptyset$ and $t_s^p = 1$. The number k of sets we want to take is directly taken for the value the auctioneer wants to get. The function performs a simple rewriting of the input is linear.

$x \in$ **SETPACK** $\Rightarrow f(x) \in$ **COMBAUCT** :

Let $\mathcal{C}$ be the packing with $|\mathcal{C}| \geq k$. Then the translated set $\mathcal{T} = \{t_s \,|\, s \in \mathcal{C}\}$ is collision-free.
The total value

$$\sum_{s \in \mathcal{T}} t_s^p \overset{\text{Def. } f}{=} |\mathcal{C}| \cdot 1 \geq k$$

is greater than k and therefore $f(\mathcal{C})$ is a solution to COMBAUCT.

$f(x) \in$ **COMBAUCT** $\Rightarrow x \in$ **SETPACK** :

Let $\mathcal{T} = \{t_s, \,|\, s \in \mathcal{C}\}$ be a solution for COMBAUCT and $\mathcal{C}$ the original set of subsets the tasks in $\mathcal{T}$ were derived from.
From $f(x) \in$ COMBAUCT we know that the value of the set is greater than k, hence we can derive

$$k \leq \sum_{s \in \mathcal{T}} t_s^p \overset{\text{Def. } f}{=} \sum_{s \in \mathcal{T}} 1 = |\mathcal{T}| = |\mathcal{C}|.$$

Therefor $\mathcal{C}$ is a solution for SETPACK.

□

This result poses a problem on the approach to use these kind of auctions to implement our scheduler. On the one hand we want a globally good choice of tasks to be executed, i.e. the ones with the highest price, but on the other hand we do not have the time to compute the best choice.

On the edge between being efficient and finding a good solution one standard approach is to approximate the solution.

Approximation In the following an approximation algorithm for the scheduling problem is given.

Theorem 5.15. *(Combinatorical Auction - Approximation)*
Let $t_1, \dots, t_n$ be a set of tasks over items R and $k = |\bigcup_{i=1}^{n} (t_i^w \cup t_i^r)|$.
Then there exists a mechanism $\mathcal{M}$ that is a $\sqrt{k}$ approximation of the combinatorical auction that runs in $\mathcal{O}(k^2 n^2)$ time. ⌟

Proof 5.15.
For later consideration the algorithm that is implemented in the scheduler is presented first and then proven correctly.

Listing 5.2: Approximation Algorithm for Combinatorical Auctions

```
approxAuction(t : Task[n])
  while (¬ t.empty)
     i = t.maxBy(\x -> x.p * x.p /
       |union(x.write, x.read)|)
     taken = i :: taken
     t = t.filter (\x -> ¬ collide(x,i))
     t = t.map (\x -> x.newReadSet(x.read \ i.read))
  return taken
```

Correctness:
The algorithm naively takes the maximum of the ranking function $(t_i^p)^2/|t_i^w \cup t_i^r|$. Then algorithm removes the colliding tasks and resumes iteratively.

The reason we cannot sort the tasks in advance is that we cannot charge the costs of an element in the read set more than once. If done simply by sorting with this ranking function the approximation could arbitrarily bad for unbounded read sets.

The returned task set is collision-free since every task colliding with a taken one was removed in the iteration. Inductively there cannot be a collision in `taken`.

Runtime Complexity:
The runtime is obviously $\mathcal{O}(k^2 n^2)$ since we iterate in the loop for each element over the remaining tasks, computing intersections of sets that are at most of size k and updating the read sets by intersection.

Approximation:
For the rest we assume the tasks $t_1, \dots, t_n$ are ordered in the way the algorithm touches them. So either an t_i was taken in that order or immediately previous taken task was the one responsible for the removal from the list.

We fix $t_i^* = t_i^w \cup \left(t_i^r \setminus \bigcap_{j<i} t_j^r\right)$ as the value in this context that is the last value assigned to a task by the algorithm.

Let OPT be an allocation with optimal value $\sum_{i \in \mathrm{OPT}} t_i^p$. Let W be the allocation taken by the algorithm with the value $\sum_{i \in W} t_i^p$.

Next define a set $\mathrm{OPT}_i = \{j \in \mathrm{OPT} \,|\, j \geq i \wedge \mathrm{col}(i,j)\}$ for each $i \in W$ containing all elements from OPT that are blocked due to the choice of i since their value was less then the value of t_i.

Obviously $\mathrm{OPT} \subseteq \bigcup_{i \in W} \mathrm{OPT}_i$ since all elements in OPT are either taken in W and collide with itself or they are blocked by some $i \in W$. Therefor it suffices to show that $\sum_{j \in \mathrm{OPT}_i} t_j^p \leq \sqrt{k} t_j^p$ to proof the approximation. This is crucial to us, since the absolute ordering from above can change due to changing values. However in the context of OPT_i the value is fixed.

Reformulating the greedy ordering for all elements $j \in \mathrm{OPT}_i$ we get the equation

$$t_j^p \leq t_i^p \cdot \frac{\sqrt{|t_j^*|}}{\sqrt{|t_i^*|}}. \tag{1}$$

Summing over all $j \in \mathrm{OPT}_i$ we get the equation

$$\sum_{j \in \mathrm{OPT}_i} t_j^p \leq \frac{t_i^p}{\sqrt{|t_i^*|}} \sum_{j \in \mathrm{OPT}_i} \sqrt{\left|t_j^*\right|}. \tag{2}$$

The right hand side can be bounded by the Cauchy-Schwarz inequality.

$$\sum_{j \in \mathrm{OPT}_i} \sqrt{\left|t_j^*\right|} \leq \sqrt{|\mathrm{OPT}_i|} \sqrt{\sum_{j \in \mathrm{OPT}_i} \left|t_j^*\right|} \tag{3}$$

By definition t_j for $j \in \mathrm{OPT}_i$ does not collide with t_i. Since OPT was an allocation all t_j do not collide. The size can be bounded by $|\mathrm{OPT}_i| \leq |t_i^*|$. For elements in the write set this is obvious. For elements in the read set we know at most one task can collide over one variable. If two tasks in $\mathrm{OPT}_i \subseteq \mathrm{OPT}$ would write the same variable they could not both be in OPT.

Next we can conclude that $\sum_{j \in \mathrm{OPT}_i} |t_i^*| \leq k$ holds, since OPT is an optimal solution that can take each element in a write set at most once and t_i^* is constructed, such that it can charge each element in a read set at most once.

Plugged into the first equality holds $\sum_{j \in \mathrm{OPT}_i} t_i^p \leq \sqrt{k}\, t_i^p$. □

5.3 Auction Scheduler

After describing the algorithm to decide the task to be taken, we will take a closer look at the details in the architecture and in the implementation. In figure 4.7 the class diagram of the execution for the agents is described. The scheduler is only a small part of it and the necessary components, which can be seen in figure 5.1.

5.3.1 Implementation

There are four major parts to think of implementing the scheduler. Firstly the tasks have some values assigned. Secondly the agent is allowed to organize its tasks before committing to the scheduler. Then follows the call to the scheduling unit of the blackboard to start the algorithm. Lastly the colliding tasks have to be removed from the lists of the tasks.

We will take a closer look at these stages and why they should be considered.

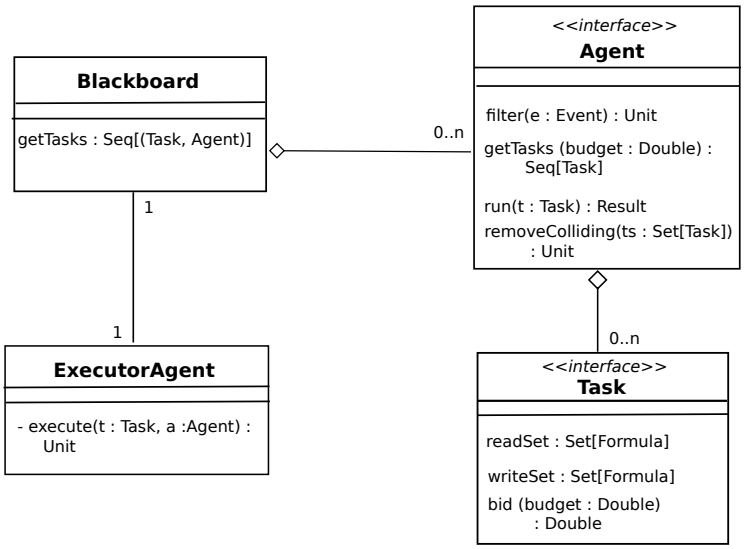


Figure 5.1: Excerpt of the execution class diagram important for the scheduler.

Task bidding In the scheduling algorithm each task must have a value or bid assigned. Therefor it should be no surprise, that *tasks* have a method, that returns exactly the bid the agent makes. In contrast to the scenario of the auction in the last section, the agents have to pay a fee if their tasks is executed. Since no real currency exists inside the system, the scheduler introduces a virtual currency. Each round of the bidding, the scheduler gives an allowance to the agents.

With this money each scheduler can now try to bid for the execution of its tasks.

The money remains in the blackboard. If the money is not manipulatable by the agents they cannot cheat. Another reason for the money to remain in the blackboard is that the blackboard needs to reconstruct the value of the task. Hence he needs access to the assets of the agents at each time.

An upside in the value dependency on the assets of the agents is that a natural form of aging is implemented into the scheduler. If the agent does assign a bid proportional to his wealth, then the task will increase in value until it is executed.

Hence we can ensure that each agent can execute tasks infinitely often if they assign bids corresponding to their wealth.

The ranking itself is not determined by this construct. The ranking can be something that is solely done inside the task, leaving external reasons aside or done through the agent, that can determine the value based on his perception of the blackboard.

What we will look at later is the possibility to determine the value of a task, by coefficients that are stored in the blackboard. This way an external agent can analyze the current workload and goal and change the values of the

tasks correspondingly. It is even possible to use on-the-fly machine learning algorithms to optimize the values.

Considering the difficult assignment of a good value function to the tasks,, the optimization through a defined set of coefficients seems more beneficial. Even if the parameters are learned in the beginning from the outside.

Preselection In the structure implemented is the possibility for the agents to first select their tasks.

The main motivation for this decision is, that the scheduling algorithm is super-linear on a set of tasks, such that more rounds of auction are faster than one big round. The preselection allows the agent to make a choice on the tasks he wants to have executed, before the scheduler comes into play.

This can be beneficial, because he has more information about his own tasks than the scheduler.

A possible problem arising from the shift of the selection to the agents that the *completeness* can be threatened. In the last paragraph it was described, that the allowance mechanism implements an aging mechanism that should ensure *completeness* in the resulting system.

To ensure this property, we are now heavily dependent that an agent may eventually send each task to the scheduler.

This is a burden made on the developer of an agent. In section 6 some abstract agents are introduced that already tackle this problem. A developer does not want to concern himself with this problem, can choose from a set of standard behaviors.

Scheduling Blackboard The scheduling datastructure used by the execution agent to select tasks to be executed is a composition of the scheduling algorithm and the set of agents.

```
getTasks : Seq[Task]
  toAuction <- []
  for (a <- agents)
     wealth[a] += ALLOWANCE
     toAcution += a.getTasks(wealth[a])

  toAcution <- curExec.filter(_.colliding)
  toExec <- approxAuction(toAuction)
  for (a <- agents)
     a.removeColliding(toExec)
  curExec += curExec
  return toExec
```

Listing 5.3: Scheduling algorithm as used by the blackboard

In listing 5.3 a simple version of the blackboard scheduling algorithm, based on the approximation scheduling algorithm, is presented. The implemented algorithm works on pairs to combine the tasks with the agents, such that they can be executed later.

We can observe that an agent is granted his allowance in the beginning of each round. With their current wealth each agent can select a set of tasks he wants to have executed.

With these gained tasks the approximation can be done. In the last step, the agents are informed which tasks where taken.

Of course the *scheduler* is a composition of the *scheduling blackboard* and the *execution agent* that calls the scheduling method and prepares the agents for execution.

Task removal The removal of tasks has two goals. First of all the cleaning of the lists itself.

Imagine the following scenario. One task is selected for execution and another conflicting task is still in the queue. If the executed task writes a formula the remaining task depends on the execution of the later task will lead to an inconsistent state.

Another possibility might be that a task with some data d_1 within the read set takes a long time to execute. Meanwhile the undeleted task with d_1 in the write set is allowed to execute and updates this data to some d_2. Depending on the invariants for the execution of the first task, we are now in another kind of inconsistent state.

For this reason it is beneficial for the task to mark data only as *read*, if it demands no changes during his execution.

The second reason for the task removal is the game nature of the auction. If the agent has no feedback of his strategies in the game, he cannot adapt. By giving him the executing tasks he can change his strategy depending on the result.

A possible adaption is to send less tasks with higher value to the scheduler, if he has not executed for a long time.

5.4 Optimality

For the scheduling method there remain two things to mention.

The auction algorithm has the proven properties, such that it is indeed an approximation for the auction problem. On the other hand, this scheduling algorithm is not optimal or an approximation. As described, the algorithm works in rounds. Every time the *execution agent* has capacity, he demands new tasks. This means on the other hand that the auction itself is a *local optimal solution* to the problem.

It is indeed not a *global optimal solution*, since we have no information on the execution time. If a task is selected, that blocks many variables and takes a long computation time, other tasks might be denied the execution.

This will happen independently of the bid of these tasks if they appear at a later time.

This problem describes another rule the execution agent has to implement. Since the overlapping of rounds might schedule tasks t_1, t_2 infinitely often such that t_3 never is executed.

Imagine that t_1 has x on the write set and t_2 has y on it. If t_3 has both, it might occur that t_1 and t_2 are scheduled overlapping infinitely often, i.e. t_1 starts before it ends t_2 begins and so on, than t_3 can never be executed.

Hence the execution agent has either to wait for all tasks to finish from time to time, to assure that *completeness* can be proven for Leo-III.

5.4.1 Testing & Optimization

The next chapter will describe some agents that are already implemented in Leo-III.

These served to prove that the concept of the scheduler can work. The abstract agents, used in all example agent implementations, implement a queue behavior or a priority queue behavior for their tasks.

Since the tested agents obeyed the rule to send only a constant number of tasks to the scheduler, the workload for the algorithm was never bigger than 20 tasks, mostly having singleton sets for read and write sets.

Hence the scheduling took very little time and was mostly done while at least 3 other agents were still executing. For a real test we would need more agent implementations, a proof calculus and real problems, such that the system can be compared to other agent-based systems and other theorem provers.

What the testing and debugging of the system showed, is that many work, could be avoided if stronger requirements are placed on the agents.

At the moment the scheduler trusts the agents little such that he has to check if the tasks are not colliding with the current execution tasks. He has also to check that no agent wants to spend more money than he has.

Work like this is done currently twice, since the agents do already check these conditions. To leave the workload to the executing agent has the advantage that he can perform faster on his own data, that is more specialized.

It remains to decide whether the burden is in the system on the cost of runtime or the developer of an agent has to maintain more invariants.

6 Agent Implementations

In this section some implementations for specific agents are described to demonstrate the concept of the architecture and show that the agent-based approach can indeed work in many ways.

We will study three major applications of agent-based theorem proving and some minor utility agents usable in many of the applications.

6.1 Loop Agent

Otter [McC90] was the first successful theorem prover that was used even outside the community. Otter is a prover based on a saturation calculus that tries to derive the empty clause. Otter and many similar provers as Prover9 [McC10], Satallax [Bro12] and even Leo-II [BTPF08] iterate in a global loop.

They handle two sets of formulae. Firstly the *unprocessed* formulae U and secondly the processed formulae P. In global manner Otter takes a formulae from U and inserts it into P using inference rules such that the formulae in P are mutually normalized.

These steps, depending on the actual proof calculus, are iterated until the empty clause could be derived.

With the agent-based approach it is possible to write an agent that simulates exactly this behavior. It remains to mention that the loop approach is not close to the agent-based approach, hence is not very comfortable to use.

To implement a Otter Loop, an abstract class for an *loop agent* is provided within Leo-III that can simulate loop like behaviors.

Task: Different then the normal agents the *loop agent* is not reactive, it does not need to implement a filter method for the blackboard.

A *loop agent* has but one task, to execute one iteration and has therefor to implement the `getTasks` method of the agent interface.

```
def getTasks() : List[Task] =
    if isAvailable
        List(getNextIteration)
    else
        Nil

def getNextIteration : Task
```

Listing 6.1: Task for a loop agent

As can be seen in 6.1 the agent does return a single task, but only if it does not currently execute a task. This prevents the *loop agent* to run parallel to itself.

This part is of course not necessary, but if the loop should be parallelized as exemplarily seen in ROO [LMS92b] the agent has to be designed much more carefully.

The abstract method `getNextIteration` can return depending on the agent an empty marker interface. The task would hereby simply trigger the next execution. More preferably the implementing loop would specify the data in the blackboard modified and read in the next iteration.

This way Leo-III could run more than one *loop agent* in parallel, where both are for themselves a simple single-threaded classical agent. But both of them can profits from proof steps or inferences made by the other one.

Run: The loop is break open through the task design. The agent has to remember either internally or through the tasks what to execute next. Therefor a loop agent resembles a state machine from the outside.

The run method of the agent is the body of the loop that will be implemented with one exception. The change will not be written immediately into the blackboard, but stored internally. The internally stored changes can influence the *loop agent*, but have no site effects to the outside.

Result: The result consists of all internally stored changes as they will be written back in one step to not destroy atomicity of the transactional agent.

6.2 Rule-Agent

The rule agent tries another approach then the classical loop by implementing inference rules directly and letting them compete with each other.

This agent is inspired by Volker Sorges approach in Ω-Ants [Sor01]. In Ω-Ants the agent where firstly used to present enabled inference rules to the user in an interactive proof. This agent adepts the idea which was also presented by Volker Sorge, to fully automize the evaluation and application of inference rules.

An important part of the implementation is the biding and selection of active inferences that gets rid of the human interaction.

The rule agent is a parallelization on the clause level as presented in **??**

Task: Each time a new formula is added to the blackboard the *rule agent* has to check if its premise is satisfied for some formulae in the blackboard.

Thus it is necessary for the *rule agent* to implement extra datastructures to check for possible partners in the inference step. If this is not the case the running time of the filter is a polynom with the degree of the inference partners for the newly added formula in the size of the blackboard.

Each tupel of inference partners can be added as a new task. Depending if it is a generating inference or writing inference the formula can be added to the read or write sets of the task.

Run: After the task is selected to be executed the inference rule can be executed.

To check whether the premise of the interference rule is active, often the solution is at least partially computed during the search. Therefor it is a good alternative to write the partial solution into the task such that the run method does not have to recompute this result.

Result: Depending if the inference is generating or writing the agent will generate a `Result` object that has the inferred formulae as new formulae or updated formulae.

$$\text{Res}\ \frac{\mathcal{C}, A \qquad \mathcal{C}', \neg A}{\mathcal{C}, \mathcal{C}'} \qquad \frac{\mathcal{C}, (A \vee B)}{\mathcal{C}, A, B}\ \vee^{+} \qquad \frac{\mathcal{C}, \neg(A \vee B)}{\mathcal{C}, \neg A \qquad \mathcal{C}, \neg B}\ \vee^{-}$$

Figure 6.1: Resolution inference rules

6.2.1 Resolution Agent

For testing the *rule agent* concept has been implementing for propositional resolution.

In figure 6.1 the three inference rules are displayed that are implemented in the resolution agent.

In resolution we deal with a set of clauses in which the elements are connected by disjunction. The two rules $\vee^{+}, \vee^{-}$ define for the minimal signature $\{\neg, \vee\}$ of propositional logic a way to bring the set of formulas in disjunction normal form. The rule *Res* is the resolution rule describing how to combine two clauses.

For this reason clauses have been added to the blackboard in a simple implementation as lists of formula. The routine of the agent checks each newly inserted clause firstly if they match the right cases. For all formulas in the clause matching one of the premises a task is generated for either $\vee^{+}$ or $\vee^{-}$. This computation can be performed locally without looking into the blackboard.

The resolution on the other hand needs another clause $\mathcal{C}'$ that is already in the blackboard. The agent iterates over all clauses and adds each inference partner as a tasks.

The action of the execution can be saved in the task already, since the major part for the inference is the search for the partner.

The difficult part is as in each other theorem prover the selection of the rule to apply. First examples showed that a queue implementation could not find a proof in reasonable time even for small problems.

For this reason a *priority-queue behavior* has been implemented. This allows the *resolution agent* to send the tasks to the scheduler that have the highest value first. The second implementation ranks the tasks by the total size of the clauses. This implementation was already able to proof simple theorems quite fast.

The easy part about the resolution inference is that no mutually normalization has to be done. Hence most of the tasks could be computed without interference using a maximum of the threads supported by the executor.

6.3 Meta Prover

Another fundamental approach uses the parallelism in the simplest ways. Just like Isabelle's *sledgehammer* tool [Pau10] Leo-III can be used to run multiple theorem provers at once, waiting for the earliest or best result.

In this concept an agent is a proxy for another theorem prover. The parallelization here is on the search level. Depending on the set of formulas commited to the prover and if different provers are taken at once this can be each of the parallelizations in this category.

Task: The agent takes the whole set of formulae or a selection of everything that should be send to the theorem prover and everything written into a task.

A possible addition to this behavior is a filtering, if the external prover is suitable for this kind of problem, by analyzing the set of formulae.

In the meta prover there could be some propositional, some first order and a few higher order solver be implemented as agents. For a propositional problem many first and higher order provers perform worse than the purely propositional ones. Hence we only want to run agents with propositional provers. The other way around it does not make sense to use propositional or first order provers for a higher order logic formulae, because they will most likely not solve the problem.

Run: Before the external prover can be run, all formula have to be translated into a format the prover can understand. As in Isabelle it is possible to optimize the translation according to the prover such that he can perform best on the task.

The translation is then send to the external prover via a native call to a script or the executable. The agent waits for the answer that is the termination of the external prover.

As soon as the result is delivered the agent has to interpret it.

In the simplest case the result, theorem or non-theorem, can be added to the context, but it is also possible to interpret the generated proof object.

In an additional step or an additional agent the generated proofs can be simplified, checked or combined with the proofs found by other theorem provers.

Result: Depending on the answer of the prover, the result can be inserted into the blackboard and the proof after both are translated into a internal representation.

6.3.1 TPTP Agent

The first simple solution to implement the meta prover is to call the SystmeOnTPTP introduced in section 2.2.2.

For the call a script supported on the TPTP homepage can be used. The agent has to transform a context of the formulae to the *thf* input format. Then the script can be called through a native call.

Depending on the context higher-order theorem provers as Leo-II or Satallax, first-order theorem provers as Vampire or E or even propositional provers as MiniSAT can be called.

The result given by the call has to be reinterpreted. If the result is a simple *SZS status* then in the positive case the context can be closed, i.e. a proof was found.

In the negative case there could still be something done if a proof object is returned. The data can be interpreted and extra information can inserted into the blackboard. This would also be the case of counter model finder such as *Nitrox* are used.

One major problem if run in parallel to an other mode, such as the inference or the loop agents, is to decide when to launch the TPTP agents. If Leo-III is run solely as a *meta prover* the *tptp agents* can be run immediately.

6.4 Utility Agents

Different to the conceptual agents for the proof calculus as described before, the agent-based approach can use many subroutines usable in all of the above mentioned design concepts.

A variety of search problems that can be parallelized, can be externally implemented as agents to the main proof calculus to decrease the effective search time compared to sequential implementation.

6.4.1 Normalization Agents

Most provers need to bring formulae into some kind of normalform. Not only that these kind of task are highly parallelizable, because they are mutually independent, but the resulting calculus can even begin to work on already normalized formulae beside the remaining normalization.

In their structure they resemble the $\vee^+, \vee^-$ inference rules of the resolution agent. Since they work on formulae and not clauses they can be considered as parallelism at the term level.

Task: A newly added formula is checked if it is in the expected normalform. If this is not the case a task containing the unnormalized formula.

Normally the check for simple normalforms is a local matter so that the rest of the blackboard remains untouched. However, caching the result about the normalform of a formula proved itself to be practical. Otherwise the test for the normal form can be performed numerous times through the process until the execution of the agent.

Run: The transformation into the normal form is performed.

Result: The new formula is marked for an update, since normalization is an updating rule.

For testing purposes an abstract normalization agent has been implemented. The agent accepts a normalization matching a suitable interface. For the test a simplification, negation normalform, skolemization and prenex normalform have been implemented.

6.4.2 Context Splitting Agent

One of the more fundamental approaches to parallelize the proof search is AND-OR-Parallelism [Bon99]. As mentioned in section 2.3 AND-OR-Parallelism replaces the current goal of the proof, by a suitable subset of new goals, all connected through conjunction or disjunction. This produces two new set of formulae, for each goal one set that can be processed in parallel.

Depending on the kind of split one or all of the sub goals have to be proven. Hence a context splitting agent is responsible to decide, when to split on a formula. After this he is responsible to check the subgoals for completion and close the initial goal.

For the context splitting there has to be support in the blackboard. And other agents have to be aware that they can only perform on formulae in the same context. In section 7 two possibilities for the context splitting on the side of the blackboard are presented.

Task: Each time a formula is added the *context splitting agent* has to decide whether he wants to split at this formula. If he decides that a split can be done on this formula he creates a task for it.

Aside from the splitting he has to observe the context if at some point the empty clause is inserted. He is the only one to now when the parent context is closed and thus he has to look for this event.

Run: If the task was a split on a formula he performs the split and remembers for each new goal a new context and its connection.

If a context is closed the task was generated when all subcontexts are closed too. The *context splitting agent* can then start to close the parent context.

Result: In the split case the result introduces new contexts to the blackboard. How this can be done will be explained in section 7.

The closing case will mark the parent context as closed which may trigger the next task for the *context splitting agent*. In the end such a close operation will bubble to the initial context where hence a proof was found.

The structure of the context splitting reminds of a tableaux calculus proof. The difference is that sometimes not all branches have to be closed to mark a node as closed. Since the context splitting should not be a complete tableaux calculus the splitting has to be strongly limited or have a low value for the scheduling.

6.4.3 Learning Agent

A state of the art theorem prover has about fifty parameters where each changes some behavior in the proof search. The amount and interaction between these parameters is not feasible for a human. Therefor machine learning techniques are used to determine the best combination for a problem.

This work is done outside the prover by running the prover numerous times on the same problems changing parameters and comparing the results.

In the agent approach the learning and updating of the already learned could be performed on the fly. One major point were this can be used is in deciding the bids for the scheduler. This is also a task not feasible for human and can change through the state of the proof search.

7 Blackboard Datastructures

In this section we will look at possible datastructures. Through this thesis I concentrated myself on the general design of the blackboard and the agents, such that only general ideas for datastructures exists.

Since most datastructures will be explicitly designed for the proof calculus, there was no way to start implementing before a proof calculus was chosen.

There are some datastructures already thought of to be implemented, but we will look only at two datastructures. The first one is needed if the *context splitting agent* is implemented. The other one is necessary to turn the Leo-III agents into real agents and not only expert systems.

7.1 Context Splitting

In the last section the *context splitting agent* was introduced as a method to implement AND-/OR- parallelism in Leo-III.

The problem arising from this approach is to implement the splitting of the formula set, such that manipulation and search is efficient and the size does not grow to large in comparison to the not splitted context.

The datastructure must support insertion, deletion and update to formulae in a context. Furthermore it must be possible to pose the same search queries as for the non splitted case.

There are two approaches that are possible depending on the datastructure used without splitting.

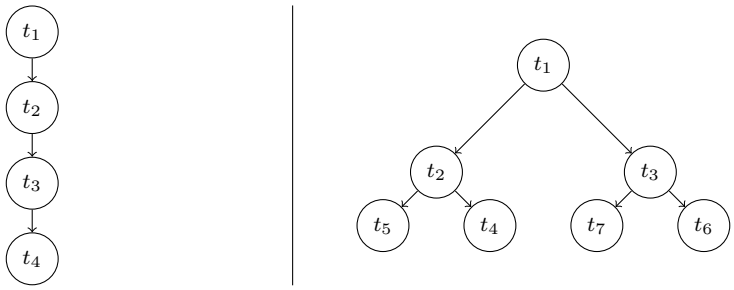


Figure 7.1: Representation of partially and fully persistent datastructures.

Persistent Datastructures Persistent datastructures can be simply demonstrated by revision tools.

The general idea of revision tools is the ability to go back in the history and look at older states of the documents. Persistent datastructures can be categorized by the capability to manipulate the old states.

Partial Persistence: It is possible to look in the past, but changes can only be applied to the most recent state.

The time is fixed in this context, but it is always possible to look at a previous state of the datastructure. The states are connected in a simple path, visualized on the left in figure 7.1.

Since the time is fixed it behaves in all linear fashion regarding this model. In revision tools the example is SVN that allows new changes only to the latest revision.

Full Persistence: These datastructures allow manipulation of past states. The effect of the manipulation is only applied to that state and will not influence changes made further in the future.

The result, as visualized on the right in figure 7.1, is a tree. The nodes of this not necessarily binary tree present states whereby a state is the result of all applications of the path to the root.

The example here is the git branch and checkout tool. This allows the user to go to previous states in the commit history and to start a new timeline by creating a new branch.

Confluent Persistence: The last category allows the user not only to look and change states from the past, but also to reintegrate two timelines.

If two states are integrated, a new state will emerge which status is the application of all the actions of the two states combined.

The example here is the git merge tool. This one allows to create a state consisting of the commits of two branches into one single state.

To use persistence for the context splitting, at least confluent persistence is needed. The following describes, how a confluent persistent datastructure can be used to implement context splitting.

Preliminaries: Let P be the confluent persistent datastructure and $\mathfrak{v}$ be a mapping of the contexts to states in the datastructure.

Invariant: For each context c, the state $\mathfrak{v}(c)$ in P contains exactly all formulas present in the context c.

Insert: Inserting an element f into the context c can be done by going to $\mathfrak{v}(c)$ and insert the element f into the datastructure in this state. The formula f now exists in one state, but not in any context below c. Hence each context c' has to be merged with the new state.

Lastly the mapping $\mathfrak{v}$ has to be updated for each of the new states.

Delete/Update: The same as insertion, except delete/update is invoked on $\mathfrak{v}(c)$.

Search: Querying the datastructure in a context c is possible by querying the datastructure in state $\mathfrak{v}(c)$.

Driscoll and Tarjan [DSST86] have shown a transformation to make any datastructure persistent. The conditions based on the datastructures are that they are pointer based. This means that the datastructure consists of nodes with finite space and pointers to other nodes. To apply the transformation to a persistent datastructure, the in-degree of these datastructures has to be finite, too.

In the case of confluent persistence some kind of merging mechanism has to be supported.

If all these requirements are satisfied, the transformation creates a datastructure that has amortized the same space and runtime complexity as the initial datastructure, but for a constant multiplier.

The confluent persistence could be omitted. A full persistent datastructure is enough if no data is inserted or deleted after it is build up for the first time. This static datastructure can be used for querying only, but there exist another transformation [SB79] for dynamization of datastructures. The downside is that the amortized runtime gets worse once again.

The conditions here are rather strong and even if they can be met, there remains the problem that the manipulation method for the context datastructure has to merge all subcontext states.

In the manipulation step we see that confluence is needed. If there would be no way to merge, the changed formula f would never be conveyed to the substates. The only other option would be to recompute all subchanges again for the new context with the changed f.

The upside of this approach is, that the query time is the same as if no splitting was used.

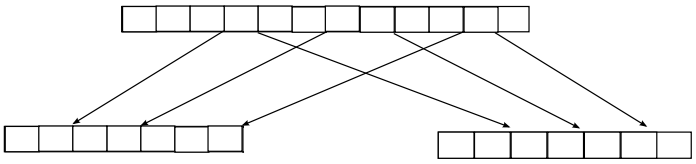


Figure 7.2: Visualization of Fractional Cascading on Tree Structures.

Fractional Cascading We looked at persistent datastructures, because the formulas in shared contexts should be stored only once. The first idea is to mimic the structure of the context split itself and save a tree of datastructures, where each node contains only the formulas that belong explicitly to this context.

On the side of space complexity this approach is optimal, since we save each formula exactly once. Manipulation has the advantage that we only insert in on of the original datastructures which have less formulae, namely only the ones new in the context.

To query the datastructure we have to query every datastructure from the root to the context node. If the sizes are $n_1, \dots, n_k$, and the query time was $q(n)$ for the original datastructure, the splitting datastructure has a query time of $q'(n) = \sum_{i=1}^{k} q(n_i)$.

If the datastructure uses some kind of binary search tree or a sorted list, this query time can be improved to $q''(n) = O(k + q(n_1))$ by fractional cascading [CG86].

The idea is to insert bridges from one list to the connected other constant k elements. This way, searching an element only consumes $q(n_1)$ time in the first list.

In every further list, we can find the the right elements in the next list by traversing a constant number of elements to the next bridge. Thereby we have to visit only a constant number of elements per context we visit below the initial context.

In figure 7.2 the bridges between the lists are visualized. The name "fractional cascading" arises from cascading waterfalls, as the elements of the lists flow like the water along the bridges to the next level.

In comparison the tree approach is much easier and has less requirements than the persistent approach.

In fact, the only advantage the persistent approach bears is that it just relies on the query time of the original datastructure.

The tree approach is slower on the query side if the original query time was sublinear. But as soon as fractional cascading can be applied, the later approach is to be preferred over persistence.

7.2 Message Queues

One of the features of agents compared to expert systems is the ability to communicate and cooperate with other agents.

To this point the architecture used the agents only as experts. The reason our agents still fulfill the definition of an agent is the blackboard architecture.

Regarding communication, people often imagine direct, synchronous communication. Thereby messages are directly conveyed to the receiver. As long known in computer science a synchronous communication is hard to simulate. Both sender and receiver have to be present at the same time and signal the receiving to the other party.

Even at this level the communication just simulates synchronicity, because the message is stored internally. The other end waits for the signal send, when the recipient read the message.

Hence an asynchronous communication is much more preferable. In the blackboard context, this is easy to implement, since a message becomes a new type of data written on the blackboard. If the agent is interested in this kind of message, he will read it and generate a new task.

The design chosen to implement messages in Leo-III orients itself on the mailbox system of Erlang [Arm10]. Each agent has a queue of messages assigned. Other agents can send messages to the agents by inserting them into the queue. If the receiving agent is ready to work on the message, he can take them from the queue and work on them.

The motivation to design the *run* method inside the agents and not in the task was, to be able to design agents just as *Erlang processes*. The communication and acting of the agents looks exactly the same.

Compared to single formulas, the messages can be optimized for the agents. First of all, only a single agent has to be informed when a message is send to him. This approach does not have to be fixed, as another agent might be interested in the information flow.

We can visualize the communication over the blackboard by writing the names of the agents on the blackboard. Each time an agent wants to send a message, he writes the message under his name. Now the receiving agent can look at the message and react to it.

In this context it might occur, that another agent can work with this information, too. For example, the message says to work on a specific subset of the formula. Then another agent could adapt and try not to touch these formulas.

Depending on the existence of such processes, the signaling must be adapted.

8 Related Work

Even though agent systems are under development since the 1970s, there are no prominent automated theorem provers using agents as their architecture.

In artificial intelligence it is more common to use agent-based architecture. An example with a real application is dMARS [DLG+04]. The dMARS architecture was used by the NASA in space shuttles to observe the reaction control system and to monitor telecommunication network systems in real time.

On the side of theorem provers there are even nowadays hardly parallelized versions. One of the most used provers that uses some kind of parallelization is Isabelle/HOL's sledgehammer [Pau10]. Sledgehammer translates the current problem into suitable import formats of external theorem provers, sends them the input and presents the user all results. This philosophy is exactly the same as the intended use of Leo-III as a meta prover through the described *metaprover agents*.

Higher-order theorem provers such as Leo-II and Satallax use first-order and propositional provers as subprogams. They reduce the problems into a suitable form and send these problems to the subprovers. Recently Yves Müller researched a way to parallelize the subprovers for Leo-II [MÜ3].

Combining the context splitting agent and the meta prover agents states an easy way to simulate this idea.

As already mentioned in section 4.2 the work of Volker Sorge [Sor01] has influenced this work in many ways. In comparison to Ω-Ants Leo-III uses the agents to manipulate the formulas directly and not to make suggestions to the user. This way Leo-III's architecture can be seen as a step to make Ω-Ants from an *interactive theorem prover* into a *automated theorem prover*.

9 Further Work

This work concentrated on the architecture of Leo-III. The further work we will look into will be related to the designed architecture.

Blackboard Datastructures This work mainly concentrated on the overall agent architecture and although everything is trimmed to use the blackboard as its main source of information, the blackboard itself does not contain many access methods.

As soon as the first calculi agents are written, there is the possibility to decide on suitable search datastructures and extra information to store in the blackboard.

One already discussed datastructure in section 7 is the α-β-splitting support datastructures, singular.

Other possibilities are search structures for resolution partners of clauses, unification partners and structures for already instantiated variables.

Auction Learning As already mentioned in section 6.4.3 the design of the scheduler leaves room for machine learning techniques to optimize the scheduling behavior.

This can be achieved at runtime in form of an agent, allowing the ranking functions to change according to the current problem and goal.

Another possibility which is used in many theorem provers, is to make a static analysis for a large problem set and perform machine learning externally. This puts less burden on the runtime, but is not as flexible as the agent approach.

Auction Games Auctions are introduced in game theory (see section 5.1). The agents, as introduced in this work, do not play a game. Instead, they only send their value of a task statically to the scheduler.

One possible enhancement is to allow agents to be more responsive, i.e. give them information about who won for which price, s.t. they can adjust their bidding.

In this context it would necessitate a different game mechanism than the *first-price-auction* mechanism chosen at the moment. But a good alternative, the *second-price-vickrey-auction*, costs just as much runtime and prohibits lying agents.

Price of Anarchy As stated in section 5.11 the scheduling algorithm makes local optimal decisions and we analyzed the approximation factor to be $\sqrt{k}$ where k is the number of all used data by the tasks.

First of all the bound is not tight and could be further analyzed. Secondly the local optimal decision can be compared the the global optimal decision. This *competitive ratio* is known as the *price of anarchy*. In many problems this ratio can be bound such that the approximation factor can be used to determine a maximal global value loss factor.

In some applications with certain restrictions on the value functions the equilibrium computed by the mechanism is indeed global optimal.

Hence further work can on the one hand compute the *price of anarchy* and on the other hand search for restrictions to the bidding functions such that there is no price to pay.

Abstract Agent Interfaces The interface for an agent is quite complex due to the methods needed in the scheduling process. For easier use a variety of abstract behaviors can be implemented and used by programmers.

For a start there are already a `StdAgent` and a `PriorityAgent` implemented. The first one inserts all new tasks in a simple queue, whereas the second one places each one in a priority queue.

Both have the advantage, that only the *filter*, *task*, *execution* and the *result* have to be implemented. These are characteristic for an agent as they define

them. The programmer has therefor not to deal with other implementation issues.

Agent scripting engine Another possibility for easier programming with agents is to define a script language for agents. This would allow agents to be written without writing in Scala or changing the system.

For compatibility this scripting language can orient itself on the *tpi* language of the TPTP syntax.

Agents & Testing As soon as some agents are written, especially for the proof calculus it is possible to test the performance of the system compared to other theorem provers.

What is interesting, aside from the overall performance, is the time used for scheduling, synchronization overhead and speed up due to parallel computing.

10 Conclusion

Through the course of this thesis an agent-based blackboard architecture has been designed and implemented.

The designed agents perform adaptive on the blackboard. The implementation divides the agents from the execution itself such that agents do not consume capacity if they do not work as would simple processes do. Rather than that, the action of an agent can be assigned to a process that executes this action.

An upside of this design is that agents can run in parallel to themselves arbitrarily often, without registering the agent more than once.

The design of an action of the agents is done as a transaction. This way an developer of a new agent has only to specify the requirements on which the action can be executed, the action itself and lastly a result.

The complete synchronization is then completely done by the system, hence not bothering the developer.

The first tested agents performed quite well. The synchronization worked good, since no conflicts occurred. The self parallelization also performed quite well.

The first test to be implemented was the normalization agents. Although they could not conflict each other by design, we could already see that they parallelized quite well. The next agent to be implemented for testing was the resolution agent.

It was of no surprise, that an unreflected application of the resolution steps performed quite badly, since many irrelevant clauses were generated. But already a simple order on the clauses, namely the size of the resulting clause, gave a first simple propositional theorem prover that could find a proof relatively fast.

In this context it remains to mention that the architecture itself does not guarantee correctness of the implemented algorithms. The agent architecture presented is only a tool like a *monitor* or a *semaphore* helping the implementer to develop concurrent and parallel programs. But as the other tools the architecture does not guarantee a correct program. Hence the agents have to be designed carefully and not all agents will be able to perform at the same time.

The goal of this thesis was to proof that an agent-based approach for an automated theorem prover is possible. Since the first tests with a resolution calculus did already run and found some proofs, we can say in this sense that the agent-based approach good work.

To test the performance of an agent-based system, it is necessarily to first write a could proof calculus. Then we could start testing the architecture for competitiveness.

But as already shown, the Leo-III architecture allows to implemented classical sequential loop based provers as state-machine programs.

Hence in the further development we can balance the amount of parallelization from either a complete parallelization on inference level, as we stated with the resolution agent, to a nearly sequential proof calculus. In the sequential calculus, there is still the advantage, that easy to parallelize subtasks can be decoupled and run while the global loop still performs.

An example, that is already used in competitive higher-order theorem provers, is to make parallel calls to first order theorem provers, just as it is already done in Leo-II.

Conclusively we can say, that the agent-based approach looks promising for the further development of Leo-II, as the first test emphasizes, that the parallelization does indeed work. Future tests will show, how much of a speed up Leo-III can get through a concurrent implementation compared to other systems.

11 Bibliography

References

[Agh86] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

[And14] Peter Andrews. Church's type theory. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2014 edition, 2014.

[Arm10] Joe Armstrong. Erlang. *Commun. ACM*, 53(9):68–75, September 2010.

[Arr51] K.J. Arrow. Social choice and individual values. 1951.

[BBK04] Christoph Benzmüller, Chad Brown, and Michael Kohlhase. Higher-order semantics and extensionality. *Journal of Symbolic Logic*, 69(4):1027–1088, 2004.

[BCO10] Nathan G. Bronson, Hassan Chafi, and Kunle Olukotun. Ccstm: A library-based stm for scala. In *In The First Annual Scala Workshop at Scala Days*, 2010.

[Ber66] A.J. Bernstein. Analysis of programs for parallel processing. *Electronic Computers, IEEE Transactions on*, EC-15(5):757–763, Oct 1966.

[BHG86] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[Bon99] Maria Paola Bonacina. A taxonomy of parallel strategies for deduction. *Annals of Mathematics and Artificial Intelligence*, 29:223–257, 1999.

[BP14] Christoph Benzmüller and Bruno Woltzenlogel Paleo. Automating Gödel's ontological proof of god's existence with higher-order automated theorem provers. In Torsten Schaub, Gerhard Friedrich, and Barry O'Sullivan, editors, *ECAI 2014*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, pages 163 – 168. IOS Press, 2014.

[BPTF07] Christoph Benzmüller, Larry Paulson, Frank Theiss, and Arnaud Fietzke. The LEO-II project. In *Proceedings of the Fourteenth Workshop on Automated Reasoning, Bridging the Gap between Theory and Practice*. Imperial College, London, England, 2007.

[Bro12] Chad E. Brown. Satallax: An automatic higher-order prover. In *Automated Reasoning - 6th International Joint Conference, IJCAR*

2012, Manchester, UK, June 26-29, 2012. Proceedings, pages 111–117, 2012.

[BTPF08] Christoph Benzmüller, Frank Theiss, Larry Paulson, and Arnaud Fietzke. LEO-II - a cooperative automatic theorem prover for higher-order logic. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, volume 5195 of *LNCS*, pages 162–170. Springer, 2008.

[CG86] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1:133–162, 1986.

[DLG+04] Mark D'Inverno, Michael Luck, Michael Georgeff, David Kinny, and Michael Wooldridge. The dmars architecture: A specification of the distributed multi-agent reasoning system. *Autonomous Agents and Multi-Agent Systems*, 9(1-2):5–53, 2004.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

[DSST86] J R Driscoll, N Sarnak, D D Sleator, and R E Tarjan. Making data structures persistent. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, STOC '86, pages 109–121, New York, NY, USA, 1986. ACM.

[EL80] Lee D. Erman and Victor R. Lesser. The hearsay-ii speech understanding system: Integrating knowledge to resolve uncertainty. *Computing Surveys*, 12:213–253, 1980.

[Fit90] Melvin Fitting. *First-order logic and automated theorem proving.* Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[Fit96] Melvin Fitting. *First-order Logic and Automated Theorem Proving (2Nd Ed.).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[Fre79] Gottlob Frege. Begriffsschrift, 1879.

[GB99] Zahia Guessoum and Jean-Pierre Briot. From active objects to autonomous agents. *IEEE Concurrency*, 7(3):68–76, 1999.

[Gö31] Kurt Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatshefte für Mathematik und Physik*, 38(1):173–198, 1931.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and

Simon Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[LMS92a] Ewing L. Lusk, William McCune, and John K. Slaney. ROO: A parallel theorem prover. In *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction, Saratoga Springs, NY, USA, June 15-18, 1992, Proceedings*, pages 731–734, 1992.

[LMS92b] Ewing L. Lusk, William McCune, and John K. Slaney. Roo: A parallel theorem prover. In Deepak Kapur, editor, *CADE*, volume 607 of *Lecture Notes in Computer Science*, pages 731–734. Springer, 1992.

[MÏ3] Yves Müller. Subprover parallelism in the automated theorem prover leo-ii, 2013.

[McC90] William McCune. OTTER 2.0. In *10th International Conference on Automated Deduction, Kaiserslautern, FRG, July 24-27, 1990, Proceedings*, pages 663–664, 1990.

[McC10] W. McCune. Prover9 and mace4. `http://www.cs.unm.edu/~mccune/prover9/`, 2005–2010.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[NRTV07] Noam Nisan, Tim Roughgarden, Eva Tardos, and Vijay V. Vazirani. *Algorithmic Game Theory*. Cambridge University Press, New York, NY, USA, 2007.

[Pau10] Lawrence C. Paulson. Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers. In *Proceedings of the 2nd Workshop on Practical Aspects of Automated Reasoning, PAAR-2010, Edinburgh, Scotland, UK, July 14, 2010*, pages 1–10, 2010.

[PBBO12] Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. Concurrent tries with efficient non-blocking snapshots. *SIGPLAN Not.*, 47(8):151–160, February 2012.

[RN03] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.

[Rus08] Bertrand Russell. Mathematical logic as based on the theory of types. *American Journal of Mathematics*, 30(3):222–262, 1908.

[SB79] James B. Saxe and Jon Louis Bentley. Transforming static data structures to dynamic structures (abridged version). In *FOCS*, pages 148–168. IEEE Computer Society, 1979.

[Sch71] T. C. Schelling. Dynamic models of segregation. *J. Math. Sociol.*, 1(2):143–186, 1971.

[Sch13] Stephan Schulz. System Description: E 1.8. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Proc. of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*. Springer, 2013.

[Sor01] Volker Sorge. *Omega-ants: A blackboard architecture for the integration of reasoning techniques into proof planning.* PhD thesis, Universität des Saarlandes, Postfach 151141, 66041 Saarbrücken, 2001.

[Sut09] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[Sut11] G. Sutcliffe. The 5th IJCAR Automated Theorem Proving System Competition - CASC-J5. *AI Communications*, 24(1):75–89, 2011.

[vH02] Jean van Heijenoort. *From Frege to Gödel : A Source Book in Mathematical Logic, 1879-1931 (Source Books in the History of the Sciences).* Harvard University Press, January 2002.

[Wei13] Gerhard et al. Weiss. *Multiagent Systems.* MIT Press, 2013.

[Woo02] Michael Wooldridge. Intelligent agents: The key concepts. In *in Proceedings of the 9th ECCAI-ACAI/EASSS 2001, AEMAS 2001, HoloMAS 2001 on Multi-Agent-Systems and Applications IISelected Revised Papers, ser. LNAI*, pages 3–43. Springer-Verlag, 2002.

[YB04] P. Castéran Y. Beriot. *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions.* Springer-Verlag, 2004.

[Yu12] NingNeil Yu. A one-shot proof of arrow's impossibility theorem. *Economic Theory*, 50(2):523–525, 2012.

Index