ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Ingegneria e Architettura
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Extending the 2P-Kt ecosystem with Concurrent Logic Programming support

Tesi di laurea in
Sistemi Autonomi

Relatore
Prof. Andrea Omicini

Correlatore
Dott. Giovanni Ciatto

Candidato
Andrea Giordano

Sessione Unica di Laurea
Anno Accademico 2020-2021

Abstract

It is widely acknowledged that logic programming is very well suited for *concurrency* and a lot of research exists on this topic. However, despite the many contributions laying under the umbrella of concurrent logic programming, only a small amount of technologies survived the passage of time. Survivors, in turn, are tailored on an ancient way of designing (and implementing) concurrent systems.

Accordingly, in this thesis we address the problem of designing and implementing a concurrent LP solution taking advantage of modern concurrent programming facilities such as co-routines and non-blocking I/O. In doing so, we leverage upon the 2P-Kt ecosystem for symbolic AI, following the purpose of enriching it towards concurrency.

Along this line, the contribution of this thesis is three-folded. First, we review the state of the art of concurrent logic programming, providing an overview of its most important aspects and problems. We then formally model the behaviour of an OR-concurrent Prolog solver as a state machine which we adopt to extend the 2P-Kt ecosystem with concurrent Prolog support. Finally, we design, implement and validate an OR-concurrent Prolog solution reifying the aforementioned model into some actually usable technology.

We put a lot of effort in the validation of our solver and then we assess the speedup of our concurrent Prolog solver via comparative benchmark with respect to sequential Prolog. Notably, our solution achieves some good results in terms of speedup and memory usage with respect to the sequential case.

To all grandparents.

Acknowledgements

I thank my university colleague Alessia Cerami for the help given in this work, especially in the correction of the thesis. She also supported and encouraged me and was always there whenever I needed her.

I thank Professor Andrea Omicini for the kindness and sympathy he always shows and for the passion he transmitted to me for the argument of this thesis. I thank Giovanni Ciatto for the great help during the design and the implementation phases but also for encouragement when I was getting disheartened.

I thank my family for the support during the whole university course.
I specially thank my mom, without whom I would not have reached this finishing line. I thank my friends, who listened to me when I needed it, but also for the awesome experience we had together.

Contents

Abstract **iii**

1 Introduction **1**

2 State of the Art **5**

2.1 Concurrent LP 5

2.1.1 Macro parallelism types 7

2.1.2 Implicit parallelism in LP 8

2.1.3 Problems of implicit parallelism 10

2.1.4 Concurrent Prolog 13

2.2 Logic Programming Ecosystems and 2P-Kt 17

2.2.1 2P-Kt overview 17

2.2.2 Resolution 19

2.2.3 Primitives 21

2.2.4 State Machine 22

2.3 Coroutines and concurrency in Kotlin 23

3 Requirements **27**

3.1 Constraints 27

3.2 Goals 28

4 Design **31**

4.1 Abstract design 31

4.1.1 Syntax and Notational Conventions 31

4.1.2 Semantics 34

4.2 Concrete design 41

5 Implementation **47**

5.1 Execution Context 47

5.2 Concurrent Solver 48

5.3 Utils 51

5.4 Concurrent State Machine 53

6 Validation **57**

6.1 Test framework 57

6.2 Metrics and benchmarks 59

7 Conclusions **63**

7.1 Open issues 65

List of Figures

2.1 2P-Kt project structure 18
2.2 Prolog State Machine 23

4.1 Concurrent Prolog State Machine 33
4.2 Main abstractions 42
4.3 Overview of user consuming solutions 44
4.4 Resolution Job and Channel interaction 45
4.5 Channel and Sequence interaction 45

Listings

5.1 Apply method of the ExecutionContext 48
5.2 The solveImpl method 49
5.3 How solveConcurrently handle the solutions channel 49
5.4 startAsync in detail 50
5.5 coroutines launch for each next possible state 50
5.6 How solutions are published when needed 51
5.7 Jvm conversion of a channel to a sequence 52
6.1 Unit test example 58
6.2 Unit test concrete class example 58

Chapter 1

Introduction

The technology for sequential implementation of logic programming languages has evolved considerably since its birth. In recent years, it has reached a notable state of maturity and efficiency. Today, a wide variety of commercial logic programming systems and excellent public-domain implementations are available that are being used to develop large real-life applications.

For years logic programming has been considered well suited for execution on multiprocessor architectures. Indeed research in concurrent logic programming is vast and dates back to the inception of logic programming itself. There is a healthy interest in parallel logic programming ever since, as is obvious from the number of papers that have been published in proceedings and journals devoted to logic programming and parallel processing.

Technology evolved year after year moving from machines with limited resources to modern computers with a multiprocessor and a lot of memory, improving their performances drastically. The software evolved hand in hand with the hardware in all areas, like the operating systems and the programming languages. This evolution raised the abstraction of the technologies available increasing the software development reachable complexity.

There are many types of parallelism and even more techniques to exploit them. Logic programming is very well suited for concurrent resolution because each subgoal can be solved in parallel. Furthermore each matching clause, when more than one is found, can create a separate resolution branch. While the theory is quite

simple, practically many problems arise like, for example, the preservation of the Prolog semantics or the handling of variable dependencies.

Various systems have been developed during the years to support concurrency in logic programming, each one with different features. The problem is that most of them are not supported anymore due to the complexity they have and, furthermore, these systems are designed for the machine available at that particular time.

We want to adapt the vast knowledge on the field of concurrent logic programming to actual technologies, like the modern programming language Kotlin and the lightweight coroutines, to develop a concurrent Prolog solver.

The goal of this thesis is to extend the 2P-Kt ecosystem with Concurrent Logic Programming support and, hence, to till the soil for future works. The contribution of this thesis is four folded: an overview of the state of art on the concurrent logic programming field, the design of an OR-parallel Prolog solver, its implementation and some comparative benchmarks.

The knowledge about concurrency in logic programming is very broad and overwhelming. For ease of reading we report the main aspects available in papers and works about this field, giving an overview to the reader. Then we deepen the concepts useful for our work to support its understanding.

The core of the contribution given by this thesis is the design of an OR-parallel Prolog solver. Firstly we formally describe the design using a labelled transition system and then we present the concrete design of the main entities of the solver.

Another important contribution is the implementation of the concurrent Prolog solver. We keep a high level of abstraction to ease the reusability of each component, but also to increase the extensibility. Furthermore we set up a framework for testing concurrent solvers, which is reusable, extensible and flexible.

The last contribution of this thesis is the measurement of the speedup gained by our solver. This is obtained through comparative benchmarks of the concurrent solver with respect to the sequential Prolog solver.

Thesis Structure Accordingly, the reminder of this thesis is structured as follows.

Chapter 2 discusses the state of the art of parallelism in logic programming

and it explains the technologies we use.

Chapter 3 contains the constraints and the goals we follow, giving a brief explanation of each of them.

Subsequently, chapter 4 discusses both abstract and concrete design of our concurrent solver. The former is a formal description of the concurrent state machine while the latter presents the main abstractions of the system.

After that, chapter 5 explains in detail the implementation step and it shows the most important code snippets.

Chapter 6 reports the validation process and the benchmarks. It also explains our test framework.

Finally, chapter 7 concludes this thesis by summarising its main contribution and the open issues.

Chapter 2

State of the Art

In this chapter, we recall the state of the art of parallelism in logic programming and the technologies we use.

It is important to remember that there are many aspects that are not explained in detail because they are not relevant for our purpose. The interested readers can explore the bibliography to deepen concurrency in logic programming.

From now on LP acronym is used instead of logic programming for ease of reading. Furthermore, to be consistent with literature, in this chapter we use *parallelism* as a synonym of *concurrency*.

Firstly, we propose an explanation of what is concurrent LP, detailing fundamentals, problems and how it is applied to Prolog. Then is reported the 2P-Kt ecosystem, showing an overview and detailing the most relevant aspects. The last part of this chapter is about concurrency in Kotlin and how it is handled.

2.1 Concurrent LP

A logic program is expressed as a set of logical formulas precisely specifying what is true. This approach to programming is different from the others, where the concern is how a specific problem should be solved. Here, the specification of how a problem is to be solved is expressed in the form of a precise sequence of steps executed by an engine.

One of the most attractive features of logic programming is the clean separation

of logic and control. The efficiency of the solution of a problem can be improved without any change in the solution itself. Concurrent LP is the set of algorithms and techniques that assess the concurrent research of the resolution tree to find answers to submitted queries.

In logic programming there is a knowledge base, an ordered set of clauses, to which a query is submitted. The answer to the query, which is a goal, is searched sequentially trying to match it with the head of each clause. When a match is found, a choice point is created to let the solver know where to backtrack if this clause fails or if other answers are requested. Likewise, each sub-goal is checked, sequentially and left to right, to find all the possible answers. When a (sub-)goal fails the backtracking takes place, moving the resolution to the last choice point created. Then the resolution restarts with the matching process from the next clause.

Before presenting the macro parallelism types, we briefly talk about non-determinism introduced by concurrency. There are two non-determinism types as explained in [36] and [22]: the *Don't-Know non-determinism* and the *Don't-Care non-determinism*.

The former occurs when the program has more than one clause to choose from, it picks one non-deterministically and creates a choice point used to backtrack in case of failure of the clause. This means that all the possible alternatives are tried and all the solutions can be found. In this type of non-determinism only successful computations are considered as results while the failings are not observed.

The latter non-determinism occurs when only a clause is chosen: no choice points are created and no backtracking is performed even if the clause fails. This means that only one alternative is explored so that only one result is obtained. In this type of non-determinism any computation is observed (hence failure is a valid result), so there is only one output that is logically correct.

When a query is submitted to a solver it is common that at least a positive answer is looked for, while the negative ones are not interesting. For this reason in this thesis, between the two kinds of non-determinism, we choose the Don't-Know one. Our choice implies a more complex resolution process but, as described in chapter 3, let us support the most known logic programming language, Prolog.

2.1.1 Macro parallelism types

In LP many kinds of concurrency can be exploited and they are grouped in three main categories: explicit, implicit, and hybrid parallelism.

Explicit parallelism This type of parallelism involves the extension of logic programming languages with explicit constructs that enable the control of parallelism. This means that the programmer has to control concurrency manually through the code.

The explicit control of concurrency needs a higher skilled programmer who can take advantage of the extending constructs to produce very efficient code, but it can create difficulties in debugging and testing so it is not suited for everyone. There are three sub-types of explicit parallel logic languages:

- Those that add *explicit message passing* primitives.
- Those that add *blackboard* primitives used by multiple processes running concurrently to communicate with each other.
- Those based on *guards*, *committed choice*, and *data flow synchronization*.

In order to keep this thesis focused, we consider these approaches only marginally and in those cases where they introduce execution mechanisms that are applicable also in the case of implicit exploitation of concurrency.

Implicit parallelism This type of parallelism involves the capability of the logic program engine to perform concurrent operations autonomously, extracting parallelism from logic programs without any programmer intervention. In contrast to the explicit parallelism, there are no extensions to the logic language so that the programmer writes classic logic programs. Instead, concurrency is achieved automatically running multiple processes that cooperate or concur to compute the results.

The aim is to speed up existing and new logic programs without any change to the code. There are three forms of implicit parallelism:

AND-parallelism The parallelism lays in the selection of the next literal to be solved. This allows the resolution of multiple literals concurrently. This form of parallelism, as mentioned into [40] and [41], can again be divided into:

- *Independent AND-Parallelism*: Literals of a clause are independent from each other, thus they do not share variables.
- *Dependent AND-Parallelism*: Literals of a clause are dependent from each other, thus they share variables.

OR-parallelism The parallelism resides in the selection of the clause to be used in the computation of the resolvent. This allows the solver to try multiple clauses in parallel.

Unification-parallelism The parallelism is at the level of the unification process when complex terms are present.

Mixing these OR-types of implicit parallelism together is possible, but implies an increase of problems that need to be handled.

Hybrid parallelism One last type of parallelism is the hybrid solution which takes advantage of both implicit and explicit parallelism. Typically the language is an extension of Prolog supplied with constructs that let the programmer handle concurrency. The engine exploits implicit parallelism unless explicit constructs are used.

An example of a hybrid parallel model is ACE [34] in which the programmer can either use standard Prolog and let the compiler take care of detecting implicit parallelism or use explicit language constructs to express parallelism.

2.1.2 Implicit parallelism in LP

Our goal, as it emerges from chapter 3, is to support Prolog improving performances of execution. To do so, we focus only on implicit parallelism. Before diving into the 2P-Kt ecosystem, we want to explain deeper this type of parallelism and the problems it involves. There are three main sources of implicit parallelism: OR-parallelism, AND-parallelism and Unification parallelism.

OR-parallelism OR-parallelism resides in the selection of the clause to be used in the computation of the resolvent. It means that this parallelism can be exploited every time a sub-goal unifies with more than one rule head: the bodies of each rule can be executed in parallel. A clear explanation can be found in [11].

This parallelism is the way of searching for all the solutions to the query by exploring the whole search space generated by multiple clauses applicable at each resolution step.

Or-parallelism not only allows the parallel execution of multiple clauses, but also enables the concurrent search into the whole search space generated for different alternatives applicable to the selected sub-goal.

This type of parallelism is more frequent in applications that explore a large search space via backtracking.

AND-parallelism AND-parallelism provides a means for performing concurrent evaluation of sub-goals in a clause. In this form of parallelism, a clause is seen as a problem that can be divided into sub-problems which can then be solved simultaneously. Each of these sub-problems will produce its own solution. At the end, the final solution is obtained by putting together the results of the sub-problems.

From a logic programming point of view, this means that each literal appearing in the selected clause body can be executed in parallel. There are two sub-types of AND-parallelism: Independent and Dependent.

Independent AND-Parallelism As mentioned in [40] and [41], Independent AND-Parallelism, IAP for brevity, parallel computation is exploited only between independent sub-goals and, therefore, is crucial to identify the independence.

Dependent AND-Parallelism The Dependent AND-Parallelism, DAP shortly, is similar to IAP with the difference that literals share dependencies from each other, as explained in [41]. It occurs when and-parallelism is allowed between sub-goals which share unbound variables at the time of invocation of the goals. Thus, processes compete in binding a shared variable or cooperate if the goals share the task of creating the binding for the common variable. In both cases some form of synchronization is needed when sub-goals are executed concurrently because their computation is affected by others.

This type of parallelism is harder to be exploited for Prolog. In fact, often, it is handled converting the language to a committed choice language or adopting other changes to operational semantics.

To deepen the concept, literals are considered *independent* when they do not share any unbound variable at run-time. That condition guarantees that the execution of one clause will not affect the other's execution. In this way, there is no need to introduce any form of synchronization during parallel execution.

The main focus in AND-parallelism is the detection of dependency between variables. In IAP only independent sub-goals are allowed for AND-parallel execution, while in DAP dependencies are handled exploiting synchronization mechanisms. Detection of dependencies can be done in three ways:

- *exclusively at run-time*, by parallelization tests that may slow down execution but guarantee the maximum grade of parallelism.
- *exclusively at compile-time*, by preprocessing the code that does not reduce performance but uses a lower grade of parallelism.
- *both at compile-time and at run-time*, by marking at compile-time selected literals and, when independence cannot be determined statically, generating a reduced set of efficient parallelization tests checked at run-time, as shown in [4].

There also exists a narrower form of IAP, which is called restricted independent AND-Parallelism, that implies stronger independence constraints.

Unification parallelism This parallelization, as explained in [19], arises during the unification of the arguments of a goal with the arguments of a clause head. The different argument terms can be unified in parallel as can the different sub-terms in a term. Thus, unification of two complex terms is broken down in pairwise unification of the different arguments. In this way, the sequence of unification can be performed in parallel.

This type of parallelism has a big disadvantage: due to its fine granularity, it requires specialized architectures in order to achieve results of any interest or, otherwise, it will incur in performance degradation due to the overhead.

2.1.3 Problems of implicit parallelism

At first sight exploiting parallelism seems simple but, in reality, many problems arise. Here, we report the main problems grouped by the parallelism type from

which they are generated.

Common problems There are some problems in common between all the parallelism types, so they are stated first. These are: *Granularity control*, *Scheduler*, *Process-based versus Processor-based* and *Architectural Influence*.

Granularity control Knowing the size of a task, in terms of execution time and space, is really important and it becomes almost a must if the parallelization is made on a distributed system where communication is a bottleneck. In reality, it is not possible to know the exact size of the task but only an estimation can be made.

Scheduler Many factors influence when a piece of work has to be executed and to which worker has to be assigned. This duty is in the hand of the scheduler, therefore, a lot of effort goes into its design and implementation.

Process-based versus Processor-based Deciding if it is better to spawn a process for each goal encountered (process-based) or to spawn multiple threads ahead of time and assign them parts of the computation (processor-based) is not simple. There are many aspects to consider, hence, both of these approaches are correct under specific circumstances.

Architectural Influence The architecture of the system can significantly impact the performances. Many architectures have been studied and each one tackles problems in different ways obtaining different results but, actually, there is no evidence that a particular architecture is the best and research is needed.

OR-parallelism problems Besides the common problems, OR-Parallelism has only a main issue to tackle that is known as *Multiple environment representation*, as mentioned in [38].

First of all, when implementing this type of parallelism, you have to visualize it and the easiest way is through the or-parallel search tree. At this point it appears obvious that the main problem in implementing this form of parallelism is the management of multiple bindings. In fact, the problem arises in the moment of representing and accessing the conditional bindings that are variables that can be assigned by more processes.

AND-parallelism problems This type of parallelism is more complicated than OR-parallelism due, but not limited, to the complexity of dependency detection, as explained in [32] and [33]. In fact there are four different problems: *Detection of data dependencies*, *Goals ordering*, *Management of shared variables* and *Backtracking*.

Detection of data dependencies Sub-goals can share variables directly or indirectly. The former can be detected at compile-time while the latter can be detected only at run-time. The condition of independence requires that there are no shared variables at the run-time execution.

Goals ordering This problem is about ordering sub-goals execution, hence determining if these are ready for execution.

Management of shared variables In case of dependencies, shared variables need to be validated and controlled.

Backtracking When the failure of a sub-goal occurs the main problem is to determine to which sub-goal execution should backtrack. This is due to the mismatch of the order of sub-goals. The problem becomes more complicated because and-parallel sub-goals may have nested sub-goals currently executing which have to be terminated or backtracked over.

AND-OR-parallelism problems The problems arising from combining AND-parallelism and OR-parallelism are various. They are not only the sum of both types of parallelism problems, but also those arising from requirements that are in contrast. Mainly:

- OR-parallelism focuses on improving the separation between parallel computations by assigning separate environments to the individual computing agents.
- AND-parallelism relies on the ability of different computing agents to cooperate and share environments to construct a single solution to the problem.

There are some other less important problems in this system: whether to recompute solutions of independent goals or to reuse them or how to deal with an additional level of scheduling, that is to determine whether an idle worker should perform OR-parallel work or AND-parallel work.

2.1.4 Concurrent Prolog

In this section we report the differences between a pure Logic Programming Language and Prolog. Moreover, we explain how to adapt concurrency to this specific language, which are the arising problems and how they can be tackled. Lastly some of the most known implementations are reported.

Prolog Logic programming is a quite powerful instrument that lets the programmer focus on what he wants to do instead of how to do it. In fact, there is a clear separation between logic and control. But there is a major drawback in those languages: there are no side-effects.

This shortage means that you can only find solutions to a problem without effectively exploiting them. Often, it is important to read or write on a file some solutions or to control the execution. These were the reasons for the birth of Prolog. This language supports many side-effects, ranging from reading or writing a file to modifying the knowledge base and, furthermore, controlling the execution by the use of *cut*.

Handling side-effects is not simple, in fact there are many articles that explain techniques and algorithms to tackle side-effects and even more implementations. Prolog execution is a sequential depth-first and left to right research of the resolution tree. This means that it is simple to know exactly when and where side-effects occur. Another important aspect that differentiates Prolog from traditional logic programming is that solutions are ordered.

Concurrency and side-effects When we introduce concurrency in the resolution process, things get complicated. Firstly, there is no solution ordering: this is not a problem corresponding to traditional logic programming, but it is for all those Prolog programs that rely on the order of solutions. In those cases, it is

necessary to add an ordering mechanism.

The other issue, the hardest one, is how to handle side-effects. Using this *impure* construction in Prolog is quite simple, thanks to its well defined path of exploration of the resolution tree. When we introduce concurrency, we need to define when a side-effect can be executed. The most effective technique is based on the *left-most principle*. This is a bit of information about the position of the current node of the resolution tree we are exploring. This knowledge lets us suspend the execution when a side-effect is encountered until that node becomes the leftmost, so that these constructs are executed in the same order of Prolog. We describe the three most effective techniques proposed in relevant papers.

In [39] is reported a method for preserving Prolog semantics in which the execution of logic programming can be seen as the process of maintaining a dynamic tree. The operational semantics of the language determines what operations on the tree are of interest. That method is used in DAP. Preventive strategies enforce the correct order of variable bindings by assigning producer or consumer status to each sub-goal that shares a given dependent variable. The strategy used is: the leftmost sub-goal that has access to the variable is designated as the producer for that variable, while all the others are consumers.

The articles [15] and [16] explain how to preserve Prolog semantics both in pure OR-parallel systems and in IAP systems. The former is the left-most principle. The latter is a technique that requires that sub-goals with side-effects have to suspend until all the preceding sub-goals (left to right precedence) finish. Furthermore, it is explained how to combine these two techniques for hybrid systems.

Another method to preserve Prolog semantics is presented in [21]. The idea is that instead of recomputing every sub-goal, some of the results can be reused and an algorithm is presented to handle this problem. The choice, *recompute or reuse*, is based on side-effects and where they appear in the resolution tree, using the left-most principle.

Relevant implementations Following, the most famous systems that support parallel logic programming are reported with a brief explanation. Readers should remind that many more exist but often these are extensions or adaptations of the most known for some specific needs. In [25] is presented *Aurora*, an or-parallel

implementation of Prolog based on a shared memory multiprocessor. This implementation uses the SRI model to represent different bindings of the same logical variable corresponding to different branches of the search space. In the SRI model, a group of workers cooperate to explore the Prolog search tree, which is defined implicitly by the program and needs to be constructed explicitly during the course of the exploration. In order to share or-parallel work, Aurora protects the choice points with locks avoiding that several workers steal the same piece of work. Furthermore, the Aurora system supports cut and standard Prolog built-in predicates including those which produce side-effects. Procedures to these predicates are required to suspend their resolution until they are in the leftmost branch of the tree.

Muse, which is reported in [3] and [2], is based on multiple sequential Prolog engines. Both conditional and unconditional variables are not shared in the MUSE system. Each processor operates as a sequential Prolog engine with its local memory. The MUSE OR-parallel Prolog system assumes a number of extended Prolog engines called workers. Each worker consists of two components: the *engine* and the *scheduler*. The former works as a sequential Prolog engine, the latter maintains the work between engines. Furthermore, the MUSE scheduler supports the sequential semantics of Prolog and efficient scheduling of speculative work.

In [11] is reported the *&-Prolog* system that exploits the independent AND-Parallelism combining a compiler performing independent detection with an efficient implementation of AND-parallelism on shared-memory multiprocessors. This system supports both automatic and user-expressed parallelisms. This can be expressed explicitly with the &-Prolog language. Input code is processed by several compiler modules: The *parallelizer* performs a dependency analysis on the input code using a conditional graph-based approach. It also receives information from the *Side-Effect Analyzer* on whether each non built-in predicate and clause of the given program is pure, or contains a side-effect. This information adds dependencies to correctly sequence such side-effects. The parallelizer then encodes the resulting graph using the & operator producing an “annotated” (parallelized) &-Prolog program. The parallelizer also receives information from the *granularity analyzer* regarding the size of the computation associated with a given goal, as can be seen in [24]. In order to improve its performance, a tool, written in Pro-

log, was implemented. IDRA (IDeal Resource Allocation) [14] collects traces from sequential executions and uses them to simulate an ideal parallel execution of the same program.

CIAO [20] is a modern, multi-paradigm programming language with an advanced programming environment. It aims at combining the flexibility of dynamic/scripting languages with the guarantees and performance of static languages. It is designed to run very efficiently on platforms ranging from small embedded processors to powerful multi-core architectures. Ciao has its main roots in &-Prolog. The most interesting aspect of this system is that it is one of the few still supported nowadays.

The *ACE* system [17], [31], [18], [30], [29] supports all form of parallelism. The dependent AND-parallelism has been incorporated in ACE using the *Filtered Binding Model* that directly encodes in the access path itself the information that allows a sub-goal to discriminate between producer and consumer accesses. Thus, the filtered binding model exploits restricted DAP and performs all operations in constant-time. The restriction is that unbound shared variables are not allowed to be bound to each other. Independent and-parallelism is also exploited via conditional graph expressions.

The idea behind the *Andorra-I*, [9], [37] is that if a goal has only one solution, it can be executed regardless of what other goals have been initiated. Such goals are called *determinate*. If no determinate goals can be found, a branch point for the leftmost goal is inserted. Each solution that is found for this branch point may spawn a different parallel branch (OR-parallelism using the binding array technique). This model leads to AND-parallel coroutines if two determinate goals are executed in parallel. In other words, dependent AND-parallelism is obtained by having determinate goals executed in parallel. The different alternatives to a goal may be executed in or-parallel. In Andorra-I, workers are arranged into teams that cooperate to exploit or-parallelism. Workers within a team cooperate to exploit and-parallelism. A team is composed of a master and some or no slaves. The configuration of workers into a team is decided by the user. In [10] it is explained that, in doing so, the problem that arises is the selection of a work, and more generally, where to redeploy workers to. The Andorra-I component responsible for doing that is the *preconfigurer*.

2.2 Logic Programming Ecosystems and 2P-Kt

We want to extend the 2P-Kt ecosystem, so first we explain what it is, giving an overview of it, and then we delve into its core aspects.

2.2.1 2P-Kt overview

To date, logic-based technologies are either built on top or as extensions of the Prolog language, mostly working as monolithic solutions tailored upon specific inference procedures, unification mechanisms, or knowledge representation techniques. Instead, to maximise their impact, logic-based technologies should support and enable the general-purpose exploitation of all the manifold contributions from logic programming. This is the idea behind tuProlog.

Nowadays, as explained in [6], it exists a reboot of the tuProlog project offering a general, extensible, and interoperable ecosystem for logic programming and symbolic AI: 2P-Kt, an open, multi-platform, multi-paradigm ecosystem for LP.

2P-Kt is the main Kotlin-based implementation available, currently targetting both JVM and JS, involving both GUI and CLI executables and providing a rich library for LP where each aspect is made individually available for re-use. More details about the history of Prolog can be found in [23] while the ISO specifications here [1]. The original idea behind the tuProlog implementation was based on four points:

- A light-weight Prolog system for distributed applications [12];
- Intentionally designed around a minimal core;
- Configurable by loading/unloading libraries of predicates;
- Native support for multi-paradigm programming [13] with a bi-directional integration among OOP and Prolog.

2P-Kt is an ecosystem of modules denoted by Gradle's notation: `:moduleName`. These modules are loosely-coupled, yet incrementally inter-dependent with an onion-like architectural design. Each one is a deployment unit, hence there is one jar file on the JVM for each module. Using a module as a dependency implies importing all its dependencies. The project structure in fig. 2.1 shows all the modules and their dependencies.

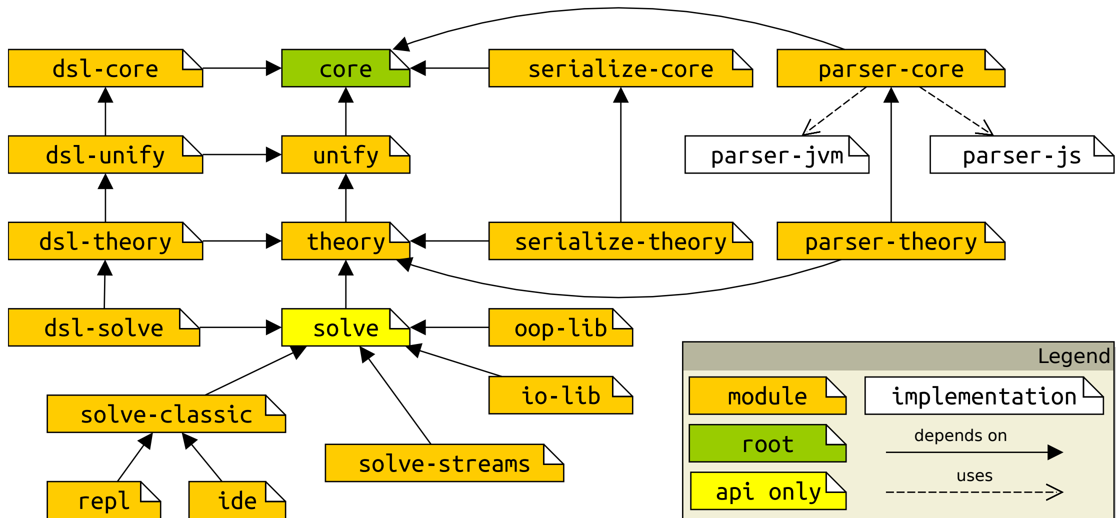

Figure 2.1: 2P-Kt project structure

`:core` provides knowledge representation facilities and common features.

`:unify` provides support for logic unification and a customisable notion of unificator based on [26].

`:theory` provides the support for in-memory storage and indexing of clauses.

`:solve` provides generic support for resolution-related stuff and it is agnostic with respect to inference procedures and resolution strategy.

`:solve-*` provides specific implementation for inference procedures and resolution strategy. It contains:

- `:solve-classic` SLD NF (Prolog-like) resolution [35, 8] based on the Piancastelli's state machine [28] (Prolog ISO [1] Compliant).
- `:solve-streams` SLD NF (Prolog-like) resolution [35, 8] based on Enrico Siboni's master thesis and on [5].

`:parser-*` supports parsing of terms and clauses in Prolog syntax. In particular:

- `:parser-core` supports parsing terms.
- `:parser-theory` supports parsing knowledge bases and streams of clauses.
- `:parser-jvm/js` platform-specific implementations, based on ANTLR [27].

`:serialization-*` supports the (de)serialization of terms, clauses, knowledge bases in YAML/JSON. It is divided in:

`:serialization-core` the (de)serialization of terms and clauses.

`:serialization-theory` the (de)serialization of knowledge bases and theories.

`:dsl-*` incrementally supports the Kotlin-based DSL for LP described in [7], aimed at blending LP, FP, and OOP. More in detail:

`:dsl-core` basic DSL for building terms/clauses in Kotlin.

`:dsl-unify` extension of the DSL including `:unify` facilities.

`:dsl-theory` extension of the DSL including `:theory` facilities.

`:dsl-solve` extension of the DSL including `:solve` facilities.

`:repl` command-line interface for Prolog.

`:ide` JavaFX-based GUI for Prolog (customisable).

`:io-lib` Prolog ISO compliant Prolog library for I/O.

`:oop-lib` Prolog library for OOP interoperability that lets Kotlin and Java OOP facilities be exploited from LP but only JVM is currently supported, due to limitations in Kotlin's reflection API.

2.2.2 Resolution

To help readers understand what we want to do in this thesis, we explain how the resolution is executed.

The `:solve` module provides generic support to logic resolution and re-use as much as possible of the Prolog standard without committing to any particular resolution strategy nor to any inference procedure. Thus, `:solve` provides the following abstractions: `Solver`s and `Solution`s. `Solver`s are reactive entities capable of answering users' queries by producing one or more `Solution`s, exploiting logic resolution. The exploration of the resolution tree to answer queries, requires a lot of information that are found in the execution context.

This container encapsulates some solver's internal state, so that resolution affects and is affected by the solver's execution context. The information contained herein are at least six: libraries, flags, knowledge bases (KB), operators, channels and, optionally, resolution-specific aspects.

Libraries are containers of built-in functionalities exploitable by resolution.

Possibly, including: some clauses containing logic predicates provided by the library, some primitives that are logic predicates implemented in Kotlin and provided by the library, some functions that are custom ways to reduce expressions of terms and some operators automatically imported into the solver along with the library.

Flags are configurable aspects of a solver. These are key-value pairs aimed at configuring a solver behaviour where the key is a string and the value is an arbitrary term.

Knowledge bases (KB) are the containers of the logic knowledge used by resolution to answer users' queries. Following Prolog's conventions, KB are of two sorts: static and dynamic. The former cannot be altered during resolution while the latter can be altered during resolution.

A set of operators is used to parse queries and to present solutions. Each solver may operate upon a dynamic set of operators that may be loaded along with libraries or be dynamically altered during resolution. Operators may affect the way users' queries are parsed and the way solutions are presented to users.

Channels are I/O facilities used to communicate with the external world. These are of two sorts: Input channels, also known as sources, which let a solver receive input messages and Output channels, also known as sinks, which let a solver provide output messages.

In the execution context there is also any resolution-specific aspect, which may vary depending on how resolution is implemented.

Solvers expose many `solve*` methods that return one or more Solutions. Each of these methods accept two parameters: a goal and the options. A goal is a `Struct` that represents the query to be answered via resolution. The options are in the form of an instance of `SolveOptions`, which describes or constrains the way answers are provided.

A `Solution` represents an answer provided by a `Solver` with respect to the user's query. It can be of three actual sorts:

- `Solution.Yes`: Representing a positive answer carrying a Unifier assigning some variable from the user's queries.
- `Solution.No`: Representing a negative answer.

- `Solution.Halt`: Representing an exceptional answer which carries a specific `TuPrologRuntimeException` locating the error with respect to the resolution process.

Notice that there are two types of solvers: `MutableSolver`s and `Solver`s. Both expose properties supporting inspection of their state but only mutable solvers expose methods to alter their state. Nonetheless the state of non-mutable solvers may change too, because of resolution and, thus, they are not immutable.

2.2.3 Primitives

Some built-in logic functionalities are easier to write in Kotlin as they require altering the solver's execution context. Furthermore some logic functionalities may exploit external computational facilities. Here is where primitives come into play because they are a means for calling Kotlin code from LP and a means for writing logic relations via OOP+FP+IP. To better explain how primitives work, we present a simple metaphor: users are clients for solvers, which act as servers, while the latter are clients for primitives, which act as servers. Thus, when needing to solve a (sub-)goal G, a solver may:

- Look into its knowledgebase.
- Pick a primitive P, compliant with G, from a library and, in order:
  - Send a request, describing G, to P,
  - Receive several responses, describing solutions to G, from P that possibly contain some side effects, to be reified.

To deepen the concept of primitives, these are:

- N-ary relations, in the eyes of logic programming,
- Servers & data producers, in the eyes of solvers,
- Callbacks, in the eyes of library implementers.

Within primitives, requests are descriptors of the current execution context and of the actual arguments of the (sub-)goal but, also, factories of responses which specify success or failure or exceptions and may provoke side effects. To expand the meaning of side effects, they represent an edit to be performed to some execution

context where only differences are represented. `SideEffects` can be applied to an `ExecutionContext` producing a new one that only differs from the former for the specific edit.

2.2.4 State Machine

To give the reader the last piece of knowledge needed to understand what we are doing, we need to explore one last module of the 2P-Kt ecosystem. This is the `:solve-classic`. Herein there is the main implementation of solver that provides Prolog ISO Standard resolution and has a State-Machine-based design, inspired by [28]. In this module there are the following main entities:

- `State` base type for states of the Prolog State Machine (PSM).
- `ClassicExecutionContext` data structure containing the execution context of (each state of) the PSM.
- `SolutionIterator` where the PSM is executed which is an object iterating over states and outputting solutions.
- `MutableSolutionIterator` a particular sort of solution iterator supporting hijack of state transitions that lets alter the execution context or the destination state.
- `AbstractClassicSolver` abstract class for classic-like solvers supporting the creation of custom solvers reusing (part of) the PSM.

The core logic dwells in the PSM, reported in fig. 2.2. This state machine features nine different states, one of which is initial while the other two are ending. Furthermore, there are two auxiliary data structure: the **Execution Context Stack** (E), tracking currently exploring items of the proof-tree, and the **Choice Point Queue** (C), tracking unexplored items of the proof-tree. The nine possible states are:

Goal Selection selects the next (sub-)goal to be proved, possibly popping from E.

Primitive Selection looks for a primitive to solve the selected (sub-)goal.

Primitive Execution if some is found, the first response is consumed, while a choice point is appended to C for the others.

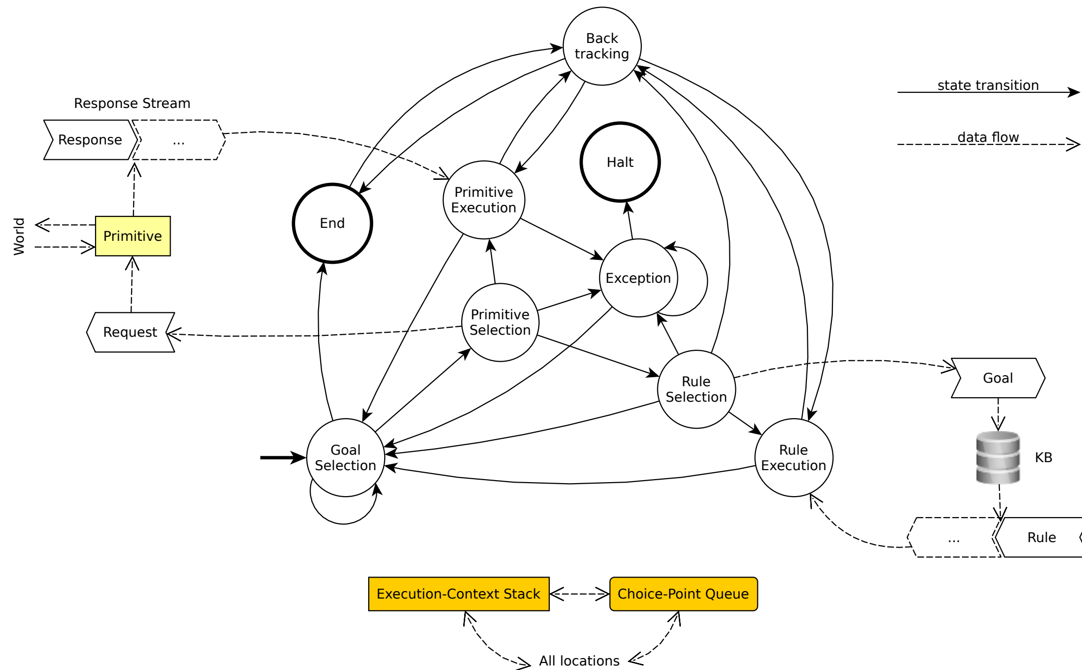

Figure 2.2: Prolog State Machine

Rule Selection if no primitive is selected, some rule is looked for instead, for the same sub-goal.

Rule Execution if a rule is found, resolution proceeds by pushing a new execution context to E, to tackle the body.

Backtracking if no rule is found, the sub-goal is considered failed and resolution continues taking the next choice point in C.

End reached when there are no more goals or no more choice points.

Exception reached when some primitive responses carry an exception. This is where catching occurs.

Halt reached when an exception is not caught.

2.3 Coroutines and concurrency in Kotlin

To explain how we exploit concurrency in Prolog firstly we must briefly expose how Kotlin can support it. Here we introduce the main constructs available, how they work and how they interact.

Coroutines Coroutines[1] are computer program components that generalize subroutines for non-preemptive multitasking, by allowing execution to be suspended and resumed. Kotlin language provides only minimal low-level APIs in its standard library to enable various other libraries to utilize coroutines. Kotlin's concept of suspending function provides a safer and less error-prone abstraction for asynchronous operations than futures and promises.

A coroutine is an instance of suspendable computation. It is conceptually similar to a thread, in the sense that it takes a block of code to run that works concurrently with the rest of the code. However, a coroutine is not bound to any particular thread. It may suspend its execution in one thread and resume in another one. Coroutines can be thought of as light-weight threads with a number of important differences that make their real-life usage very different from threads.

Coroutines follow a principle of structured concurrency which means that new coroutines can be launched only in a specific `CoroutineScope`[2] which delimits the lifetime of the coroutine. Structured concurrency ensures that coroutines are not lost and do not leak. An outer scope cannot complete until all its children coroutines complete. Structured concurrency also ensures that any errors in the code are properly reported and are never lost.

Coroutines always execute in a context represented by the `CoroutineContext`[3] type, defined in the Kotlin standard library. The coroutine context is a set of various elements. The main elements are the `Job`[4] of the coroutine and its dispatcher. The coroutine dispatcher determines what thread or threads the corresponding coroutine uses for its execution. The coroutine dispatcher can confine coroutine execution to a specific thread, dispatch it to a thread pool, or let it run unconfined.

When a coroutine is launched in the `CoroutineScope` of another coroutine, it inherits its context and the `Job` of the new coroutine becomes a child of the parent coroutine's job. When the parent coroutine is cancelled, all its children are recursively cancelled, too.

[1] https://kotlinlang.org/docs/coroutines-basics.html

[2] https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-scope/index.html

[3] https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-scope/coroutine-context.html

[4] https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-job/index.html

Channels Coroutines are the components that allow execution to be suspended and resumed and, thus, let the programmer exploit concurrency. Often these components need to communicate. When coroutines need to exchange many values, a stream, channels[5] comes into play.

A Channel is a non-blocking primitive for communication between a sender and a receiver. It is conceptually very similar to a `BlockingQueue`. One key difference is that instead of a blocking put operation it has a suspending `send`, and instead of a blocking take operation it has a suspending `receive`. Unlike a queue, a channel can be closed to indicate that no more elements are coming. Conceptually, a close is like sending a special close token to the channel. The iteration stops as soon as this close token is received, so there is a guarantee that all previously sent elements before the close are received.

[5] `https://kotlinlang.org/docs/channels.html`

Chapter 3

Requirements

Concurrent Logic Programming, as shown in chapter 2, is a complex and vast world where many choices can be made. To avoid losing sight of our goal, in this chapter we define what we want and which are the constraints.

3.1 Constraints

To avoid reinventing the wheel, we want to exploit something that already exists. Our choice falls on the adoption of 2P-Kt since we already have the source code and we can use it without any further problem. Moreover, this ecosystem is already prepared for adding new modules and to support reuse. This choice implies two constraints:

Kotlin The adopted programming language is Kotlin. This technological commitment entails three main advantages:

- A wide support for concurrency through coroutines. The documentation is easily available and the library provides simple mechanisms with a high level of abstraction, so design and implementation are simplified.
- Multi-platform support. Artefacts creation for different platforms is extremely simplified and does not require any particular effort.
- The management of synchronization mechanisms is simple. Coroutines are defined to reduce synchronization problems, in fact they are equipped with ad-hoc mechanisms to handle these aspects.

However, there are also two main disadvantages to consider:

- The level of abstraction increases. This is a trade-off because, to lower it, a much more complex design and implementation process would be required.
- Work is bound to the coroutines library. This drawback can be mitigated by abstracting the model of the concurrent resolution mechanism, such that, theoretically, it is possible to realize alternatives but functionally equivalent implementations on different platforms with similar abstractions.

Multi-platform The 2P-Kt is a multi-platform project which target both the JVM and JS. This is not a hard constraint because, to further reduce complexity, we can focus on one platform at a time. In the first implementation we work only on the JVM due to its better support for concurrency.

3.2 Goals

The main goal we want to achieve in this thesis is to create an application programming interface for a concurrent Prolog's resolution. As reported in chapter 2, many types of parallelism and even more techniques and algorithms exist to obtain concurrency. Furthermore, there are a large number of problems to tackle.

For these reasons, we choose to focus on the creation of all the abstract components that support a concurrent solver. We add a module to 2P-Kt, that contains these abstractions which can be concretely implemented in a concurrent solver. Then, we have to implement a concrete solver which supports one type of concurrency, specifically the OR-parallelism.

The abstractions are the basis for new implementation while our solver is the starting point for further developments. This means that the design has to support features added in the future.

The first goal is to implement a concurrent resolution module that supports *pure* logic programming, where *pure* means without side effects. To pursue our goal, which is the creation of a basic concurrent solver, we need to analyze the problems that the use of Prolog entails and how they can be handled. The main

ones are to respect the Prolog semantics and the handling of side-effects.

Respecting the Prolog semantics, which means adhering to the ISO standards, is not a priority to us. Nonetheless our goal is to design the abstraction to be reusable and extensible so that a mechanism that preserve the Prolog semantics can be added. The drawback of this mechanism is that it reduce drastically the benefits of the concurrent exploration of the resolution tree.

Another goal is to support the injection of a mechanism of explicit control over what tree exploration model the solver uses. This control tool allows us to respect Prolog semantics only when necessary.

This leads us to initially create a pure solver with a level of abstraction and encapsulation that allows the injection of new mechanisms as additional components without changes to the solver itself.

The other core problem is the *side-effects handling*: in pure logic programming side-effects do not exist, but to support Prolog in its entirety, we have to prepare the solver to support these constructs. The idea, as with the semantics problem, is to encapsulate components and to make them as much reusable as possible in order to add side-effects management mechanisms. In literature, the most used technique is the one related to the *leftmost* concept. Thus, solver must allow the injection of a mechanism that can:

- Suspend computation.
- Resume computation.
- Calculate the current node position with respect to the resolution tree.

This level of encapsulation allows us to re-elaborate some constructs in order to give them a different semantics and, thus, obtain greater flexibility. An example is the *cut* that, instead of the classic behavior, can become a limiter when infinite recursive computation occurs.

Chapter 4

Design

In this chapter we define the design of the concurrent solver for pure logic programming. Firstly we report the abstract design that formalizes the state machine used by the solver. Then we present the concrete design describing the main abstractions in terms of structure, interaction and behavior. Furthermore we discuss how this design meets the requirements described in chapter 3.

4.1 Abstract design

4.1.1 Syntax and Notational Conventions

Knowledge Bases Let $\mathcal{A}$ be the set of all atomic logic formulæ, and let $\mathcal{H}$ be the set of all well-formed Horn clauses in the form:

$$a \leftarrow a_1, \ldots, a_n \quad \text{s.t. } n > 0$$

(also known as rules) where $a, a_1, \ldots, a_n \in \mathcal{A}$ are logic predicates of arbitrary arity. Let us enumerate rules in $\mathcal{H}$ by h. Thus, we define knowledge bases as ordered containers of rules of the form $[h_1, h_2, \ldots]$.

Let K denote the knowledge base, with $K \in \mathcal{H}^*$, where $\mathcal{H}^*$ is the set of all possible KB.

Finally, let $get : \mathcal{H}^* \times \mathcal{H} \rightarrow \mathcal{H}^*$ be a function defined as follows:

$$get(K, h) = \begin{cases} [h' \mid get(K')] & \text{if } K \equiv [h' \mid K'] \wedge mgu(h, h') \neq \bot \\ [get(K')] & \text{if } K \equiv [h' \mid K'] \wedge mgu(h, h') = \bot \\ [] & \text{if } K \equiv [] \end{cases}$$

aimed to select all rules in K unifying with the clause h.

Primitives Let $\mathcal{F}$ be the set of all predicate symbols, enumerated by f, and let $\mathbb{N}$ be the set of natural numbers. Thus,

- let $\mathcal{F} \times \mathbb{N}$ denote the set of all possible signatures and
- let $(f, n) \in \mathcal{F} \times \mathbb{N}$ denote the generic signature of a n-ary predicate whose functor is f.

Let us define the function *signature* : $\mathcal{A} \rightarrow \mathcal{F} \times \mathbb{N}$ as:

$$signature(f(t_1, \ldots, t_n)) = (f, n)$$

that computes the signature of any possible atomic predicate.

Let Θ be the set of all possible substitutions – including the failed substitution $\bot$, the empty unifier $\varnothing$, and any unifier in the form $\{V_1 \mapsto t_1, V_2 \mapsto t_2, \ldots\}$ where V_i are logic variables and t_i are logic terms – and let us enumerate elements in Θ by θ.

Let then $\mathcal{X}$ be the set of all possible *exceptions* – i.e., arbitrary terms describing error situations –, enumerated by x.

Accordingly, $\mathcal{R} = \mathcal{X} \cup \Theta$ represents the set of all possible primitive *responses*, enumerated by r.

Then, let $\mathcal{P}$ be the set of all possible primitives with the form:

$$p : \mathcal{H}^* \times \mathcal{A} \rightarrow \mathcal{R}^*$$

In other words, we call "primitive" any function $p \in \mathcal{P}$ accepting a knowledge base $K \in \mathcal{H}^*$ and a goal $a \in \mathcal{A}$ as input and producing a stream of responses $p(K, a) =$

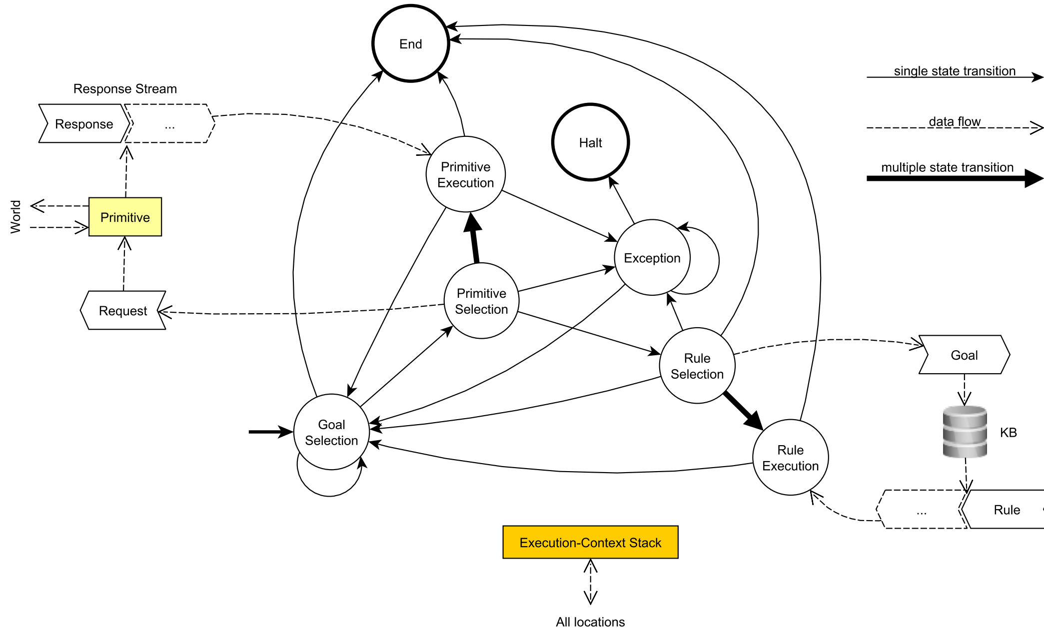

Figure 4.1: Concurrent Prolog State Machine

$\bar{r} \in \mathcal{R}^*$ as output—where each response $r \in \bar{r}$ may either be a substitution or an exception.

Finally, we define a primitive store I as a relation of the form:

$$I \subseteq \mathcal{F} \times \mathbb{N} \times \mathcal{P}$$

that is, any possible indexing of primitives by signatures. Given a particular primitive store I, we enumerate its elements by (f, n, p).

Solver Automaton We define an *execution context* E as a tuple of the form $(\theta, \bar{a}, h, p)$ where:

- $\theta \in \Theta$ is a substitution;
- $\bar{a} \in \mathcal{A}^*$ is a stream of *goals*;
- $h \in \mathcal{H} \cup \varnothing$ is a *rule*;
- $r \in \mathcal{R} \cup \varnothing$ is a primitive *response*.

Accordingly, letting $\mathcal{E}$ denote the set of all possible execution contexts, we define an execution-context *stack* $\mathbf{E}$ as a list of the form $[E_0, E_1, \ldots] \in \mathcal{E}^*$, where E_0 can either be called the "*current* execution context" or the "*top* of the execution-context stack". Finally, let $\mathcal{L}$ denote the set containing all the locations depicted in fig. 4.1:

$$\mathcal{L} = \{\mathsf{Goal\ Selection}, \mathsf{Primitive\ Selection}, \mathsf{Primitive\ Execution}, \mathsf{Rule\ Selection}, \\ \mathsf{Rule\ Execution}, \mathsf{Exception}, \mathsf{End}, \mathsf{Halt}\}$$

We enumerate items in $\mathcal{L}$ by L.

4.1.2 Semantics

The semantics of our concurrent Prolog state machine can be described in terms of *states* and *transitions* between them. In this regard, a *state* is a tuple of the form $\langle L, \mathbf{E} \rangle$ where

- $L \in \mathcal{L}$ is a location,
- $\mathbf{E} \in \mathcal{E}^*$ is an execution-context stack

Accordingly, the state machine semantics is defined as a *labelled transition system* $\langle \mathcal{S}, \Lambda, s_0, \longrightarrow \rangle$ where:

- $\mathcal{S}$ is the set of all possible states,
- $\Lambda = \{\tau\} \cup \mathcal{X} \cup \Theta$ is a set of labels,
- $s_0 \in \mathcal{S}$ is the initial state,
- $\longrightarrow \subseteq \mathcal{S} \times \Lambda \times \mathcal{S}$ is a transition relation dictating how state may evolve in time.

There, transition labels in Λ denote relevant observable events, while the anonymous label τ denotes internal events. Observable events of interest can be, for instance, the production of either a positive or negative solution – denoted by the corresponding substitution in Θ – or the production of an exceptional solution—denoted by the corresponding exception in $\mathcal{X}$. In the following, for the sake of notation simplicity, we write $s \xrightarrow{\lambda} s'$ instead of (s, λ, s') to refer to transitions in $\longrightarrow$.

Assuming that a KB K and a primitive store I are provided, and that the initial state $s_0 \in \mathcal{K}$ is always a tuple of the form:

$$\langle \textsf{Goal Selection},\ [(\varnothing, [g_0], \varnothing, \varnothing)] \rangle$$

for some initial goal g_0 – meaning that *(i)* the initial location is always Goal Selection, *(ii)* the current context initially only contains the empty unifier $\varnothing$, g_0, the empty rule $\varnothing$ and the empty primitive response $\varnothing$ – we can *intentionally* define the admissible transitions in $\longrightarrow$ via the following transition rules—each one corresponding to an arrow in fig. 4.1. It is important to highlight that when more than one rule is found and when more than one primitive result is obtained, concurrency starts, because each possibility is handled in a new resolution branch. This is depicted with a bold arrow in the picture.

Goal Selection The purpose of the Goal Selection location is to decide what to do next depending on which and how many (sub-)goals are in the current execution context. There are three relevant situations handled in this location, by as many transition rules. More precisely, if the current execution context does not contain any goals, this implies either that a new solution should be yielded, or the top execution context should be popped from the stack. Conversely, if the current execution context does hold at least one goal, the automaton commits to that goal—meaning that it tries to prove its truth via subsequent transitions.

Accordingly, the following transition rule handles the case where there is only one last execution context on the stack with no more goals—thus, implying the automaton should move into the End location and a new solution should be yielded:

$$\frac{E = (\theta, [], \varnothing, \varnothing)}{\langle \textsf{Goal Selection}, [E] \rangle \xrightarrow{\theta} \langle \textsf{End}, [E] \rangle}$$

The positive or negative solution depends on the θ substitution of the last execution context E, which can be either a unifier or the failed substitution $\bot$. In the former case, θ synthesises all the variable assignments computed so far by the automaton.

Conversely, the following transition rule handles the case where the current execution context has no more goals, but the stack contains more execution contexts.

When this is the case, the automaton simply pops the current execution context from the stack and holds the Goal Selection location:

$$\frac{E_0 = (\theta_0, [], \varnothing, \varnothing) \qquad E_1 = (\theta_1, \bar{g}, \varnothing, \varnothing) \qquad E_1' = (\theta_0, \bar{g}, \varnothing, \varnothing)}{\langle \textsf{Goal Selection}, [E_0, E_1 \mid \mathbf{E}] \rangle \xrightarrow{\tau} \langle \textsf{Goal Selection}, [E_1' \mid \mathbf{E}] \rangle}$$

Before popping the current execution context E_0, the automaton spreads its θ_0 substitution to the parent execution context E_1—as θ_0 can contain more assignments than θ_1.

Finally, the following transition rule handles the case in which the current execution context contains a non-empty stream of goals $\bar{g}$. When this is the case the automaton applies the most recent substitution θ to all the goals in $\bar{g}$ before moving into Primitive Selection location and tries then to prove the truth of the first sub-goal in $\bar{g}$:

$$\frac{E = (\theta, \bar{g}, \varnothing, \varnothing) \qquad E' = (\theta, \bar{g}', \varnothing, \varnothing) \qquad \bar{g}' = \bar{g}/\theta}{\langle \textsf{Goal Selection}, [E \mid \mathbf{E}] \rangle \xrightarrow{\tau} \langle \textsf{Primitive Selection}, [E' \mid \mathbf{E}] \rangle}$$

where by $\bar{g}/\theta$ we mean the application of substitution θ to all goals in $\bar{g}$.

Primitive Selection The purpose of the Primitive Selection location is to select a primitive in the primitive store I in order to prove a (sub-)goal—provided a primitive matching the goal's signature exists in I. If this is the case, the primitive is triggered and *all* the primitive responses are handled, firing a new concurrent resolution branch for each response. Thus, there are three relevant situations handled in this location, by as many transition rules. A pivotal role in discriminating among situations is played by the first goal g of the current execution context. If primitive p is indexed in I via signature of g, then p is triggered and a response stream is generated in return. All the items in the stream are consumed individually, each in a new concurrent resolution branch with its own execution context. Each item can be either an exception or a substitution. Each case is handled by a different transition rule. Otherwise, if the signature of g matches no primitive in I, no primitive is selected and, via subsequent transitions, a rule for the same goal is searched.

In particular, the following transition rule handles the case where a primitive

p is found in I and the response r, one of the streams, is a substitution. When this is the case, a new execution context is pushed on the stack for handling the response. After that, the automaton moves to the Primitive Execution location.

$$\frac{E_0 = (\theta, \bar{g}, \varnothing, \varnothing) \quad \bar{g} = [g \mid \bar{g}'] \quad (f, n) = \mathit{signature}(g) \quad (f, n, p) \in I \qquad p(K, g) = (r_1, r_2, \ldots, r_i, \ldots) \in \mathcal{R}^* \quad r_i \in \Theta \quad E_1 = (\theta, \bar{g}, \varnothing, r_i)}{\langle \textsf{Primitive Selection}, [E_0 \mid \mathbf{E}] \rangle \xrightarrow{\tau} \langle \textsf{Primitive Execution}, [E_1, E_0 \mid \mathbf{E}], \rangle}$$

Conversely, the following transition rule handles the case in which the primitive response r is an exception. The automaton moves to the Exception location:

$$\frac{E_0 = (\theta, [g \mid \bar{g}'], \varnothing, \varnothing) \quad \bar{g} = [g \mid \bar{g}'] \quad (f, n) = \mathit{signature}(g) \quad (f, n, p) \in I \qquad p(K, g) = (r_1, r_2, \ldots, r_i, \ldots) \in \mathcal{R}^* \quad r_i \in \mathcal{X} \quad E_1 = (\theta, \bar{g}, \varnothing, r_i)}{\langle \textsf{Primitive Selection}, [E_0 \mid \mathbf{E}] \rangle \xrightarrow{\tau} \langle \textsf{Exception}, [E_1, E_0 \mid \mathbf{E}] \rangle}$$

In this case as well a new execution context is pushed on the stack in order to make the exception visible in Exception.

Finally, the following transition rule handles the case in which a primitive is not available in I for the goal g. When this is the case, the automaton simply moves to the Rule Selection location:

$$\frac{E = (\theta, [g \mid \bar{g}'], \varnothing, \varnothing) \qquad (f, n) = \mathit{signature}(g) \qquad (f, n, p) \notin I}{\langle \textsf{Primitive Selection}, [E \mid \mathbf{E}] \rangle \xrightarrow{\tau} \langle \textsf{Rule Selection}, [E \mid \mathbf{E}] \rangle}$$

Primitive Execution The purpose of the Primitive Execution location is to *lazily* handle the response produced by primitives. To this regard, there are three relevant situations, handled by as many transition rules. In fact, while consuming a solution response, the automaton may either encounter an exception or a substitution, which can be positive or negative. The latter case is the most interesting one, as the substitution must be kept into account in the next computational steps. Conversely, in the other case, the exception needs to be handled accordingly.

In particular, the following transition rule handles the case where the response is a positive unifier θ'. When this is the case, θ' is merged with the current execution

context substitution θ and execution proceeds to the Goal Selection location.

$$\frac{E = (\theta, [g \mid \bar{g}], \varnothing, \theta') \qquad \theta' \in \Theta - \{\bot\} \qquad E' = (\theta \cup \theta', \bar{g}, \varnothing, \varnothing)}{\langle \textsf{Primitive Execution}, [E \mid \mathbf{E}] \rangle \xrightarrow{\tau} \langle \textsf{Goal Selection}, [E' \mid \mathbf{E}] \rangle}$$

Conversely, the following transition rule handles the case where the response is a failure (i.e. $\bot$). In this case, the automaton simply moves to the End location.

$$\frac{E = (\theta, \bar{g}, \varnothing, r) \qquad r = \bot \qquad E' = (\theta, \bar{g}, \varnothing, \bot)}{\langle \textsf{Primitive Execution}, [E \mid \mathbf{E}] \rangle \xrightarrow{\tau} \langle \textsf{End}, [E' \mid \mathbf{E}] \rangle}$$

Finally, the following transition rule handles the case where the response is an exception. When this is the case, the automaton simply moves into the Exception location.

$$\frac{E = (\theta, \bar{g}, \varnothing, x) \qquad x \in \mathcal{X} \qquad E' = (\theta, \bar{g}, \varnothing, x)}{\langle \textsf{Primitive Execution}, [E \mid \mathbf{E}] \rangle \xrightarrow{\tau} \langle \textsf{Exception}, [E' \mid \mathbf{E}] \rangle}$$

Rule Selection The Rule Selection location is for clauses what the Primitive Selection location is for primitives. It aims at querying the KB and selecting the rules to be executed to prove a particular (sub-)goal true, provided that no primitive has been selected for the purpose. Accordingly, three transition rules are defined, each one handling a particular situation. The most common situation here is that a rule r is selected from K in order to solve some goal g. However, there is a small set of goals for which a particular treatment is reserved. These are: `true`, `fail`, and `false`. The first is always considered successful, while the others are always considered failed.

More precisely, the following transition rule takes care of the goal `true`. As it must always be evaluated successfully, this transition simply makes the automaton move to the Goal Selection location, after consuming the current goal g_0. More formally:

$$\frac{E = (\theta, [\texttt{true} \mid \bar{g}], \varnothing, \varnothing) \quad E' = (\theta, \bar{g}, \varnothing, \varnothing)}{\langle \textsf{Rule Selection}, [E \mid \mathbf{E}] \rangle \xrightarrow{\tau} \langle \textsf{Goal Selection}, [E' \mid \mathbf{E}] \rangle}$$

Conversely, the following transition rule takes care of goals such as `fail` and `false`. As they must always be evaluated unsuccessfully, this transition simply makes the automaton move into the End location, after consuming the current goal

g. This transition rule also handles the case where K is queried for all rules whose head unifies with the current goal g, but no one is found.

$$\frac{E = (\theta, [g, \ldots], \varnothing, \varnothing) \qquad g \in \{\texttt{false}, \texttt{fail}\} \vee get(K, g) = []}{\langle \textsf{Rule Selection}, [E \mid \mathbf{E}] \rangle \xrightarrow{\tau} \langle \textsf{End}, [E \mid \mathbf{E}] \rangle}$$

Finally, the following transition rule handles the general case where K is queried for all rules $\bar{h}$ whose head unifies with the current goal g. These rules are handled individually, thus for each rule h a concurrent resolution branch is created and the automaton must then move to location Rule Execution, after pushing a new execution context on the stack aimed at handling h:

$$\frac{\begin{array}{c} E_0 = (\theta, \bar{g}, \varnothing, \varnothing) \qquad \bar{g} = [g \mid \bar{g}'] \qquad g \in \mathcal{A} - \{\texttt{true}, \texttt{false}, \texttt{fail}\} \\ get(K, g) = [h_1, \ldots, h_i, \ldots] \qquad E_1 = (\theta, \bar{g}, h_i, \varnothing) \end{array}}{\langle \textsf{Rule Selection}, [E_0 \mid \mathbf{E}] \rangle \xrightarrow{\tau} \langle \textsf{Rule Execution}, [E_1, E_0 \mid \mathbf{E}] \rangle}$$

Rule Execution The Rule Execution location is for clauses what the Primitive Execution location is for primitives. Thus, the purpose of this location is to *lazily* handle the rule produced by KB. Accordingly, two transition rules are defined, each one handling a particular situation. In both situations, the current execution context is assumed to carry a non-empty rule to be handled. One situation concerns the case where the rule has a head matching the current context goal. In this case, the execution can go on and focus on the body of that rule. The other situation concerns the opposite case, where execution must proceed to the End.

Accordingly, the following transition rule handles the first situation. The current execution context's first goal is g and the rule is h. Provided that the head of h unifies with g, and letting θ' be their unifier, the current execution context is updated in such a way that the new substitution is $\theta \cup \theta'$ and the new goal stream contains all the atoms from the body of h, subject to the substitution θ'. After that, the automaton moves to the Goal Selection location.

$$\frac{\begin{array}{c} E = (\theta, [g \mid \bar{g}], h, \varnothing) \quad h = (a \leftarrow a_1, \ldots, a_n) \quad \theta' = mgu(g, a) \neq \bot \\ \bar{g} = [g_1, \ldots, g_n] \quad E' = (\theta \cup \theta', [g_1/\theta', \ldots, g_n/\theta'], \varnothing, \varnothing) \end{array}}{\langle \textsf{Rule Execution}, [E \mid \mathbf{E}] \rangle \xrightarrow{\tau} \langle \textsf{Goal Selection}, [E' \mid \mathbf{E}] \rangle}$$

Conversely, the following transition rule handles the case where the head of h does not unify with g. In this case, the automaton simply moves to the End location.

$$\frac{E = (\theta, [g \mid \bar{g}], h, \varnothing) \quad h = (a \leftarrow \ldots) \quad mgu(g, a) = \bot \quad E' = (\theta, [g \mid \bar{g}], \varnothing, \varnothing)}{\langle \textsf{Rule Execution}, [E \mid \mathbf{E}] \rangle \xrightarrow{\tau} \langle \textsf{End}, [E' \mid \mathbf{E}] \rangle}$$

Exception Handling The Exception location has the purpose of managing exceptions possibly raised by primitive responses in the Standard Prolog way—in example by climbing the proof tree towards the root, looking for a `catch/3` (sub-)goal whose second argument unifies with the raised exception and setting its third argument as the next sub-goal to be proved.

Following this purpose, the Exception location may encounter two notable situations. In both ones, the current execution is assumed to be carrying an exception $x \in \mathcal{X}$. The first situation concerns the case where the exception can be *caught* since there exists on the stack an execution context of the form `catch/3` which may intercept the exception and let resolution continue. The second situation concerns the opposite case where the exception *cannot* be caught – since no such an execution context is contained into the stack – and resolution must be therefore interrupted.

In particular, the following transition rule handles the first situation where an execution context E' exists on the stack whose first goal is $\texttt{catch}(g_1, g_2, g_3)$. When this is the case, we denote by θ' the MGU among x and g_2. Then, the automaton pops from the stack all execution contexts up to E' (included), pushes a new execution context carrying g_3 as the first goal, and moves to the Goal Selection location.

$$\frac{\begin{array}{c} E = (\theta, [g \mid \bar{g}], \varnothing, x) \quad x \in \mathcal{X} \quad E' = (\vartheta, [\texttt{catch}(g_1, g_2, g_3), \ldots], h, r) \\ \theta' = mgu(x, g_2) \neq \bot \quad \theta'' = \vartheta \cup \theta' \quad E'' = (\theta'', [g_3/\theta''], \varnothing, \varnothing) \end{array}}{\langle \textsf{Exception}, [E, \ldots, E' \mid \mathbf{E}] \rangle \xrightarrow{\tau} \langle \textsf{Goal Selection}, [E'' \mid \mathbf{E}] \rangle}$$

Conversely, the following transition rule handles the opposite situation where no execution context on the stack carries $\texttt{catch}(g_1, g_2, g_3)$ as the first goal—or, if it does, g_2 does not unify with x. When this is the case, the automaton simply

moves to the Halt location, yielding an exceptional solution to the users.

$$\frac{E = (\theta, [g \mid \bar{g}], \varnothing, x) \quad x \in \mathcal{X} \qquad \nexists E \in \mathbf{E} : E = (\vartheta, [\mathtt{catch}(g_1, g_2, g_3), \ldots], \ldots) \land mgu(g_2, x) \neq \bot}{\langle \mathsf{Exception}, [E \mid \mathbf{E}] \rangle \xrightarrow{x} \langle \mathsf{Halt}, [E \mid \mathbf{E}] \rangle}$$

Next Solutions Both the End and Halt locations are final, meaning that the automaton reaches them immediately after the production of a novel solution—be it positive, negative, or exceptional. In a concurrent Prolog state machine these states are both sink, certainly provoking the automaton termination, thus terminating the current resolution branch. Resolution terminates only when all the fired branches terminate unless a maximum number of solutions is requested, in which case computation ends when the solution count reaches the specified limit.

Ordering of solutions Given the initial goal g_0, it is possible to compute the admissible transitions using the transition rules. The solutions to the query contained in the initial state are obtained by collecting all the labels of transitions other than τ. These transitions represent output events of a solution.

The state machine is nondeterministic because more than one transition rule is available in some locations. Moreover each resolution branch has its own length. This means that the number of consecutive transitions that lead to a solution varies from one resolution branch to the other. Due to the nondeterminism of the state machine the ordering of solutions is unpredictable.

4.2 Concrete design

To isolate as much as possible each component of our module, we define some abstractions to improve the reusability. Some already exist, so we give their overview, while others are made from scratch as shown in fig. 4.2.

Solver This already existing abstraction is a general type for logic solvers, which is any entity capable of solving some logic query provided as a goal according to some logic. This entity needs to implement one or more inference rules, via some resolution strategy. Solvers are not immutable entities. Their state

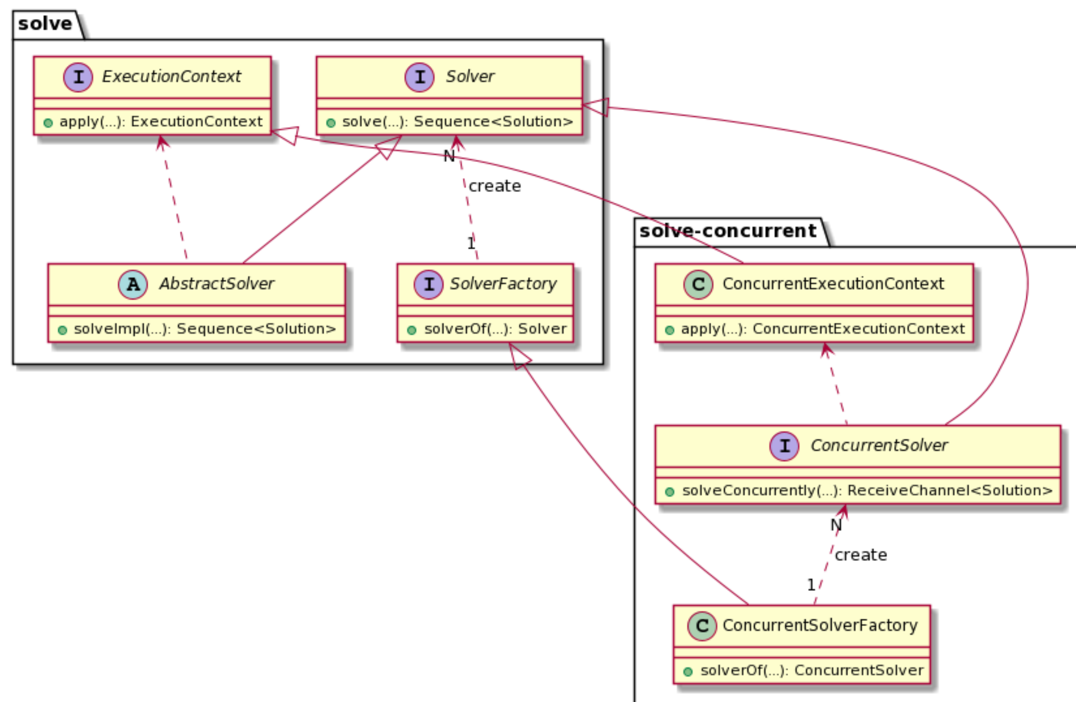

Figure 4.2: Main abstractions

may mutate as an effect of solving queries. Herein there are all the *solve** methods, each of which is an overload of a core `solve` function.

AbstractSolver It is part of the `:solve` module and it handles the general setup shared by any `Solver`. Furthermore, it contains some utility functions. The main role of this abstraction other than to reduce code redundancy, is to introduce the `ExecutionContext`. The last important aspect of this entity is that it defines the `solve` method based on a new concept of `solveImpl` which needs to be implemented in concrete classes.

ConcurrentSolver This new abstraction is the concurrent version of the `Solver`. Herein we define a `solveConcurrently` method that, like the `solve` one, accepts a goal and some options and returns a `ReceiveChannel` and, thus, computes solutions concurrently.

ExecutionContext It is an already existing concept defined in the `:solve` module. It is an interface representing the Solver execution context, containing important information that determines its behaviour. Herein there are:

- The current procedure being executed;
- The set of current substitution till this context;
- The Prolog call stack-trace till this `ExecutionContext`;
- The custom-data;
- The utility methods to create a sub-solver;
- The `apply` method that handle side-effects;

ConcurrentExecutionContext It is the newly created entity used by concurrent solvers which store, as the `ExecutionContext`, the information needed in the resolution process. The main difference with the classic context is that herein there is at most one rule and at most one primitive response.

SolverFactory This already present entity has the duty of creating solvers giving standard methods. It contains many overloads to provide a huge reusability and flexibility. Furthermore, this abstraction holds the default data structures needed to create a solver.

ConcurrentSolverFactory This component, like the `SolverFactory`, lets create solvers but, conversely, it instantiates concurrent ones.

Utilities These abstractions define all the methods and data structures needed in the resolution process. These entities are meant for raising up the abstraction from the platform.

These abstractions are meant for meeting as many requirements as possible. The design is based on encapsulation and reusability so that it is possible to replace any component without affecting the whole system.

Furthermore the abstraction level is high enough to allow adding a new platform without any change to the solver.

Another important aspect of this design is the possibility of injection of a synchronization mechanism to support the Prolog ISO standard. In the same way, it is possible to add side-effects handlers as well as introducing different schedulers or granularity control entities. The only flaw of our design is that the concurrent PSM, defined in section 4.1.2, needs to be modified to support AND-parallelism, but this change affects only the *Goal Selection* state while leaving the rest untouched.

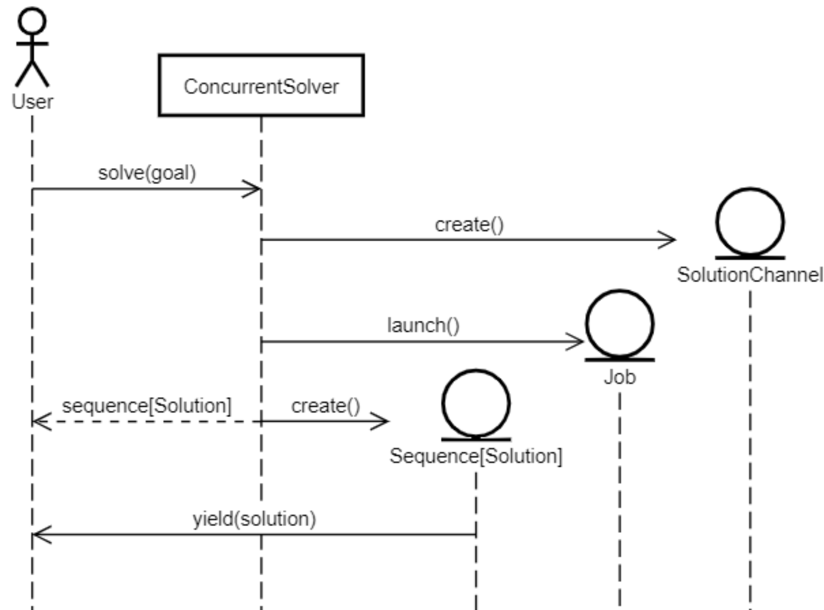

Figure 4.3: Overview of user consuming solutions

Interactions Another important part of our design is how the components of the system interact with each other. Here we call a component each entity that has its own control flow. There are four main components: *User*, *Job*, *SolutionChannel* and *Sequence*. The *User* in our system is whoever uses the solver to compute an answer of a query. This component can be a real user by a GUI, but it can also be another entity with its own control flow. The user, after submitting a query to the concurrent solver, consumes solutions from the sequence as shown in fig. 4.3

We design our concurrent solver to handle multiple control flow. In fact each resolution branch has its own control flow. We call *Job* each separate resolution branch. As can be seen in fig. 4.4, this component has to compute the next states of the FSM. For each state it checks whether it is an end state or not. In the former case the Job sends the solution of the state to the channel. In other cases a new Job is launched, which behaves in the same way.

The communication between each entity of the system takes place by means of the *SolutionChannel*. This component is passive, meaning that it never starts an interaction with another entity but, instead, it can only receive requests. The channel behaves as a temporary store for solutions that the *Jobs* send. The last

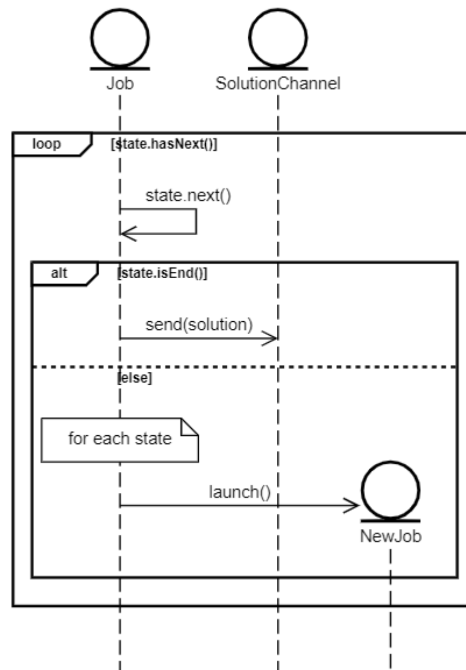

Figure 4.4: Resolution Job and Channel interaction

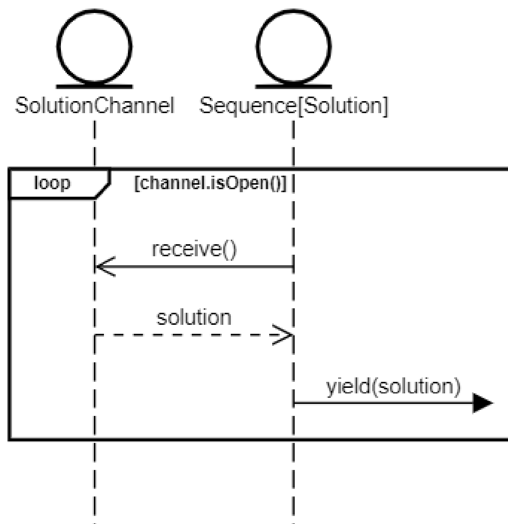

Figure 4.5: Channel and Sequence interaction

main component of our system is the *Sequence*. As can be seen in fig. 4.5, this entity is in charge of consuming the solutions from the channel, yielding each one to the user.

Chapter 5

Implementation

In this chapter we report the relevant implementation parts, explaining in detail the most delicate aspects. To help the readers understand what we do, some snippets of code are shown.

5.1 Execution Context

The first entity we report is the `ConcurrentExecutionContext` which is the container of all the information required during the resolution process, each referring to a specific step. The core data structures it holds are:

- `substitution`: The substitution unifier up to the current resolution step;
- `staticKb`: The static knowledge base, but also the other source of knowledge;
- `flags` and `operators`: The set of flags and operators defined for the resolution;
- `*Channels`: The input, output and warning channels;
- `customData`: User defined custom data;
- `goals`: The list of goals to be solved;
- `rule`: An optional rule of the current resolution step;
- `primitive`: An optional primitive of the current resolution step.

This context contains other utility information required during the computation. It is not only a data container, in fact another important part is the `apply` method

Listing 5.1: Apply method of the ExecutionContext

```
fun apply(sideEffects: Iterable<SideEffect>): ExecutionContext {
  var current = this
  for (sideEffect in sideEffects) {
    current = sideEffect.applyTo(current)
  }
  return current
}
```

and its overloads. As shown in listing 5.1, this function takes a list of `SideEffect` and returns a new `ExecutionContext` obtained applying all the side-effects.

The `ConcurrentExecutionContext`, which implements this interface, overrides the `apply` method casting the returned object to its concurrent version while the concrete class overrides `createSolver` to output a `ConcurrentSolverImpl`.

5.2 Concurrent Solver

The core of the module `:solve-concurrent` is the `ConcurrentSolverImpl` which extends `AbstractSolver`.

Intuitively, it is a concurrent version of the classic solver where the resolution process is based on a state machine but, instead of moving sequentially between states, this solver launches a new coroutine for each next state creating many disjoint resolution branches.

We report the most relevant methods of the `ConcurrentSolverImpl` with their respective snippets of code.

`solveImpl` shown in listing 5.2: It is the override of its abstract version. It requires a goal and the options to solve it, and output a sequence of solutions. This method calls the `solveConcurrently` one, converts the channel to a sequence and ensures that at most there is one negative solution in the output sequence.
The conversion of a Kotlin channel to a sequence is platform dependent so, to unbound the implementation from specific technologies, we use an `expect` function.

Listing 5.2: The solveImpl method

```
override fun solveImpl(goal: Struct, options: SolveOptions):
    Sequence<Solution> {
  return solveConcurrently(goal, options).toSequence().
      ensureAtMostOneNegative()
}
```

Listing 5.3: How solveConcurrently handle the solutions channel

```
override fun solveConcurrently(goal: Struct, options: SolveOptions
    ): ReceiveChannel<Solution> {
  val channel = KtChannel<Solution>(KtChannel.UNLIMITED)
  val initialState = initialState(goal, options)
  val handle = ConcurrentResolutionHandle(options, channel)
  val resolutionScope = createScope()
  resolutionScope.launch {
    startAsyncResolution(initialState, handle)
  }
  return channel
}
```

`solveConcurrently` shown in listing 5.3: It is the heart of this concurrent solver. This method takes a goal and the options to solve it and returns the channel where the solutions are sent. It instantiates the channel, the initial state, the handler utility and the resolution scope. After that, it launches a coroutine where the resolution is started.

`startAsyncResolution` shown in listing 5.4: It is the method that launches the first resolution step, the `handleAsyncStateTransition`, and then suspends until it completes. The completion of the computation occurs when all the child jobs complete and, hence, no more steps are available. At this point, using the handler utility, the output channel is closed.

`handleAsyncStateTransition` shown in listing 5.5: It is a `CoroutineScope` extension method which requires a state and the handler utility and returns its `Job`. It is the engine of the state machine, in fact, it iterates the next states and for each call itself recursively. Furthermore, this method checks if the current state is an `EndState` and, in that case, publishes its solution.

`publishSolutionAndTerminateResolutionIfNeed` shown in listing 5.6: It is the

Listing 5.4: startAsync in detail

```
private suspend fun startAsyncResolution(initialState: State,
    handle: ConcurrentResolutionHandle) = coroutineScope {
  handleAsyncStateTransition(initialState, handle).join()
  handle.closeSolutionChannelWithNoSolutionIfNeeded(initialState.
      context.query)
}
```

Listing 5.5: coroutines launch for each next possible state

```
private fun CoroutineScope.handleAsyncStateTransition(state: State
    , handle: ConcurrentResolutionHandle): Job =
  launch {
    if (state is EndState) {
      handle.publishSolutionAndTerminateResolutionIfNeed(state.
          solution, this)
    } else {
      for (it in state.next()) {
        handleAsyncStateTransition(it, handle)
        yield()
      }
    }
  }
```

Listing 5.6: How solutions are published when needed

```
suspend fun publishSolutionAndTerminateResolutionIfNeed(
  solution: Solution,
  resolutionScope: CoroutineScope
): Boolean {
  if (solutionChannel.isClosedForSend) return false
  solutionChannel.send(solution)
  if (!solution.isNo && solutionCounter.incAndGet() >=
      solveOptions.limit && solveOptions.isLimited) {
    terminateResolution(resolutionScope)
  }
  return true
}
```

most important part of `ConcurrentResolutionHandle`, which is the handler utility class. This method has the duty of sending solutions to the output channel and, to do so, it first verifies whether the channel is still open and then checks if the solutions limit has been reached, terminating the resolution prematurely in that case.
In the same class there are other utility methods used to:

- Terminate the execution of the resolution.
- Append a negative solution.
- Close the output channel.

The `ConcurrentSolverImpl` respects the design defined in chapter 4 but it can be improved with a small refactoring to extract the logic of coroutine launch. In this way we can add a scheduler and a granularity controller.

Instead, the mechanism to preserve the Prolog semantics is a bit more tricky and affects both the solver and the execution context. In fact, it requires additional information to adopt the *leftmost* principle explained in chapter 2.

5.3 Utils

The most complex part we tackle in this thesis is the conversion of a Kotlin channel to a sequence, which is platform specific.

This conversion bridges the concurrent solver to the abstract one. To do this

Listing 5.7: Jvm conversion of a channel to a sequence

```
@OptIn(ExperimentalCoroutinesApi::class)
actual fun <T> ReceiveChannel<T>.toSequence(coroutineScope:
    CoroutineScope): Sequence<T> {
    val queue = LinkedBlockingQueue<Any?>()
    coroutineScope.launch {
        val iterator = this@toSequence.iterator()
        try {
            while (iterator.hasNext()) {
                queue.add(iterator.next())
            }
        } catch (e: Throwable) {
            e.printStackTrace()
            throw e
        } finally {
            queue.add(PoisonPill)
        }
    }
    return sequence {
        while (true) {
            val current = queue.take()
            if (current == PoisonPill) {
                break
            } else {
                @Suppress("UNCHECKED_CAST")
                yield(current as T)
            }
        }
    }
}
```

conversion we define, in a utility class, an `expect` function called `toSequence` which is an extension of `ReceiveChannel`.

2P-Kt currently supports both Jvm and Js but the implementation for the latter platform requires some machinery so we focused only on the Jvm version. The `actual` implementation, shown in listing 5.7, firstly instantiates a jvm data structure, the `LinkedBlockingQueue`, and then features two parts:

- The first part launches a coroutine which, through the channel iterator, consumes each element of the channel. These are added to the queue. This coroutine terminates when the channel is closed and all the elements are consumed. Before terminating, it adds a `PoisonPill` to the queue. We use a detached `CoroutineScope` for this coroutine to isolate as much as possible the resolution from the conversion.
- The second part returns a lazy sequence which takes an element from the queue and yields it. The sequence terminates when the `PoisonPill` is encountered. This computation executes on the caller's thread which usually is the main one as in the sequential version.

There are two side notes we want to highlight about this specific method. The first is that it is annotated with `@OptIn` which is the mechanism for requiring and giving explicit consent for using certain elements of APIs. The channel API may change at any time, meaning that also this implementation may change at any time. The same annotation is present in the code where channels are used.

The other side note we highlight is referred to the annotation `@Suppress` for the unchecked cast over the `yield` method. This warning derives from the generic implementation, useful for the reusability of the code.

5.4 Concurrent State Machine

Our solver is based on a non-deterministic state machine. To develop this state machine we define three interfaces: `State`, `EndState` and `ExceptionalState`. Each concrete class has to implement one of these interfaces, based on which is the specific state.

The `State` interface contains the `next` method, which returns a list of states, and some utility functions. The non-determinism of our concurrent solver dwells in the return type of the core function of this interface. In fact the returned list can contain from zero to infinite states. For each element of the list a new resolution branch is launched.

`EndState` is the interface that describes a sink state of the concurrent state machine. It contains a solution and its next method always returns an empty list. `ExceptionalState` is an extension of the `EndState` that carries with it an exception thrown during the execution.

As described in chapter 4, our state machine features eight states:

- `StateGoalSelection`
- `StatePrimitiveSelection`
- `StatePrimitiveExecution`
- `StateRuleSelection`
- `StateRuleExecution`
- `StateEnd` which implement `EndState`
- `StateHalt` which implement `ExceptionalState`
- `StateException` which implement `ExceptionalState`

For each state we briefly describe the behaviour of the `next` method highlighting its important aspects.

`StateGoalSelection` handles three cases:

- The goal is over and the context is root: the returned list contains a `StateEnd` with a positive solution.
- The goal is over but the context is not root: the execution context is popped from the stack and a new `StateGoalSelection` is returned.
- The goal is not over: the substitution is applied to the goal and a `StatePrimitiveSelection` is returned.

`StatePrimitiveSelection` when the current goal is a *struct* a matching primitive is looked for in the knowledge base. If a match is found, the solve method of that matching primitive is called and each response is mapped to a `StatePrimitiveExecution`. Conversely, when no match is found, the returned state is `StateRuleSelection`.

`StatePrimitiveExecution` handles the three possible primitive responses. When it is a `Solution.Yes` the state machine moves to `StateGoalSelection`. When the primitive response is `Solution.No` it goes to `StateEnd` with a negative solution. Instead, the `Solution.Halt` is converted to `StateException`.

`StateRuleSelection` works similarly to the selection of a primitive. The knowledge base is plumbed for matching rules. Each rule found is mapped to `StateRuleExecution`.

`StateRuleExecution` tries to unify the current goal with the head of the rule. If it succeeds the returned state is `StateGoalSelection` while, in case of failure, a `StateEnd` with `Solution.No` is returned.

`StateEnd` simply returns an empty list.

`StateHalt` forces the end of the computation throwing an exception.

`StateException` checks whether the current goal is a *Catch* and it unifies with the exception. If this is the case the returned state is `StateGoalSelection`. Otherwise the exception is handled in the parent context or, when the root is reached, the state machine moves to `StateHalt`.

Chapter 6

Validation

In this chapter we present the validation process, explaining in detail the testing ecosystem. Furthermore, we report the test metrics and some benchmarks to compare our concurrent solver with the classic one.

6.1 Test framework

2P-Kt contains a specific module for testing solvers which is designed for reusability. The problem is that it works for solvers compliant with the Prolog ISO standard.

Our base implementation works concurrently but for now it does not feature a mechanism for the solution ordering. This forces us to design a more abstract and reusable test framework.

The purpose of unit tests for concurrent solvers is to verify the amount and type of solutions produced, while their order may vary. To fulfill this goal, we define two interfaces:

`WithAssertingEquals`: This interface exposes the `assertingEquals` method which is used like `equals` but, instead of returning a Boolean, it checks equality through the assertion methods.

`FromSequence`: This interface is generic in `T` with `WithAssertingEquals` upper bound and extends `SolverTest`. It exposes the function `fromSequence` which takes a list of solutions and returns an instance of the generic type.

Listing 6.1: Unit test example

```
interface TestConcurrentTrue<T : WithAssertingEquals> :
    FromSequence<T>, SolverFactory {
    fun testTrue() {
        prolog {
            val solver = solverWithDefaultBuiltins()
            val query = atomOf("true")
            val solutions = fromSequence(solver.solve(query,
                mediumDuration))
            val expected = fromSequence(sequenceOf(query.yes()))

            expected.assertingEquals(solutions)
        }
    }
}
```

Listing 6.2: Unit test concrete class example

```
class TestConcurrentTrueImpl :
    TestConcurrentTrue<MultiSet>,
    SolverFactory by ConcurrentSolverFactory,
    FromSequence<MultiSet> by ConcurrentFromSequence {
    @OptIn(ExperimentalCoroutinesApi::class)
    @Test
    override fun testTrue() = multiRunConcurrentTest { super.
        testTrue() }
}
```

These interfaces let us define abstract and reusable tests. The other part of our framework are the unit tests. We define an interface for each Prolog construct, generic in `T` with `WithAssertingEquals` upper bound, which extends the `FromSequence` casting its generic also to `T`. Inside these interfaces we implement a function for each unit test of the construct we want to verify.

In listing 6.1 we show the unit test of the *True* construct, one of the most straightforward. The body of each unit test is standard, firstly the solver is instantiated, then the query is defined and it is solved. The sequence of solutions is converted to the utility class that implements the `WithAssertingEquals` interface and it is compared by the `assertingEquals` method to the expected result.

To use this framework the only part that needs to be implemented is the concrete class of each Prolog construct. The example of the *True* construct test class is shown in listing 6.2. The class has to extend the respective interface, but also the `SolverFactory` and the `FromSequence` to cast the generic type. In this class we override all the unit test functions to add the `@Test` annotation and call the equivalent `super` method. The last detail that can be seen in the code is the `multiRunConcurrentTest` which is a method to run the same body multiple times to check that there are no deadlock or other concurrency problems.

To reuse this framework it is enough to define a class that implements the interface `WithAssertingEquals` defining the equality assertion based on the need. The other pieces that need to be implemented are the concrete class for each construct.

6.2 Metrics and benchmarks

The validation of the concurrent solver is, probably, the most important part of the system. In fact, we put a lot of effort into it to reach a great number of unit tests.

Unit tests		
Passed tests	Skipped tests	Total tests
286	53	339

Table 6.1: Test metrics

In table 6.1, we report the number of unit tests. As can be seen, most of them pass, but 53 are skipped. These come from some constructs we have not implemented because they contain side effects like, for example, the *cut*. All the other constructs are tested successfully.

Machine with two physical core			
Prolog program	Sequential time	Concurrent time	Speedup
7-Queens	2,23	1,58	1,41
8-Queens	10,05	5,97	1,68
9-Queens	—	28,96	—

Table 6.2: Benchmarks comparing solvers on a two physical core machine

Machine with four physical core			
Prolog program	Sequential time	Concurrent time	Speedup
7-Queens	2,39	1,14	2,10
8-Queens	9,30	4,30	2,16
9-Queens	45,38	20,62	2,20

Table 6.3: Benchmarks comparing solvers on a four physical core machine

Furthermore, we make some benchmarks to compare our concurrent solver with the sequential one. We use two different machines to run the benchmarks:

- A two physical core machines: its CPU is an i7-6th generation Intel core at 2,5GHz and 12GB RAM.
- A four physical core machines: its CPU is an i7-8th generation Intel core at 1,8GHz and 16GB RAM.

In table 6.2 are reported execution times of both types of solvers and the speedup achieved using the two physical core machines. The same benchmarks are executed on the four physical core machines and the results are reported in table 6.3.

We use the problem of n-queens with different values of n. This problem comes from chess and requires placing n queens on a board with size $n \times n$ so that none attack each other. We use this specific program because it has a good grade of parallelism and it requires a lot of computation. We test the same program with different values of n to better understand how our concurrent solver performs.

The results show a good speedup, which means that concurrency works efficiently. Nevertheless, there are three important points we want to highlight:

- The classic solver execution time for 9-queens in the two physical core machines is missing because the computation fails due to a stack overflow. This prevents us from comparing the results and, hence, to calculate the speedup.
- Our solver is a *work-in-progress* and features only the basic OR-parallelism. This means that performance can improve significantly adding a granularity control mechanism or a scheduler or AND-parallelism.
- We use only the *n-queens* for our benchmarks because it is a well known problem. Other CPU-intensive Prolog programs may perform differently, depending on the grade of parallelism.

We do not compare performance of a non CPU-intensive Prolog program because we know that the concurrent solver performs worse than the sequential one. This is due to the overhead introduced by the handling of threads and coroutines.

Chapter 7

Conclusions

In this thesis, our main goal is to extend the 2P-Kt ecosystem to support the concurrent logic programming. To fulfill it, we proceed by steps. Firstly we study in details the state of the art about parallelism in logic programming to understand what and how to do to reach our goal. Due to the great number of problems that arise from each type of parallelism, we decide to focus on the OR-parallelism. This, with respect to the other type of parallelism, entails the best improvements of performances.

The next step is to design the concurrent solver reusing as much as possible the 2P-Kt ecosystem. We decide to model our solver's core logic as a state machine. Concurrency in our solver entails nondeterminism and, hence, also the state machine is nondeterministic. In fact, we have to handle a list of state at each transition, possibly empty, instead of just one and, to do so, we manage the concurrent resolution branches. Furthermore, we design each component of our solver to be encapsulated and isolated as much as possible. This helps us to fulfill the requirement of extensibility of the solver.

After the design step, we move to the implementation phase, where we handle technical problems. We firstly implement the nondeterministic state machine, following the design and, then, we work on the concurrent solver. To conform our solver to the already present interfaces and abstraction, we have to create the bridge from the concurrent world to the sequential one. This connection is possible converting a channel to a sequence, which is a platform specific implementation.

The last step is to verify the correct functioning of the solver. We develop a test framework that lets reuse unit tests without much effort. It features tests for all the Prolog constructs. We use this environment to test our solver but, because it does not support the Prolog semantics, we have to skip the tests of the constructs that causes side effects.

After the solver is done, we make some benchmarks to compare it to the sequential one in terms of performances, which can be seen in chapter 6. The results are far better than what we expect. We explain the two relevant improvement achieved: *Speedup* and *Memory usage*.

Speedup The execution time decreases significantly with a CPU-intensive Prolog program with a high grade of OR-parallelism. However the speedup rate is far from the linearity for at least three reasons:

- We do not implement the AND-parallelism mechanism, losing a significant amount of speedup.
- A granularity control mechanism can reduce the overhead of spawning and handling a coroutine when it is not needed.
- A scheduler can further improve performances because it can balance the workload.

Memory usage The amount of memory needed during the resolution is reduced. This is evident when, executing the *9-queens* Prolog program, the sequential solver throws a stack overflow exception, while our concurrent solver succeeds. This comes from the removal of choice points which take up a fair amount of memory.

To conclude this thesis, we have met the goals defined in chapter 3. Our design defines an abstract layer that gives an application programming interface that can be reused and extended to create new concurrent solvers. We then develop a *pure* OR-concurrent solver. Our implementation is extensible so that new mechanisms can be added. We have a high abstraction level so that the injection of new functionalities does not entail any change to our base solver.

7.1 Open issues

Due to its nature of being a base for future implementations, our solver is not completed yet and many open issues are present. We list the most relevant ones with a brief overview and we provide some tips on how to address them: *AND-parallelism*, *Granularity control*, *Scheduler*, *Memory usage*, *Preprocessing*, *Js channel conversion* and *Smart solver controller*

AND-parallelism: This means that each sub-goal is executed in parallel. It involves many synchronization problems that need to be handled. A first version should focus only on this specific type of parallelism, which can be developed creating a list of state when the next goal is selected instead of picking one at a time. After that, the OR-parallelism and the AND-parallelism can be mixed together addressing the arising problems.

Granularity control: A mechanism that tries to estimate the size of each resolution branch and that decides if it is useful to spawn a coroutine. This mechanism would require an analysis at compile time of the program to generate a partial resolution tree and a set of checks to evaluate at runtime.

Scheduler: This component would balance the workload. This work is based on the granularity control mechanism, whose results would be used by the scheduler to balance how many and which piece of work it has to give to each thread.

Memory usage: In the actual implementation an issue is the big memory usage. It comes from the execution context, which contains some debugging data and, thus, can be reduced. To further decrease the memory used, some other data structures can be streamlined.

Preprocessing: Analysis at compile time are required to enhance the granularity control and the scheduler. This mechanism has to parse the Prolog program to make estimates, which can be done when the knowledge base is loaded.

The *Js channel conversion* implementation is missing. This requires some machinery to use multiple threads on this platform.

Smart solver controller: This mechanism is an high level controller that runs a set of checks and, based on the results, it decides which of all the existing solver should be used. This component requires a lot of preprocessing, but

can also be used at run time to pick a sub-solver for each primitive.

Bibliography

[1] ISO/IEC JTC 1/SC 22 Technical Committee. ISO/IEC 13211-1:1995: Information technology — Programming languages — Prolog — Part 1: General core. International Standard ISO/IEC 13211-1, ISO/IEC, 1995.

[2] Khayri A. M. Ali and Roland Karlsson. Full prolog and scheduling or-parallelism in muse. *Int. J. Parallel Program.*, 19(6):445–475, 1990.

[3] Khayri A. M. Ali and Roland Karlsson. The muse or-parallel prolog model and its performance. In Saumya K. Debray and Manuel V. Hermenegildo, editors, *Logic Programming, Proceedings of the 1990 North American Conference, Austin, Texas, USA, October 29 - November 1, 1990*, pages 757–776. MIT Press, 1990.

[4] Francisco Bueno, Maria J. García de la Banda, and Manuel V. Hermenegildo. Effectivness of abstract interpretation in automatic parallelization: A case study in logic programming. *ACM Trans. Program. Lang. Syst.*, 21(2):189–239, 1999.

[5] Mats Carlsson. On implementing prolog in functional programming. *New Generation Computing*, 2(4):347–359, 1984.

[6] Giovanni Ciatto, Roberta Calegari, and Andrea Omicini. 2P-Kt: A logic-based ecosystem for symbolic AI. *SoftwareX*, 16:100817:1–7, December 2021.

[7] Giovanni Ciatto, Roberta Calegari, Enrico Siboni, Enrico Denti, and Andrea Omicini. 2P-Kt: logic programming with objects & functions in kotlin. In Roberta Calegari, Giovanni Ciatto, Enrico Denti, Andrea Omicini, and Giovanni Sartor, editors, *WOA 2020 – 21th Workshop "From Objects to Agents"*,

volume 2706 of *CEUR Workshop Proceedings*, pages 219–236, Aachen, Germany, October 2020. Sun SITE Central Europe, RWTH Aachen University.

[8] Keith L. Clark. Negation as failure. In Hervé Gallaire and Jack Minker, editors, *Logic and Data Bases, Symposium on Logic and Data Bases, Centre d'études et de recherches de Toulouse, France, 1977*, Advances in Data Base Theory, pages 293–322, New York, 1977. Plemum Press.

[9] Vítor Santos Costa, Ricardo Bianchini, and Inês de Castro Dutra. Evaluating parallel logic programming systems on scalable multiprocessors. In Hoon Hong, Erich Kaltofen, and Markus A. Hitz, editors, *Proceedings of the 2nd International Workshop on Parallel Symbolic Computation, PASCO 1997, July 20-22, 1997, Kihei, Hawaii, USA*, pages 58–67. ACM, 1997.

[10] Inês de Castro Dutra. Distributing and-work and or-work in parallel logic programming systems. In *29th Annual Hawaii International Conference on System Sciences (HICSS-29), January 3-6, 1996, Maui, Hawaii, USA*, pages 646–655. IEEE Computer Society, 1996.

[11] Jacques Chassin de Kergommeaux and Philippe Codognet. Parallel logic programming systems. *ACM Comput. Surv.*, 26(3):295–336, 1994.

[12] Enrico Denti, Andrea Omicini, and Alessandro Ricci. tuProlog: A lightweight Prolog for Internet applications and infrastructures. In I.V. Ramakrishnan, editor, *Practical Aspects of Declarative Languages*, volume 1990 of *Lecture Notes in Computer Science*, pages 184–198. Springer Berlin Heidelberg, 2001. 3rd International Symposium (PADL 2001), Las Vegas, NV, USA, 11–12 March 2001. Proceedings.

[13] Enrico Denti, Andrea Omicini, and Alessandro Ricci. Multi-paradigm Java-Prolog integration in tuProlog. *Science of Computer Programming*, 57(2):217–250, August 2005.

[14] M. J. Fernández, Manuel Carro, and Manuel V. Hermenegildo. IDRA (ideal resource allocation): Computing ideal speedups in parallel logic programming. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors,

Euro-Par '96 Parallel Processing, Second International Euro-Par Conference, Lyon, France, August 26-29, 1996, Proceedings, Volume II, volume 1124 of *Lecture Notes in Computer Science*, pages 724–733. Springer, 1996.

[15] Gopal Gupta and Vítor Santos Costa. Complete and efficient methods for supporting side-effects and cuts in and-or parallel prolog. In *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing, SPDP 1992, Arlington, Texas, USA, December 1-4, 1992*, pages 288–295. IEEE Computer Society, 1992.

[16] Gopal Gupta and Vítor Santos Costa. Cuts and side-effects in and-or parallel prolog. *J. Log. Program.*, 27(1):45–71, 1996.

[17] Gopal Gupta and Bharat Jayaraman. Analysis of or-parallel execution models. *ACM Trans. Program. Lang. Syst.*, 15(4):659–680, 1993.

[18] Gopal Gupta and Enrico Pontelli. High performance parallel logic programming: The ACE parallel prolog system. In Moreno Falaschi, Marisa Navarro, and Alberto Policriti, editors, *1997 Joint Conf. on Declarative Programming, APPIA-GULP-PRODE'97, Grado, Italy, June 16-19, 1997*, pages 25–32, 1997.

[19] Gopal Gupta, Enrico Pontelli, Khayri A. M. Ali, Mats Carlsson, and Manuel V. Hermenegildo. Parallel execution of prolog programs: a survey. *ACM Trans. Program. Lang. Syst.*, 23(4):472–602, 2001.

[20] Manuel V. Hermenegildo, Francisco Bueno, Manuel Carro, Pedro López-García, Edison Mera, José F. Morales, and Germán Puebla. An overview of ciao and its design philosophy. *Theory Pract. Log. Program.*, 12(1-2):219–252, 2012.

[21] Zhiyi Huang, Chengzheng Sun, and Abdul Sattar. Selective recomputation for handling side-effects in parallel logic programs. In Hugh Glaser, Pieter H. Hartel, and Herbert Kuchen, editors, *Programming Languages: Implementations, Logics, and Programs, 9th International Symposium, PLILP'97, Including a Special Trach on Declarative Programming Languages in Education,*

Southampton, UK, September 3-5, 1997, Proceedings, volume 1292 of *Lecture Notes in Computer Science*, pages 275–289. Springer, 1997.

[22] Matthew M. Huntbach and Graem A. Ringwood. Programming in concurrent logic languages. *IEEE Softw.*, 12(6):71–81, 1995.

[23] Philipp Körner, Michael Beuschel, João Barbosa, Vítor Santos Costa, Verónica Dahl, Manuel V. Hermenegildo, Jose F. Morales, Jan Wielemaker, Daniel Diaz, Salvador Abreu, and Giovanni Ciatto. A multi-walk through the past, present and future of prolog. *(Submitted to) Theory and Practice of Logic Programming (TPLP)*, 2021.

[24] Pedro López-García, Manuel V. Hermenegildo, and Saumya K. Debray. A methodology for granularity-based control of parallelism in logic programs. *J. Symb. Comput.*, 21(4):715–734, 1996.

[25] Ewing L. Lusk, Ralph Butler, Terrence Disz, Robert Olson, Ross A. Overbeek, Rick Stevens, David H. D. Warren, Alan Calderwood, Péter Szeredi, and Seif Haridi. The aurora or-parallel prolog system. *New Gener. Comput.*, 7(2-3):243–271, 1990.

[26] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, April 1982.

[27] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[28] Giulio Piancastelli, Alex Benini, Andrea Omicini, and Alessandro Ricci. The architecture and design of a malleable object-oriented Prolog engine. In Roger L. Wainwright, Hisham M. Haddad, Ronaldo Menezes, and Mirko Viroli, editors, *23rd ACM Symposium on Applied Computing (SAC 2008)*, volume 1, pages 191–197, Fortaleza, Ceará, Brazil, 16–20 March 2008. ACM. Special Track on Programming Languages.

[29] Enrico Pontelli and Gopal Gupta. Data parallel logic programming in &ace. In *Proceedings of the Seventh IEEE Symposium on Parallel and Distributed*

Processing, SPDP 1995, San Antonio, Texas , USA, October 25-28, 1995, pages 424–431. IEEE, 1995.

[30] Enrico Pontelli and Gopal Gupta. On the duality between or-parallelism and and-parallelism in logic programming. In Seif Haridi, Khayri A. M. Ali, and Peter Magnusson, editors, *Euro-Par '95 Parallel Processing, First International Euro-Par Conference, Stockholm, Sweden, August 29-31, 1995, Proceedings*, volume 966 of *Lecture Notes in Computer Science*, pages 43–54. Springer, 1995.

[31] Enrico Pontelli and Gopal Gupta. Parallel symbolic computation in ACE. *Ann. Math. Artif. Intell.*, 21(2-4):359–395, 1997.

[32] Enrico Pontelli and Gopal Gupta. Efficient backtracking in and-parallel implementations of non-deterministic languages. In *1998 International Conference on Parallel Processing (ICPP '98), 10-14 August 1998, Minneapolis, Minnesota, USA, Proceedings*, pages 338–345. IEEE Computer Society, 1998.

[33] Enrico Pontelli and Gopal Gupta. Backtracking in independent and-parallel implementations of logic programming languages. *IEEE Trans. Parallel Distributed Syst.*, 12(11):1169–1189, 2001.

[34] Enrico Pontelli, Gopal Gupta, and Manuel V. Hermenegildo. &ace: a high-performance parallel prolog system. In *Proceedings of IPPS '95, The 9th International Parallel Processing Symposium, April 25-28, 1995, Santa Barbara, California, USA*, pages 564–571. IEEE Computer Society, 1995.

[35] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, January 1965.

[36] Ehud Shapiro. The family of concurrent logic programming languages. *ACM Comput. Surv.*, 21(3):413–510, 1989.

[37] Marcio G. Silva, Inês de Castro Dutra, Ricardo Bianchini, and Vítor Santos Costa. The influence of architectural parameters on the performance of parallel logic programming systems. In Gopal Gupta, editor, *Practical Aspects of*

Declarative Languages, First International Workshop, PADL '99, San Antonio, Texas, USA, January 18-19, 1999, Proceedings, volume 1551 of *Lecture Notes in Computer Science*, pages 122–136. Springer, 1999.

[38] Rui Vieira, Ricardo Rocha, and Fernando M. A. Silva. Or-parallel prolog execution on multicores based on stack splitting. In Umut A. Acar and Vítor Santos Costa, editors, *Proceedings of the POPL 2012 Workshop on Declarative Aspects of Multicore Programming, DAMP 2012, Philadelphia, PA, USA, Saturday, January 28, 2012*, pages 1–10. ACM, 2012.

[39] Yao Wu, Enrico Pontelli, and Desh Ranjan. Computational issues in exploiting dependent and-parallelism in logic programming: Leftness detection in dynamic search trees. In Geoff Sutcliffe and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference, LPAR 2005, Montego Bay, Jamaica, December 2-6, 2005, Proceedings*, volume 3835 of *Lecture Notes in Computer Science*, pages 79–94. Springer, 2005.

[40] Kang Zhang. A review of exploitation of and-parallelism and combined and/or-parallelism in logic programs. *ACM SIGPLAN Notices*, 29(2):25–32, 1994.

[41] Yuhua Zheng, Honglei Tu, and Li Xie. And/or parallel execution of logic programs: Exploiting dependent and-parallelism. *ACM SIGPLAN Notices*, 28(5):19–28, 1993.