THE UNIVERSITY *of* EDINBURGH

An implementation methodology for using concurrent and collaborative approaches for theorem provers, with case studies of SAT and LCF style provers

Sripriya G

Doctor of Philosophy
Centre for Intelligent Systems and their Applications
School of Informatics
University of Edinburgh

2013

Lay summary

Logic is the study of *formal* (symbolic) systems of *reasoning* (i.e. formal deductive inference) and of methods of attaching meaning to them. Computers enabled(s) the automation of formal reasoning: mechanised reasoning systems (*theorem provers*) are software systems which execute the *reasoning* without or with (partial or step-by-step) human intervention. A typical problem scenario is: given a set of assumptions/axioms (i.e. all the available relevant information about the given problem), A and a conjecture/goal, G (the question being asked), can you find a proof of G, using A? The problem is given to an automated reasoning system to work on until it arrives at an answer or until it runs out of resources or the execution is terminated by the user. Theorem provers have a wide range of applications, e.g. hardware and software verification. These provide significant challenges, in terms of size and complexity of the problems, fueling the need for better theorem provers, capable of handling bigger and more complex problems.

This thesis investigates the *scope and efficacy* of using *concurrent/ distributed programming* paradigms to engineer *better* theorem provers (speed and/or ease of programmability, i.e. implementing one's own proof search procedures, using the system's existing machinery). We have investigated this in the context of two case studies of diverse, representative classes of theorem provers: the propositional satisfiability problem, **SAT** (based on propositional logic; predominantly, fully automated systems; very popular choice for industrial applications; and an active research field) and **LCF** style (first-order) theorem proving (geared towards semi-automated, interactive theorem proving; focuses on programmability).

The improved accessibility of parallel computing power (e.g. multicore machines, GRIDs and better software tools) and saturation of processor speeds of conventional single-processor computers has made *parallelisation* and application of concurrent/distributed paradigms a popular choice and almost an imperative for engineering better/faster systems. Application of concurrent paradigms to theorem provers can provide more processing power. More crucially, it can open up opportunities for implementing novel approaches to address theorem proving tasks hitherto infeasible in a sequential setting. Some such previously unexplored opportunities have been investigated in this thesis, for the two case studies considered. Concurrent techniques have been developed to tap these opportunities and proof-of-concept prototypes have been developed for the same. Empirical results show significant performance gains for the criteria considered, as explained below.

An orthogonal focus of the work has been the *implementation approach* used to apply the techniques. Here is why this has been investigated: Concurrent programming is an established field. However, choosing the most effective concurrent technique to address a given task is a non-trivial task. Theorem proving problems vary a lot in their structure and hardness and can depend on problem-domain, logic of formulation, inference system used etc.. This in turn impacts the efficacy of a given concurrent technique too. So, a generalised solution of concurrent-technique-application is unlikely to work for theorem provers. This is in contrast to some other application domains which have adopted parallelisation, e.g., numerical computation, which possesses a fair amount of regularities which can be exploited for parallelisation. For theorem proving, an iterative, experimental, developmental cycle of application and empirical analysis is required to develop effective application of concurrent techniques, to address specific theorem proving tasks. However, concurrent programming is notoriously error prone, hard to

debug and evaluate. Thus, implementation approaches which promote easy prototyping, portability, incremental development and effective *isolation of design and implementation* can greatly aid the enterprise of experimentation. In this thesis, we have explored one such approach, by using Alice ML, a functional programming language with support for concurrency and distribution, to implement the prototypes. We have used programming abstractions, i.e. a programming construct that captures a (concurrent/sequential) computational pattern, to encapsulate the implementations of the concurrent techniques used. These allow for easy prototyping and code reuse and incremental development. Functional programming languages are known to be particularly well suited for concurrent programming and allow for concise and effective expression of programming abstractions (as higher-order constructs). The utility of this approach is illustrated via the proof-of-concept prototypes of concurrent systems developed for the two diverse case studies of theorem proving investigated in this work, addressing some previously unexplored parallelisation opportunities for each, as described below:

SAT: We have developed two novel, concurrent approaches for SAT and developed prototypes for the same, using Alice ML and employing programming abstractions where appropriate: (1) *DPLL-Stalmarck* is a novel hybrid approach for SAT and uses two complementary SAT-algorithms, DPLL and Stalmarck's, where the two systems run asynchronously and dynamic information exchange is used for co-operative solving. Compared to the standalone DPLL solver, *DPLL-Stalmarck* shows significant performance gains for two of the three problem classes considered and comparable behaviour otherwise. As an exploratory research effort, we have developed a novel algorithm, *Concurrent Stalmarck*, by applying concurrent techniques to the Stalmarck algorithm and early empirical results show significant gains, compared to the (sequential) Stalmarck algorithm. For *DPLL-Stalmarck*, the interaction of the two systems in the asynchronous setting has been encapsulated as a programming abstraction and has been used to experiment with variants of the algorithms used in the individual asynchronous solvers . Implementation of the *saturation* technique of the Stalmarck algorithm in a parallel setting, as implemented in *Concurrent Stalmarck*, has been encapsulated as a programming abstraction.

LCF: Provision of *programmable* concurrent primitives enables customisation of concurrent techniques to specific theorem proving scenarios. We have developed a *multilayered approach* to support programmable, sound extensions for an LCF prover: use programming abstractions to implement the concurrent techniques; use these to develop novel tacticals (control structures to apply tactics; a tactic is an encapsulation of an inference rule), incorporating concurrent techniques; and use these to develop novel proof search procedures. This approach has been implemented in a prototypical LCF style first-order prover, using Alice ML. New tacticals developed are: fastest-first; distributed composition; crossTalk: a novel tactic which uses dynamic, collaborative information exchange to handle unification across multiple sub-goals, with shared meta-variables; a new tactic, performing simultaneous proof-refutation attempts on propositional (sub-)goals, by invoking an external SAT solver (SAT case study), as a counter-example finder. Examples of concrete theorem proving scenarios are provided, demonstrating the utility of these extensions. Synthesis of a variety of automatic proof search procedures has been demonstrated, illustrating the scope of programmability and customisation, enabled by our multilayered approach.

Abstract

Theorem provers are faced with the challenges of size and complexity, fueled by the increasing range of applications. The use of concurrent/ distributed programming paradigms to engineer better theorem provers merits serious investigation, as it provides: more processing power and opportunities for implementing novel approaches to address theorem proving tasks hitherto infeasible in a sequential setting. Investigation of these opportunities for two diverse theorem prover settings with an emphasis on desirable implementation criteria is the core focus of this thesis.

Concurrent programming is notoriously error prone, hard to debug and evaluate. Thus, implementation approaches which promote easy prototyping, portability, incremental development and effective isolation of design and implementation can greatly aid the enterprise of experimentation with the application of concurrent techniques to address specific theorem proving tasks. In this thesis, we have explored one such approach by using Alice ML, a functional programming language with support for concurrency and distribution, to implement the prototypes and have used programming abstractions to encapsulate the implementations of the concurrent techniques used. The utility of this approach is illustrated via proof-of-concept prototypes of concurrent systems for two diverse case studies of theorem proving: the propositional satisfiability problem (SAT) and LCF style (first-order) theorem proving, addressing some previously unexplored parallelisation opportunities for each, as follows:.

SAT: We have developed a novel hybrid approach for SAT and implemented a prototype for the same: *DPLL-Stalmarck*. It uses two complementary algorithms for SAT, DPLL and Stalmarck's. The two solvers run asynchronously and dynamic information exchange is used for co-operative solving. Interaction of the solvers has been encapsulated as a programming abstraction. Compared to the standalone DPLL solver, *DPLL-Stalmarck* shows significant performance gains for two of the three problem classes considered and comparable behaviour otherwise. As an exploratory research effort, we have developed a novel algorithm, *Concurrent Stalmarck*, by applying concurrent techniques to the Stalmarck algorithm. A proof-of-concept prototype for the same has been implemented. Implementation of the *saturation* technique of the Stalmarck algorithm in a parallel setting, as implemented in *Concurrent Stalmarck*, has been encapsulated as a programming abstraction.

LCF: Provision of *programmable* concurrent primitives enables customisation of concurrent techniques to specific theorem proving scenarios. In this case study, we have developed a multilayered approach to support programmable, sound extensions for an LCF prover: use programming abstractions to implement the concurrent techniques; use these to develop novel tacticals (control structures to apply tactics), incorporating concurrent techniques; and use these to develop novel proof search procedures. This approach has been implemented in a prototypical LCF style first-order prover, using Alice ML. New tacticals developed are: fastest-first; distributed composition; crossTalk: a novel tactic which uses dynamic, collaborative information exchange to handle unification across multiple sub-goals, with shared meta-variables; a new tactic, performing simultaneous proof-refutation attempts on propositional (sub-)goals, by invoking an external SAT solver (SAT case study), as a counter-example finder. Examples of concrete theorem proving scenarios are provided, demonstrating the utility of these extensions. Synthesis of a variety of automatic proof search procedures has been demonstrated, illustrating the scope of programmability and customisation, enabled by our multilayered approach.

Acknowledgements

My PhD supervisors, Prof Alan Bundy and Dr Alan Smaill, with their infinite patience, encouragement and support have made the PhD journey (despite the many uncertainties involved) a pleasant and enriching one and I am earnestly grateful for that. They have helped me to acquire the invaluable perspective of seeing the Phd project not just as an end in itself, but more as a means to an end, of it being training in research. I would like to thank Alan Bundy for all his patient support and encouragement and for being a lighthouse always through out this journey and for sharing his knowledge and wisdom of the subject and research in general. Alan Smaill for bringing the Alice ML language to our attention leading to its subsequent use in this project, for all his clever and useful insights and ideas in shaping this project, for being an enormous source of encouragement and support and for being so generous with his time.

My Thanks to Dr Makarius Wenzel, formerly of Technical University of Munich, Germany for sharing my early interest and excitement of doing parallel theorem proving and for his efforts to facilitate the same in the Isabelle system. I am very much grateful to Dr Andreas Rossberg, the key architect of the Alice ML language, for all his enthusiasm and support in answering my long list of questions.

I am profoundly grateful to different parts of the University of Edinburgh: the Principal's scholarship, CISA, Informatics Graduate school and the ORS programme for the funding support extended for my PhD and to the Informatics Graduate school for their flexibility, understanding and support.

And to Aditya, for all his patience, understanding and support in various forms in this entire journey that this Phd project has been.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(Sripriya G)

Table of Contents

List of Figures

List of Tables

List of code samples

Chapter 1

Introduction

> The tools that we use have a profound influence on our thinking habits and therefore on our thinking abilities
>
> *-Edsger Dijkstra*

In a similar vein, it can be said that our thinking habits inspire the tools that we create and more pertinently so for a domain like mechanised reasoning systems. Our thinking patterns are not always sequential or linear, why should the tools that we create be so?

1.1 Why should parallelisation of theorem provers be considered?

The field of mechanised reasoning systems (theorem proving), with its ever increasing applications, is faced with challenges of complexity and size, i.e. harder and bigger problems. This calls for exploration of research directions that enable engineering of better theorem provers that can tackle these challenges. Most theorem prover implementations and the underpinning techniques were developed for a sequential mode of execution, which, in turn, has limited the possibilities of the approaches employed as well as their implementations.

Application of concurrent and distributed programming techniques [1] to engineer faster applications is fast becoming an ubiquitous trend across application domains. A strong

[1] In this work, we use the terms *concurrent* and *parallel* synonymously to refer to asynchronous processes that execute simultaneously and possibly interact with one another. We use the term *distributed* to refer to the special case where the processes run on different physical machines.

motivation for this approach is the saturation of processor speeds which in turn, means that applications can no longer expect to achieve speedups purely by virtue of being run on a faster processor, a phenomenon discussed in a much cited recent paper titled *The free lunch is over* [Sutter, 2005]. This trend has been fueled by the surge in the accessibility and availability of a wide variety of parallel and distributed computing architectures aided by the emergence of new paradigms of computing and related software that enable optimal utilisation of these emerging computing architectures.

In addition to providing more processing power, the concurrent and distributed programming paradigms can open up *novel* ways of tackling problems that are not possible in a sequential mode of execution. E.g., consider the following scenario: There are multiple choices of computation that can be pursued, the solution possibly occurring in any one of them and where a judgement on the speed of each computation cannot be made beforehand. Tackling this using a sequential mode of execution would typically entail execution of each computation one at a time. This does not help if e.g., the first computation takes a very long time and the solution happens to be in a subsequent computation. However, in a concurrent asynchronous programming paradigm, we can spawn all the computations simultaneously and pick the fastest returning computation. With the improved accessibility and the diversity of emerging architectures, it becomes more interesting now than ever before to investigate novel ways of using these technologies to tackle the challenges faced by today's theorem provers and to identify latent parallelisation, distribution and collaboration opportunities present in theorem prover implementations. This need is echoed in a recent work [Kaufmann and Moore, 2009], where *Parallel, Distributed and Collaborative Theorem Proving* is cited as being one of the key research problems for automated theorem proving.

1.2 Implementation methodology for application of concurrent techniques to theorem proving

Theorem proving systems are diverse in the logics and proof calculi they implement and theorem proving problems come from a variety of domains and vary vastly in their problem structure, hardness and solution distribution. These factors influence the scope for applying the new programming paradigms and utilising the emerging architectures. For the effective application of concurrent technologies to tackle the

challenges of size and complexity faced by theorem provers, a one-solution-fits-all approach is unlikely to work and an iterative developmental life cycle of implementation and experimentation involving empirical studies and analysis is required. However, this experimentation phase can often be stifled by the difficulties of implementation as concurrent programming is notoriously error prone and difficult to program. Thus, it will be hugely beneficial to adopt an implementation methodology that allows for rapid prototyping of and experimentation with, application of concurrent techniques to theorem provers.

Over the years, parallelisation has been explored among many of the theorem proving flavours. We discuss some of these in chapter 2. Most of these systems have relied on complicated OS level thread and socket programming for implementing the concurrency features. From a software engineering perspective, the concurrency features used are very much tied to their individual application and this does not encourage incremental development or code-reuse. A recent review paper on trends in parallel computing and multi-core technologies [Asanovic et al., 2006] emphasises the need for effective software implementations that will enable optimal utilisation of the available processing power in emerging architectures. To the same end, the authors also argue for the need for powerful

> "distributed programming abstractions that can capture the common requirements of classes of applications which are related but have quite different computational methods at a lower level of granularity."

In this thesis, we discuss an implementation methodology that addresses these issues of ease of prototyping and experimentation, facilitation of incremental development and code reuse. Our approach uses distributed programming abstractions to encapsulate the concurrent techniques applied to address theorem proving tasks and a concurrent functional programming language, Alice ML [Rossberg et al., 2006], for implementation. The abstractions, in turn, are implemented as higher-order functions in Alice ML. Using this methodology, we have developed novel proof search approaches using concurrent techniques.

1.3 Case studies

The discussion of the developmental approach is aided by our experience of development and experiments with two diverse case studies of theorem proving: the proposi-

tional satisfiability problem (SAT) and LCF style (first-order) theorem proving. These are representative of two vastly different styles of theorem proving, the former being brute-force, machine-oriented search while the latter is closer to human reasoning and is thus a good vehicle for testing the utility of our developmental approach for a wide range of scenarios of the application of concurrent techniques.

As part of the development of these prototypes, we have developed concurrent programming abstractions. These abstractions can be used in a variety of theorem proving scenarios, examples of which we discuss later in the thesis.

1.4 Parallelisation options investigated in this work

In particular, in this thesis, we focus on applying the following parallel programming techniques to tackle the challenges of theorem proving:

Task parallelisation Use of multiple asynchronous computational processes operating simultaneously to achieve a task by effectively partitioning the work between them.

Dynamic exchange of information between concurrent processes Co-operative approaches to solving a task by harnessing the opportunities of (possibly partially evaluated) information exchange between processes working on the same problem or sub-problems

Use asynchronicity to synthesise novel computational patterns Some examples are: spawn multiple computations and return the fastest returning computation; *data-driven* execution, i.e. perform computations on the data as and when they are available.

Computational model There are various computational models for concurrent and distributed programming. In this thesis, we focus on the local-state, message-passing model and the use of higher-order programming abstractions for implementation of concurrent techniques.

1.5 Contributions

Implementation methodology

A prescriptive discussion of desirable features of an implementation approach that will allow for rapid prototyping of and experimentation with, novel approaches to theorem proving that can be achieved by applying concurrent programming techniques and aid portability and incremental development. This thesis focuses on one such approach:

- Use a functional programming language with language-based support for concurrency (as opposed to API based) to implement the concurrent techniques. We have used Alice ML as the implementation language.
- Use programming abstractions to encapsulate the concurrent techniques. In particular, we have developed the programming abstractions as higher-order functions in Alice ML.

The utility of the approach in terms of the ease of prototyping and experimentation and the portability and incremental development criteria are demonstrated via two case studies representing diverse styles of theorem proving: SAT and LCF style first-order proving. The case studies serve an orthogonal purpose of investigating previously unexplored opportunities of applying concurrent techniques to the respective systems and the key contributions arising from these are described below.

SAT

DPLL-Stalmarck a hybrid approach for SAT has been developed using two different, but complementary SAT algorithms: DPLL and Stalmarck.
A prototype that implements this approach has been developed in Alice ML and uses solvers based on the two algorithms in an asynchronous setting and uses dynamic information-sharing to enable co-operation between the solvers. The DPLL solver is the main solver and the Stalmarck solver acts as a clause-learning process and supplies the learned clauses to the DPLL process, thereby helping the DPLL process to potentially prune its search space.

Concurrent Stalmarck As an exploratory research effort, a novel concurrent algorithm,*Concurrent Stalmarck* has been developed. This algorithm has

been developed by applying concurrent techniques to the original Stalmarck algorithm. It demonstrates an alternative to the task-partitioning techniques observed in existing parallel SAT solvers (which are largely DPLL-based) and is well suited for implementing on architectures with large scale parallel processing resources, e.g. large clusters.
A prototype implementing this new algorithm has been developed in Alice ML. The implementation demonstrates a novel form of implementing work distribution using the declarative concurrency features of Alice ML and uses minimal communication to achieve work distribution.

Concurrent programming abstractions for the following:

- Implementation of the *saturation* technique (used in the Stalmarck algorithm) in a concurrent setting.
- Encapsulation of the interaction of DPLL with the external solver, allowing for it to be extended to incorporate one or more external solvers as helpers.

LCF

- We have developed a multilayered approach to introduce sound extensions to an LCF prover by applying concurrent programming techniques to synthesize novel concurrent tacticals (control structures for applying tactics). The multilayered approach involves implementation of the concurrent techniques as abstractions which are in turn, used to implement the concurrent tacticals and which in turn, can be used interactively as well as within automatic proof search methods.
- We have developed a prototype in Alice ML, as a proof-of-concept for this multilayered approach. HAL, a prototypical LCF style first-order theorem prover [Paulson, 1996], was ported to Alice ML and the multilayered approach was applied to it to introduce a variety of novel concurrent tacticals. These concurrent tacticals are available for interactive and automatic use. They have been used within the automatic proof search procedures as well, resulting in some novel and interesting proof search methods.
- Asynchronous, collaborative implementation of the unification tactic within HAL has been developed. Multiple goals (sharing the same meta-variables

asynchronously) compute the unifiers and subsequently collaborate to come up with the unifier compatible with all the goals. This implementation is illustrative of a co-operative approach between multiple goals working asynchronously, sharing partially evaluated information.

1.6 Layout of the thesis

In this section, we signpost the material discussed in the thesis.

In chapter 2, we provide an overview of a selection of published research on parallelisation of theorem proving. Though by no means an exhaustive list, it spans a broad range of theorem proving flavours as well as parallelisation approaches. The field of parallel SAT solving has had a relatively high proportion of published research in comparison to parallelisation of other flavours of theorem proving and is covered in detail in §2.1. In §2.5, we provide a discussion of the state of the field of parallelisation of mechanised reasoning systems in the light of the works discussed in the chapter and identify the scope for further investigation of the topic, setting out an agenda for the work reported in this thesis.

In §3.1, we give a concise statement of the hypothesis of the work reported in this thesis, giving the rationale for our choice of the case studies used.

In chapter 4, we provide the relevant background related to theorem proving, in particular, SAT and LCF style theorem proving. Also included are explanations of relevant parallel programming terminology used in this thesis.

In chapter 5, we give a detailed discussion of why parallelisation of theorem proving should be considered. This discussion includes perspectives of hardware imperatives as well as theorem proving specific issues that motivate the need. This chapter also introduces the developmental methodology proposed in this thesis. This is done by providing a discussion of how parallelisation can be implemented, formulating desirable criteria for the same. In §5.4.1, the notion of concurrent programming abstractions is introduced and its applicability in the context of the work reported in this thesis is explained. Also included are explanations of a few relevant standard concurrent programming abstractions. In §5.5.1, the advantages of using a functional programming language for implementing concurrency are listed. The chapter ends with §5.6, explaining how Alice ML, the implementation language used for implementing the

prototypes discussed in this thesis, serves as a good vehicle to implement the proposed developmental methodology.

In chapter 6, we report our investigation of the focus of this thesis for the case study of the propositional satisfiability problem (SAT). In §6.1, we set out the agenda for the two novel approaches of application of concurrent techniques to SAT that we have developed. Details of the approaches and the implementation of their proof-of-concept prototypes are provided in §6.3 and §6.5. Details of the empirical evaluation carried out for these prototypes are provided in §6.7. Conclusions and pointers to future work are provided in §6.9.

In chapter 7, the investigation of LCF style first-order proving is reported. The multilayered approach developed for the same is explained with the aid of the proof-of-concept prototype of HAL, a LCF-style first-order theorem prover. The novel concurrent tacticals developed are explained with examples. Automatic proof search procedures implemented using these novel tacticals are described including proof attempts in HAL where they outperform their sequential counterparts.

In chapter 8, we provide a unified picture of the aims of the thesis and how they have been achieved in the light of the material described earlier in the thesis.

Chapter 2

Parallelisation of mechanised reasoning systems: An overview

Theorem proving, with its ever increasing suite of applications, is faced with the challenges of problem size and complexity. New avenues of exploration are crucially needed to tackle these challenges. One such direction is parallelisation. Recent years have seen a huge increase in the availability and accessibility of a variety of parallel processing architectures including multicore machines and a variety of distributed computing environments. Appropriate (re)engineering of applications is crucial to harness the power of these emerging architectures. There are a variety of possibilities for parallelisation of an application domain like theorem proving which has an established set of algorithms (a detailed discussion on parallelisation techniques is provided in chapter 5).

Parallelisation has been explored in the context of many theorem proving approaches using a variety of parallel and distributed architectures, including, but not limited to: propositional satisfiability (SAT) solvers (e.g. [Singer, 2006]), term rewriting based systems (e.g. [Yelick, 1992]), equational deduction based systems (e.g. [Denzinger et al., 1996]), model checking (e.g. [Heyman et al., 2002]), resolution based systems (e.g. [Bonacina, 1992]) and natural deduction based systems (e.g. [Benzmüller et al., 2008]).

The availability and accessibility of technological infrastructure often tends to define the directions and boundaries of research whose ultimate end products are system

implementations. The body of published research in the field of application of parallelisation techniques to theorem proving and automated reasoning too reflects this phenomenon: from the early work on parallel Prolog efforts designed for transputers (e.g. [Böhm and Speckenmeyer, 1996]) to SAT solvers designed for grids (e.g. [Chrabakh and Wolski, 2003]) to higher-order theorem provers for multicore machines (e.g. [Matthews and Wenzel, 2010]).

The parallelisation approaches used have been diverse. In this chapter, we provide a discussion of some of these approaches and related implementations [1]:

- Search space partitioning, information sharing (§2.1.7, §2.3, §2.2.1)
- Use of heterogeneous reasoning systems (§2.2.3)
- Portfolio-based approaches that use multiple solvers, matching the problem with the solvers (§2.1.10, §2.3.2)
- Approaches using concepts and notions of agent based systems (§2.2.3)
- Approaches targeted at specific architectures e.g. grids, employing techniques for effective load-balancing and utilisation of idle resources (§2.1.9)

In the work reported in this thesis, we focus on the following two flavours of theorem proving: SAT solvers and LCF style provers (see §4.4.6 for more details about LCF). In §2.1, we provide a survey of key published research on parallelisation for the SAT domain, highlighting the techniques that have formed the basis of the parallelisation efforts and the major issues faced in engineering efficient parallel SAT solvers. In §2.2, we provide a discussion of the parallelisation approaches adopted by interactive theorem provers. In §2.3, we provide a discussion of some work partitioning approaches used in parallel automatic theorem provers. An orthogonal dimension of relevance is the implementation platform and developmental methodology used to incorporate concurrent techniques into theorem provers. To this end, in §2.4, we consider the implementation viewpoint with a discussion of parallel functional languages as implementation languages. We end the chapter with a summary of the different flavours of parallelisation, with a discussion of the issues related to the implementation methodologies adopted by the various systems and our observations on what more can be done

[1] Given the growing list of published research in this field, the authors would like to emphasise that this list is by no means exhaustive.

to promote easy prototyping of and experimentation with the application of concurrent techniques to tackle theorem proving challenges.

2.1 Parallel SAT solving

Parallelisation has been investigated widely for the SAT domain in the past few years. In this section, we discuss some representative work related to parallelisation of SAT solvers, in relation to our SAT case study, discussed in chapter 6.

In §2.1.1, we provide an overview of some key techniques used in many state-of-the-art (sequential) DPLL-based SAT solvers, some of which are used in the parallelisation approaches reviewed in this chapter. In §2.1.2, we discuss some of the specific challenges posed by the SAT domain for effective parallelisation. In §2.1.5, we describe, in detail, two concepts that have been widely used in the DPLL-based parallel SAT solvers reviewed in this section: guiding path (GP), which has been used extensively for search space partitioning and conflict driven clause-learning (CDCL), which has been used in many systems that use collaborative learning. Among the non-DPLL solvers, in §2.1.4.2, we discuss the work on using the DPLL and Stalmarck's algorithm in a synergetic manner, as part of a heterogeneous proof engine. In §2.1.8, we review an early work based on a non-DPLL algorithm that uses collaborative learning. In §2.1.10, we review portfolio-based systems that use multiple solvers on the same problem.

2.1.1 Overview of techniques used in modern DPLL solvers

Many of the state-of-the-art, complete solvers of today continue to use variants of the DPLL algorithm, augmented with various techniques. Over the past decade or so, a huge amount of research has been invested in formulating a variety of techniques that have enabled modern DPLL-based SAT solvers to push their tractability threshold. These have included developing various heuristics, novel techniques to prune search spaces and effective data structures and implementations. We enumerate some of these below, with appropriate references for the interested reader.

Efficient data structures, efficient unit propagation, watched literals [Zhang and Stickel, 1994] introduced *tries*, an efficient data structure for the CNF based SAT prob-

lem and related algorithms enabling a very efficient form of unit propagation and was implemented successfully in the SATO system [Zhang, 1997]. Work reported in [Moskewicz et al., 2001] built on this further and introduced the notion of *watched literals*.

Better branching heuristics A wide variety of effective static and dynamic branching heuristics have been developed ranging from the maximal occurring variable to more sophisticated ones based on a function of the current variable and search-state. A detailed survey of branching heuristics can be found in [Hooker and Vinay, 1995].

Backjumping, conflict driven clause learning Non-chronological backtracking is a common technique used in most modern SAT solvers. It allows for jumping to a decision level, based on the reason for the conflict rather than merely tracing the way back up the search tree in a chronological order.

The size of the search tree is exponential for the DPLL algorithm. So, heuristics to prune the search space are crucial to make the approach to work in practice. Conflict driven clause learning (CDCL) [Marques-Silva et al., 1996], was introduced to address these and was implemented in the SAT solver, GRASP [Marques-Silva and Sakallah, 1996]. CDCL is discussed in detail in §2.1.6.

Randomised restarts It has been identified that even for some relatively easy instances certain orders of search may take the algorithm into parts of the search space that do not produce useful conflict clauses, leaving it floundering. Restarts were proposed in [Giles et al., 1998] as an approach to deal with high variance in running times over similar instances [Gomes et al., 2000]. A restart is the operation of throwing away the current partial assignment (excluding assignments at decision level zero), and starting the search process from scratch or with a (new) randomly chosen assignment. A restart is performed after a certain number of unsuccessful backtracks (in the execution of the DPLL algorithm). The clauses learnt are retained and the original problem is augmented with them for the restart.

Structural information, Formula preprocessing DPLL relies crucially on the CNF format and hence a given problem has to be converted to CNF, to be used with a DPLL-based SAT solver. This conversion often destroys the implicit structural

information that may be present in the problem instances. Hence, DPLL-based solvers fail to capitalise on the implicit structure, often observed in SAT instances derived from real world problems [Thiffault et al., 2004]. This issue has been addressed in different ways: by trying to identify and exploit the structural symmetry present in some problem instances; by introducing techniques to extract the structural information from the CNF problem or from the native problem format in a pre-processing stage and use it as auxiliary information for the DPLL to use in its branching heuristics [Sabharwal et al., 2003] and also to enhance the performance of clause-learning algorithms.

Runtime variations, Benchmarking, Phase transition The search spaces as spanned by the DPLL algorithm are highly irregular as it is hard to predict the effect of unit propagation. This irregularity is further accentuated in the SAT cases as the time taken to find the satisfying assignment can vary hugely for even different instances of the same class of problems. A rigorous analysis of runtime distributions of backtrack procedures for propositional satisfiability and constraint satisfaction has been carried out in [Gomes et al., 2000]. This shows the huge variation that is observed in the time taken to solve the same instance, by varying the order of branching (the branching is done using randomisation and the random seed is varied). Benchmarks have been developed for the SAT domain, e.g., SATLIB [Hoos and Stützle, 2000] provides a wide variety of CNF format benchmarks spanning random instances and real-world instances.

2.1.2 Search space partitioning, Dynamic load balancing

Functional partitioning and *data partitioning* (described in §4.8.3) are two common techniques adopted in parallel programming, to perform work decomposition. The former is not a viable option for parallelisation of the DPLL algorithm, because, the DPLL algorithm relies on the coherence of the state updates performed sequentially by the various functions. The latter, achieved in the case of the DPLL algorithm, by partitioning the search space using efficient techniques and heuristics is the approach adopted by many of the parallel SAT solvers based on the DPLL algorithm.

However, it is hard to predict the time needed to solve a given branch, as the effect of unit propagation in reducing a problem cannot be predicted always. This irreg-

ularity in the search spaces poses a significant challenge for performing static work decomposition (partitioning the search space) for effectively parallelising SAT solvers implemented using the DPLL algorithm. To address this, efficient dynamic workload balancing strategies have to be used, making it an important focus area especially for parallel SAT solvers which focus on optimally utilising bulk parallel processing resources by distributing work amongst them.

One approach to tackle such scenarios is to do some form of dynamic search space partitioning as evidenced by many parallel SAT solver implementations [Böhm and Speckenmeyer, 1996], [Zhang et al., 1996], [Sinz et al., 2001], [Blochinger et al., 2005a]. This introduces the need for effective dynamic load-balancing strategies for optimal utilisation of idle resources, without the load-balancing related communication causing too much of an overhead. Another approach is to use heuristics to pick a subset of variables and use assumptions based on them as units of work for parallelisation, e.g. as seen in [Gil et al., 2008].

2.1.3 Evaluation related challenges

Issues related to evaluation of parallel SAT solvers have been investigated in [Speckenmeyer et al., 1988], [Speckenmeyer et al., 1997]. One of the issues considered in this work is the anomalies in the super-linear speedups produced by some parallel implementations of backtracking search procedures. This is attributed to the non-deterministic treatment of the search tree by a parallel execution. The work also discusses the irregularity of the distribution of solutions for the SAT cases and the need to separate SAT and UNSAT cases for the purpose of evaluation of parallel SAT solvers based on DPLL. Irregular distribution of solutions, SAT vs UNSAT cases and architecture dependency make the task of comparison of sequential and parallel implementations of SAT very difficult.

2.1.4 DPLL-Stalmarck

In §2.1.4.1, we describe a well known drawback of the DPLL algorithm for SAT, its inability to use implicit structural information. In §2.1.4.2, we review a work that uses the DPLL algorithm along with other algorithms within a proof engine framework.

2.1.4.1 Using implicit structural information in the problem

Real world problems often possess a lot of implicit structure. However, much of this is lost in the process of encoding them as SAT problems, mostly by virtue of the CNF conversion process. However, this is unavoidable since the DPLL algorithm relies crucially on the CNF format. This drawback of the DPLL algorithm has received a lot of attention in the literature in recent years, e.g., see [Thiffault et al., 2004]. Different approaches have been proposed to address them: using non-clausal solvers [Thiffault et al., 2004], extracting structural information after the CNF conversion by exploiting variable dependency and/or symmetry, e.g., [Dubois and Dequen, 2001], [Beame et al., 2003], [Sabharwal et al., 2003].

Stalmarck's algorithm [Sheeran and Stalmarck, 1998], [Borälv, 1997] addresses the problem by avoiding the need for CNF conversion and by using relations between subformulas as the basis for the inference rules in the algorithm. It adopts a breadth-first search approach. The algorithm is described in detail in §4.5.3.

2.1.4.2 A compositional approach, using the DPLL and Stalmarck algorithms

DPLL adopts a depth-first approach and Stalmarck's algorithm adopts a breadth-first approach. The complementary nature of these approaches means that they explore different parts of the search space and thus there is potential to engineer a co-operative framework using the two approaches.

The work described in [Andersson et al., 2002] uses these two algorithms in a compositional manner, to solve SAT. It uses a proof engine framework approach to solve combinational design automation problems encoded as SAT problems. The approach is to engineer different proof techniques as *strategies*, i.e. functions between proof states and allow for composition of the strategies. Each strategy also takes an additional parameter which determines the time it is allowed to run. Both the DPLL and Stalmarck algorithms have been implemented as strategies in this system. The framework is essentially a sequential compositional system and hence the Stalmarck strategy has to be run for a pre-defined period of time and then composed with the DPLL strategy. Thus, it does not allow for dynamic interaction and cooperative information-sharing between the two techniques. In effect, it works as a pipeline of the different solvers used, each

solving a sub-problem independently.

The information produced by the Stalmarck process is independent of the DPLL's search-state. Thus, the Stalmarck process(s) can run autonomously and communicate their result dynamically to the DPLL solver. The results in turn, can potentially help to prune the DPLL's search space. So, there is clearly scope here for the two processes to be running concurrently. However, this is not the case in the work reviewed above [Andersson et al., 2002]. We have addressed these opportunities in our work on a hybrid SAT solver, based on the DPLL and Stalmarck algorithms, described later in the thesis, in §6.3.

2.1.5 Parallel SAT solver on transputers, PSATO, Guiding path

One of the early works on parallelising SAT using workload balancing is described in [Böhm and Speckenmeyer, 1996]. This work describes a parallel SAT solver deployed on a message-passing based parallel architecture, a **transputer** system (every processor is connected with at most 4 other processors) with upto 256 processors. Each processor runs a copy of a highly optimised sequential Davis-Putnam algorithm based SAT solver and solves small subformulas. A naive way of search tree decomposition is used as a starting point. It employs a dynamic workload balancing strategy based on a technique for estimating the workload for a sub-problem, based on a problem-class dependent constant and the number of unset variables in its partial truth assignment. The strategy is varied depending on the architectures and it involves the overhead of communication. The work focused primarily on UNSAT instances. It reports good performance with near linear speedup for the class of UNSAT formulas considered: random 3-CNF UNSAT instances.

Another pioneering parallel SAT solver implementations is **PSATO** [Zhang et al., 1996], a parallel SAT solver, based on SATO [Zhang, 1997], a highly efficient (sequential) implementation of the Davis-Putnam algorithm for SAT. A key contribution of this work is the introduction of the notion of a *guiding path* (GP), a technique useful for dynamically partitioning the search space into *non-overlapping* portions. GPs have since been used as a key technique for work distribution in many of the parallel SAT solvers, [Sinz et al., 2001], [Blochinger et al., 2005a], [Feldman et al., 2005].

2.1.5.1 Guiding path

A Guiding path (GP) is the path in the search tree from the root to the current node, with additional information attached to the edges as follows: Considering the binary search tree generated by the recursive calls to the DPLL algorithm, at a given choice point (i.e. a case-split on the truth values of the variable), the GP records the list of variables which have been assigned a value till that point. Each case-split corresponds to an entry in the GP along with the following information (i) Literal L_{d+1} , which was selected at level d (ii) A flag indicating if both branches have been explored (*closed*) or not (*open*), i.e. if backtracking is not needed or is needed respectively. An entry in the GP with an *open* flag is a potential candidate for search space division, as at some point, the algorithm will need to backtrack to that point and explore the subtree rooted at the other branch, say, T. E.g., if a process P_1 has a node N, given by $< L_1, open >$, then another process, say, P_2 can come along and take up the work of exploring the subtree T and the flag for node N, at process P_1 is updated to *closed*.

GPs also provide a way of recording work that has been done already. E.g., if the solver halts unexpectedly (e.g, by running out of memory, occurrence of some extraneous fault), with the following guiding path, $(< x_1, open >, < x_5, closed >, < x_3, closed >)$, then, when the solver is restarted with this guiding path as the input, the information in the GP can be used to avoid parts of the search space that have been explored already, such as $(< x_1, open >, < x_5, open >, ...)$.

It allows for dynamic work load balancing by providing a means to divide the search space on-the-fly. If a process, say *IP*, becomes idle , it can potentially ask another busy process, say *BP*, for a sub-problem from its search-space. *BP* can then pick a new variable from its GP to generate a new and unexplored (sub-) problem and give it to *IP* and the variable that was picked can now be closed, thus removing the sub-search space that has been given to IP from its own work.

To use GPs in these ways, a SAT solver should be able to start at any point within the search space enocoded in the given GP. This typically calls for modifications to the system.

2.1.5.2 Distinguishing features of PSATO

The main objective of this work was to utilise idle resources in a network of workstations, e.g., during out of work hours. In view of this, PSATO provides for start-suspend-resume facilities. This was realised by using GPs as a way of accumulating intermediate results of separate runs of the prover on the same problem. These facilities also allow for possibilities of a solver working on a particularly hard problem over many days or longer durations even, with possible interruptions.

PSATO adopts a *master-slave* model of distributed computation. A *slave* process is a Davis-Putnam algorithm based SAT solver that accepts as input a problem and a GP. For a given guiding path, the solver process picks a node from the GP to proceed, using a case-splitting rule as the guide to make the choice. The *master* takes care of task partitioning among slaves. The slave reports to the master upon task completion/interruption, with a result in the former case and a GP in the latter case. The master process maintains a list of GPs with the number of GPs being 10% higher than the number of slaves. If it falls below that, the GPs are split and work is distributed to the slaves. PSATO ran on a network of workstations and used a public domain distributed language, called **P4**, developed at the Argonne National Laboratory, [Butler and Lusk, 1994]. P4 provided a C library for programming a variety of parallel machines.

[Zhang et al., 1996] also discusses the inherent difficulties of evaluating the performance of a parallel SAT solver, because of the rapid fluctuations in the hardness of the problem. This work reports experiments on random 3-CNF UNSAT cases drawn from the quasi-group problem domain with a clause-variable ratio of 4.25 , which has been known to be the phase transition boundary [Gent and Walsh, 1994a] for SAT. The experiments were run for a number of variables = 100, 150, 200, with 50 cases for each and the average time was taken. The number of workstations used for the experiments were 1, 5, 20. The work reports better performance on speedup and overhead, for the harder cases, which is explained by the fact that the master got more chance to manage GPs and balance workloads, thus being able to score gains over the sequential version.

2.1.6 Conflict driven clause learning for DPLL

The size of the search tree is exponential for the DPLL algorithm. So, heuristics to prune the search space are crucial to make the approach to work in practice. Conflict driven clause learning (CDCL) [Marques-Silva et al., 1996], was introduced to address these. CDCL is a technique that grew out of AI research on explanation-based learning [Stallman and Sussman, 1977]. Whenever a conflict occurs in the DPLL algorithm and the algorithm is forced to backtrack, the system derives a reason for the conflict in the form of a new clause, by employing a powerful conflict analysis procedure which analyses the implication structure generated by the unit propagation procedure of the DPLL algorithm. The clause(s) thus derived, often referred to as the *learnt clauses* can be added to the problem, thus ensuring that the same assignment (that led to the conflict) is not made again.

CDCL was originally introduced to enable non-chronological backtracking. It has been further augmented with effective techniques for caching and reuse of learnt clauses, which can be added to the original set of clauses (i.e. the given problem). It is in this form that it has been widely employed in the context of parallelising DPLL-based SAT solvers. CDCL, along with other efficient implementation techniques, has boosted the tractability threshold of SAT solvers by a huge margin and is currently used as a standard technique in many of the state-of-the-art SAT solvers.

Clause length and potentially exponential number of learnt clauses (a learnt clause is generated for every conflict) are related issues of importance. Thus, the topic of management of *learnt clauses* is an important focus area for effective use of CDCL, as adding all of them to the problem will quickly exhaust the memory.

Though CDCL was introduced in the context of sequential SAT solvers, the resulting possibilities of information sharing have been exploited by many recent parallel SAT solvers primarily as a tool to prune search spaces. This is discussed in §2.1.7.

In the parallel SAT solver scenario, management of clauses assumes high significance. E.g. for systems that rely on the Message Passing Interface (MPI), communicating vast amounts of data per worker over a whole range of workers can significantly increase the communication overhead and can slowdown the master process as well, thus

significantly affecting the overall performance of the system. Heuristics have to be employed to balance the length of the clauses and the number of clauses communicated and tradeoffs have to be made in communicating them.

Information sharing is especially useful if the shared information is consistent throughout the problem, and not for a particular context (e.g, for a particular case-split) and it is advantageous if the information-finding work can be autonomously organised without interfering with the main algorithm, as it helps to avoid bottlenecks. The potential of information sharing has been explored in a non-DPLL setting without using CDCL as well, as discussed in §2.1.8.

2.1.7 DPLL-based parallel SAT solvers using search space partitioning, dynamic workload balancing and CDCL

Use of search space partitioning invariably necessitates that some form of dynamic workload balancing strategy. Search space partitioning along with workload balancing were the prominent directions pursued in the early works on parallelisation of SAT. A large proportion of published research on parallel SAT which use the DPLL algorithm employ a GP related notion for search space partitioning. More recently, CDCL is being used with different forms of clause sharing, catering to different parallel computational models and architectures and a variety of heuristics have been developed to filter the clauses. In this section, we provide a discussion of some of these, mentioning their distinguishing features and performance.

PaSAT [Blochinger et al., 2005b] describes a parallel DPLL solver, using *GPs* based search-space partitioning and exchange of lemmas derived using CDCL. It is implemented on a proprietary distributed computing platform called DOTS (Distributed Object-oriented threads system). It uses C++ as the implementation language, message-passing for communication between the threads and works on distributed computing environments like clusters. It implements a form of distributed learning and restricts the length of the clauses that can be shared. Parameters are used for workload balancing by employing a work stealing strategy. However, a high level of communication is required to accomplish this form of load balancing. The clauses are exchanged between the individual sub-processes and thus the traffic can become prohibitively high.

It is a crucial consideration for any parallel system to keep the inter process communication low.

PaMiraXT [Schubert et al., 2005] uses MPI technology, CDCL and GPs to implement a form of distributed learning and is targeted at distributed computing environments. It uses MiraXT as the core solver. MiraXT is a thread based parallel SAT solver desgined for shared memory architectures (see §4.8.2 for defintion). PaMiraXT uses a shared clause-database which stores all the learnt clauses and the workers can choose the relevant clauses that they want to use from this database. This reduces the message latency and also eliminates the need to restrict the length of the conflict clauses generated.

PMSat [Gil et al., 2008] is a parallel implementation for SAT solving, based on the MiniSAT SAT solver [Eén and Sörensson, 2004], targeted at distributed computing environments like clusters. This has been implemented in C++ using MPI. MiniSAT is a DPLL-based SAT solver that incorporates many recent developments in heuristics and allows for satisfiability search based on a given set of assumptions (a set of literals set to *True*). This feature is crucial for the PMSat implementation. The parallelisation effort adopts a search space partitioning approach as follows: A subset of the set of variables of the given problem is chosen and *assumptions* are generated based on these variables. An *assumption* defines an implicit subspace of the problem's original search space. These assumptions form the *units of work* for the parallelisation effort. The solver is based on a master-slave architecture. The master explicitly distributes the work as described above to the workers. The workers are instances of the MiniSAT solver. The workers work on their individual subspace using the DPLL algorithm and report their result to the master. The workers are not given any time restrictions and are assumed to work in the absence of infrastructure fault. In the case of SAT, the satisfying assignment is communicated and the master stops with the answer. In the case of UNSAT, a clause is derived based on the conflict and the worker's assumptions using the notion of *GP* as described in [Zhang et al., 1996]. This is then communicated to the master along with the UNSAT status. The master maintains a database of learnt clauses received from different workers and uses it to prune search spaces of unexplored units of work (assumptions which are in turn, subspaces) or in some cases to eliminate complete units of work. Performance statistics of comparisons of sequential MiniSAT and PMSAT for 25 instances drawn from the SATLIB benchmarks show super linear speedups for some SAT instances. The authors also discuss the difficulties involved in making a conclusive empirical analysis of the gains of par-

allelisation based on these speedups. However, no explicit load balancing strategies have been implemented. Work allocation is explicitly done by the master when there is an idle worker. This requires the master to incorporate mechanisms for monitoring the workers. Communication of learnt clauses happens via the master and there is no peer-to-peer clause sharing.

2.1.8 PaModoc : a non-DPLL co-operative parallel SAT solver

[Okushi, 1999] describes a parallel propositional theorem prover called *Parallel Modoc*, based on the system *Modoc* [Gelder, 1999]. The spirit of this approach has been to use communication as a vital part of the algorithm and not just as a means of load-balancing. *Modoc* adopts a backward-chaining, goal-oriented, model-elimination approach to SAT. It uses the notion of *autarkies*, first introduced in [Monien and Speckenmeyer, 1985], which are partial truth-assignments with pruning information encoded in them and *Modoc* uses these to prune unfruitful branches. Furthermore, *Modoc* also records *lemmas* based on its sub-refutation attempts. *Parallel Modoc* executes multiple instances of *Modoc* as separate processes, one for each goal clause. The processes cooperate in finding a solution by sharing lemmas and *autarkies* via a shared data structure called the *blackboard*. The work reports speedup over the sequential version of *Modoc* on SAT encodings of planning problems.

There is an obvious limitation to this work, in that it targets a very specific implementation, i.e. Modoc, and thus cannot be used in conjunction with other DPLL-based systems and is unable to benefit from the huge advances made in the DPLL solver related techniques and heuristics. Nevertheless, it holds conceptual significance, as it adopts a different emphasis and direction compared to other trends in parallel SAT solving.

The availability of information that can be readily used, without any preconditions on their applicability, is very desirable for the purpose of effective and instantaneous information-sharing and to allow for autonomous agents to work on the same problem. However this is not the case in this work. Not all autarkies found can be immediately used by other processes, as they have associated preconditions that have to be met. Furthermore, there can be scenarios wherein there are conflicts between the autarkies themselves and a conflict-resolution policy needs to be in place. In the work on par-

allel Modoc, the policy adopted has been to give priority to the ones already on the blackboard over to those contributed by an individual Modoc process. This leads to both wasted effort as well as the additional time spent on working out what the relevant autarkies are. But, it is an indispensable step as given *Modoc*'s approach and the fact that *autarkies* are the shared information that is communicated via the blackboard, it becomes necessary to check the preconditions to produce consistent information to put on the blackboard. There is no published research available on further work on this system.

2.1.9 GRID based implementations

The **SDSAT** (Simple Distributed SAT) approach [Hyvärinen et al., 2008b], exploits the phenomenon of variation in the run times for the same instance (see §2.1.1) to run randomised SAT solvers in a grid-like distributed environment. **CL-SDSAT** (Clause Learning Simple Distributed SAT) is a parallel implementation specifically targeted to address the aspects of a grid-like computing environment. It uses a master-worker (a.k.a master-slave) architecture. The master process distributes the same problem instances to the workers each of which run instances of a randomised clause learning SAT solver based on the solver MiniSAT [Eén and Sörensson, 2004] (using randomised restarts and randomised branching decisions). The master stops when one of the workers finishes. However, unsuccessful workers (due to exhausting the allocated resources) transfer some or all of their learnt clauses to the master. These classes are added to the problem instance that is given to subsequent workers. Thus, this allows for a way of both accumulating and reusing the learnt clauses. But, the clause learning process itself is still based on CDCL and hence tied to the DPLL algorithm. Also, the learnt clauses cannot be communicated to workers that are already running. In particular, this work does not use any search space partitioning. The paper reports results of solving previously unsolved problems from the SAT 2007 competition, by using a version of CL-SDSAT deployed on a production level GRID environment.

zetaSAT [Blochinger et al., 2005a] is a solver using the same ideas as PaSAT, with some modification and re-engineered to address GRID specific issues. **GRIDSAT** [Chrabakh and Wolski, 2003], is a DPLL-based solver designed for the GRID, using the highly successful and optimised zchaff [Yogesh Mahajan, 2004] as the individual solver at each node of the GRID. Being a GRID application, the focus is on dynamic

resource allocation for optimal management of resources.

2.1.10 Others

ySAT [Feldman et al., 2005] is a parallel multithreaded DPLL-based SAT solver on a single multiprocessor workstation with a shared memory architecture. Though the core algorithm is DPLL, this system incorporates many of the optimisation techniques introduced in recent years. The emphasis has been on providing an efficient portable implementation using the computation model of shared memory architecture. It also demonstrates the disadvantages of parallel execution of a backtrack search procedure, like DPLL, on a shared memory architecture, e.g. a multiprocessor machine, due to issues related to increased cache-misses.

ManySAT [Hamadi and Sais, 2009] adopts a portfolio based approach aimed at shared memory architectures such as multicore architectures, and is targeted at addressing the sensitivity to parameter tuning exhibited by modern DPLL-based sequential SAT solvers. The implementation uses a portfolio of complementary sequential SAT solvers, obtained from careful variations of the DPLL algorithm. Restarts are used and are executed using heuristics based on the potential backjumping effect of learnt clauses.

[Cope et al., 2001] investigates parallelisation of SAT in a functional setting using the recursive version of the DPLL algorithm along with CDCL. It uses GpH (Glasgow Parallel Haskell, a parallel dialect of Haskell) as the implementation language and relies on asynchronous evaluation of both the branches at each case-split. It reports better performance for hard instances but no speedup for others, but the experimental results provided are fairly limited and there has been no subsequent published work on it.

NAGSAT [Forman and Segre, 2002] describes a SAT solver based on a more general technique called *nagging* (described in §2.3.2). In brief, the *nagging* technique allows for asynchronous solvers to work on reformulations of the same (sub-) problem. The NAGSAT system uses this technique with a DPLL-based solver, using the 3-SAT problem specification. The sub-problems that the worker gets is typically a sub-tree of the search tree of the master's current state. The worker applies one of the following reformulations to the sub-tree: (i) Reorder the list of variables that are awaiting assignment (ii) Randomly flip the logical meaning of the variables, thereby switching the

order in which the positive and negative literals will be split upon. The work reports sub-linear speedup for 64 nodes and compares it with the performance in the 2-node case speculating that the framework is quite scalable.

The 32/64 bit architectures of modern computers enable 32/64 1-bit operations to be performed simultaneously. [Heule and van Maaren, 2008] discusses work on using this feature to boost the performance of the DPLL algorithm by modifying assignments to variables in parallel. This is applied to an incomplete procedure on the lines of the one described in WalkSAT [Selman et al., 1996]. The payoff of modifying assignments in parallel is big here due to its high reliance on assignment modifications.

There has been work along lines of applying interdisciplinary approaches, e.g., of using market-inspired approaches to SAT. [Walsh et al., 2001, 2003] discuss approaches of formulating the SAT problem as production on a supply-chain and use the distributed market protocol for supply-chain management to solve the SAT problem.

2.1.11 Summary of key works on parallel SAT solving

In this section, we provide a summary of the work discussed above.

As we have seen in this section, in relation to complete methods for SAT solving, the vast majority of parallelisation efforts have been along the lines of either or both of the following

- Use (dynamic) search-space partitioning techniques primarily. Most of these systems employ load-balancing strategies using the *guiding path* technique [Zhang et al., 1996]
- Use DPLL with *conflict-driven clause learning(CDCL)* [Marques-Silva et al., 1996] in a distributed setting, often referred to as *distributed learning* in the literature.
- It is instructive to observe that all these parallel systems are based on the DPLL algorithm [2].

[2]There has been work on parallelising incomplete methods. Among complete methods, almost all the parallelisation efforts have focused on the DPLL algorithm. We do not address incomplete methods in this work.

- An orthogonal direction of research is the use of portfolio based approaches of using multiple SAT solvers and published research in this category report on systems where all the solvers in the portfolio use DPLL as the core algorithm. Other techniques include:
- Exploring alternative formulations of the problem asynchronously using a DPLL implementation
- Using a non-DPLL method with collaborative learning using the notion of *autarkies*.

Thus, though in comparison to parallelisation of other forms of theorem proving, there has been relatively large amount of published research in parallel SAT, there are still opportunities that merit serious investigation. Some of these are listed below and have been addressed in the SAT case study, discussed in this thesis, in chapter 6.

DPLL and need for other complementary players DPLL has been the dominant algorithm among complete algorithms for SAT and has been used in highly optimised implementations with sophisticated heuristics. However, as discussed in §2.1.4.1, DPLL suffers from a fundamental inability to leverage on implicit structural information present in real world problem instances [Thiffault et al., 2004]. This is due to its heavy reliance on the CNF encoding and the loss of structural information that happens as a result of the process of conversion to CNF. Recent works have tried to address this by supplying the structural information as an auxiliary input. However, this approach entails bespoke and often complicated domain specific analysis is required to enable mining of structural information for a given class of problems [Beame et al., 2003]. Despite these limitations, tremendous amount of research and development has been invested in the development of heuristics and efficient implementations of DPLL-based solvers. Thus, it makes sense to capitalise on the advanced technology available for DPLL-based solvers and use complementary solvers along with it. These complementary solvers should be chosen so as to address DPLL's shortcomings.

An additional desirable characteristic, particularly in the context of designing hybrid solvers will be solvers that enable exploration of the search space in a manner complementary to that of DPLL's search method. The depth-first and

breadth-first search are known to be complementary approaches. Thus, a hybrid co-operative system building on algorithms based on these two approaches holds a lot of potential. As discussed earlier, [Andersson et al., 2002] describes a sequential compositional system that uses the DPLL and Stalmarck solvers, along with other solvers in a proof engine framework. The sequential nature of the framework did not allow for asynchronous running of solvers, thus making dynamic information sharing infeasible.

Asynchronous running of the solvers and dynamic information sharing can be powerful tools in the context of creating a co-operative solver based on one or more algorithms, in view of both enabling effective forms of interaction and being able to use distributed computing architectures. Furthermore, information sharing is especially useful if the information-finding work can be autonomously organised without interfering with the main algorithm, thus avoiding bottlenecks. These opportunities have been addressed in our work on the hybrid SAT solver, engineered by combining the DPLL and Stalmarck algorithms, discussed in §6.3.

Information sharing and learning in non-DPLL solvers The clause-learning technique employed in the collaborative SAT solvers reviewed in this section is based on the conflict-driven clause learning technique of §2.1.6. Though CDCL has proved to be effective in boosting performance for sequential solvers, in the context of using it as an information provider for a concurrent co-operative architecture for SAT solving, its efficacy can be restricted for the following reasons. CDCL is embedded with the DPLL framework and this influences the clause-learning process itself, which can now learn only by spanning the search tree in the same way as the DPLL and does not bring any alternative viewpoints of the problem. Furthermore, the learning process also suffers from one of the main drawbacks of the DPLL algorithm, its inability to use implicit structural information §2.1.4.1. Added to this is the issue of the number of clauses generated by CDCL, as discussed in §2.1.6. To address this, it is useful to investigate alternative forms of learning clauses, independent of the DPLL algorithm and preferably in a way that can capitalise on the structure. Furthermore, it can be beneficial, if this learning is based on complementary approaches that can potentially span the search space in different ways. We have explored one such possi-

bility in our development of the hybrid solver, *DPLL-Stalmarck*, as discussed in §6.3.

Asynchronous solvers and dynamic interaction *Asynchronous* running of the participating solvers and enabling *dynamic* information sharing can be powerful tools in the context of creating a co-operative hybrid solver. These can enable effective forms of dynamic interaction, potentially pruning search spaces and also enable optimal use of distributed computing architectures. This is not feasible in a sequential, compositional approach,e.g., as the one discussed in the compositional approach described above, [Andersson et al., 2002]. Furthermore, information sharing is especially useful if the shared information is consistent throughout the problem and if the information-finding work can be autonomously organised without interfering with the main algorithm, thus avoiding bottlenecks. These aspects have been addressed in our work on the hybrid approach, described in §6.3.

Need for exploring work partitioning in non-DPLL solvers Effective work partitioning either in terms of task decomposition or data decomposition (see §4.8) is of crucial importance for effective parallelisation of an application. Thus, to enable effective parallelisation of SAT and utilisation of large scale parallelisation capabilities like those provided by clusters of workstations, developing effective work partitioning techniques for SAT is of tremendous importance. The vast majority of DPLL-based implementations use work decomposition by allocating subtrees to multiple parallel processes. The tasks of decomposition, allocation and management of subtrees and load balancing related communication, incur overheads. These overheads are offset, if the number of subtrees is significant and/or the average computational cost (time,space) of the subtrees is significantly high. It is well known that the search spaces of many of the SAT problem classes are irregular, thus making work decomposition very difficult. This in turn, proves as a serious limitation to parallelisation approaches using work decomposition based on subtrees. To address the difficulties posed by DPLL-based solvers for effective work decomposition, a useful line of investigation is the exploration of work-partitioning possibilities for algorithms other than DPLL. We have explored this for the Stalmarck algorithm, as discussed in §6.5.

Developmental/developmental aspects The features discussed above relate to the object-level aspects of parallelisation of SAT (as discussed in §5.3.1). Of par-

ticular importance to the objectives of this thesis are the developmental/developmental aspects. For the SAT domain and particularly for DPLL-based solvers, use of an implementation language like C has become the default choice. This choice has been motivated by the possibilities of employing techniques like effective cache optimisations etc. Almost all the works described in published research on parallel SAT solvers have been developed using C++ or C, a trend dictated by and shared with the state-of-the-art in sequential SAT solvers. These use APIs to manage the spawning of processes and inter-process communication. As discussed in §5.5.2, §5.5.1 and §5.4.1, these impose the following limitations: prohibitive developmental costs hampering the ease of prototyping and experimentation; less scope for portability and incremental development. F Our work has used Alice ML as the implementation language. This choice has enabled: easy prototyping, potential porting possibilities to a C-based implementation e.g. and development of distributed programming abstractions that can be used to address other theorem proving scenarios.

Of particular interest to the material discussed in this thesis are the developmental aspects of these systems. As can be gathered from the preceding descriptions of the various systems, almost all of them are based on fine tuned implementations of DPLL. In almost all cases, this entails use of a C like programming language and there are justified reasons for these choices, in terms of speed and machine-level fine tuning of the sequential implementations. However, for the purpose of parallelisation, these platforms may not always be conducive to easy prototyping and experimentation. Also, concurrent programming for imperative programming languages is known to be extremely difficult. Almost all the parallel SAT systems discussed here have used C or C++ and some form of MPI style communication. Also, there has been negligible contribution on portable techniques reported in any of the works, as speed and success rate have been the primary objectives for these systems. Consequentially, there has not been much of incremental development of the systems either. Given the vast body of work done on efficient implementations of SAT in C-like platforms, it is unlikely and perhaps not very efficient for the state-of-the-art to move to functional programming language platforms. However, a middle path can be the following:

- Have an experimental prototype system in a functional setting with support for concurrency and distribution (e.g. using a functional programming

language like Alice ML [Rossberg et al., 2006])

- Use this to prototype to apply concurrent and distributed techniques to address SAT
- Use programming abstractions to implement the concurrent techniques, focusing on portability and ease of implementing new techniques and prototyping new experiments
- Use the prototype to conduct experiments and perform empirical evaluation and to iteratively improve the concurrent approach employed
- Once a particular concurrent approach has been found to be effective, it can be implemented in other parallel SAT solvers with the aid of the abstraction used for the implementation. In particular, these target parallel SAT solvers can be ones that use a C-like platform with parallelisation support.

We have adopted this implementation methodology in the prototypes reported in this work.

2.2 Interactive theorem provers

In this section, we discuss some of the works that address parallelisation in the context of interactive theorem provers.

2.2.1 MetaPRL

[Hickey, 1999] discusses a prototype distributed proving architecture implemented within the MetaPRL logical framework, a system derived from the Nuprl proof development system. It aims to provide a distribution mechanism for general purpose tactics, thus making it theoretically feasible for it to be applied to any definable logic. The focus is on fault tolerance: for cases where large proofs are run on a cluster and proofs should not be lost due to machine failure or network failure. The distributed tactic module replaces the sequential tactic module, which is an intermediate layer between the tactic library and the logic engine, in the context of the MetaPRL logical framework.

The prototype was implemented using *Ensemble* [3], a generic communications toolkit developed using *OCaml*, an ML dialect. It treats two different parallellisation options at the level of sub-goal generation: (i) and-parallelism, the case when all sub-goals have to be proved for the goal to be proved (ii) or-parallelism, the case when proving any one of the sub-goals is sufficient. The parallelism opportunities considered are those provided by the compositional and choice related tacticals (control structures for applying the individual tactics).

A *scheduler* is used and a client submits a *job* to it. A *job* consists of a goal and a tactic that needs to be applied to it. The scheduler maintains a constant number of threads in its thread pool and allocates the jobs to the individual threads from the thread pool. Ensemble provides an implementation of the *global shared memory* abstraction to maintain a queue of pending jobs and to provide a space for the individual threads to post their progress to; the scheduler adds jobs to this queue. Every process in MetaPRL holds a copy of this shared memory and locks are used to manage the read-write conflicts.

The scheduler can perform the following communication operations with the threads: issue a new job, cancel a running thread, ask a thread for an unfinished job, receive a result from a thread. It does similar communication operations with the client: receives a job, sends the results back, accepts a job cancelation request from the client.

The scheduler maintains a pending-job pool and a running-job pool. When a new job is submitted by a client, the scheduler places the job in the pending-job pool, and enters the scheduler loop in which it allocates jobs from the pending-job pool to idle threads and updates the running-job pool. If the pending-job pool is free, it requests all threads to return part of their proof trees to the scheduler. This can be considered as a form of *work stealing*. When a thread completes, the result is used to prune the proof tree of which its goal was a node. The pruning of the proof tree is done depending on: (i) the success and failure of the job, i.e. the application of the given tactic to the given goal and (ii) if it was or-parallelism or and-parallelism.

The difficulties faced in implementation are discussed. The *Ensemble* toolkit was not designed to support multiple threads. This necessitated communication between the

[3] http://dsl.cs.technion.ac.il/projects/Ensemble/

MetaPRL processes and the Ensemble processes to be routed through a (physical) shared memory. The limited serialisation capabilities of OCaml were used as communication mechanisms for communicating tactics as functions. This required careful engineering of mechanisms to make the right choices of variables (e.g., bound variables should not be sent as part of the message).

Results comparing the unthreaded sequential prover and the distributed architecture are discussed for: (i) fully automatic proofs for the pigeon hole problem and a first-order logic formulation of proof of ancestry in a large genealogical database. (ii) automated replays of proof transcripts for interactively generated proofs for domains related to the Nuprl type theory. The work reports good speedups for an ensemble group of 5 processors.

The genealogical case showed super linear speedup and the work cites attributes this to the fact that the random scheduling algorithm performed better than the default depth-first search performed by the unthreaded prover. However, the number of cases tried are fairly small as are the problem sizes: the results presented are only for individual instances from each problem, for instance for the pigeon hole problem with the number of holes as 3 and 4. The problems considered are fairly small and it is hard to get a clear picture of the efficacy of the architecture, because, as the problem size grows, the communication overheads and workload increase.

2.2.2 Parallel theorem proving in Isabelle using PolyML

The work discussed in [Wenzel, 2009], [Matthews and Wenzel, 2010] aims to provide parallelisation support for Isabelle via the PolyML (an ML dialect) platform. It addresses the *multicore* architecture specifically. It reports the details on the significant reworking of the ML layers undertaken to facilitate support for parallelism in the PolyML platform. It lays out a few possible scenarios where the PolyML's parallel features can be used for the Isabelle/Isar system. The work reports experiments on parallel theory loading. However, no concrete case studies or examples of parallel proof checking are provided.

- The authors have focused on facilitating PolyML to support multicores and runtime systems that support (to use the authors' words) *truly parallel system threads*. They further state that Alice ML's runtime system does not support

this feature. However, Alice ML provides good support for high level language constructs that facilitate rapid prototyping, experimentation, modularity, incremental development and allows for programming abstractions to be synthesised as higher-order functions and can thus be an ideal choice to base a prototypical experimental workbench on.

- The focus of this work has been to enable *implicit* parallelism leveraging on the Isabelle/Isar document structure rather than enabling *explicit* parallelism and/or giving the user the choice and flexibility to develop their own parallel implementations. Large Isabelle-Isar proof documents possess some structure in their collection of theories (a directed acyclic graph (DAG) to be specific). Thus, there is scope for independent nodes in that graph to be loaded in parallel [Wenzel, 2009].

- The work reports significant reworking of the PolyML internals. Though not covered in detail in the papers, the work has entailed significant reworking of Isabelle's stateful bootstrapping process which relies on the notions of *heap* (a dump of the bindings at the ML top level environment) and usage of a non-standard ML feature *use*. As discussed in §7.4, our efforts to port Isabelle to Alice ML helped to highlight some of these issues. The Isabelle reorganisation entailed is echoed in the conclusion of the work reported in [Wenzel, 2009] as follows:

  > "impure programming might well be considered as premature optimisation from the past that is better avoided in highly parallel programs - if correctness and performance matter. The sources for Isabelle/ML was already *almost* purely functional. We merely had to throw out a small amount of stateful code that had crept in over the years."

- Given the diversity in structure and solution space of theorem proving problems, the scope of applying concurrent techniques in a fruitful manner can vary vastly from one problem class to another. Thus, providing the user with the flexibility to develop their own extensions enabling them to develop novel proof search procedures tailored to address specific problem classes can be a very useful feature. However, this is not addressed in this work as the emphasis is on *parallel proof checking* rather than on *parallel proof search*.

2.2.3 OANTS

The OANTS project [Benzmüller and Sorge, 2000; Benzmüller et al., 2008] builds on the OMEGA system [Melis and Siekmann, 1999]. It aims to provide a flexible framework for integrating (specialist) external reasoners in a central theorem proving environment. Proof rules, tactics, methods and external systems are encapsulated as single reasoning agents. The central proof object plays a pivotal role in the system, for the purpose of exchanging results with the external reasoners. The heterogeneous setup allows for multiple proof attempts to be executed in parallel, by possibly different reasoners. Furthermore, the design allows for parallelisation opportunities potentially on different levels: term level and proof search level. The term level possibilities are explored in the implementation of the *command suggestion* mechanism in OANTS, which is discussed below. The use of external reasoning systems can be interpreted as parallelisation at the proof search level. The system has been developed in Allegro Common Lisp and uses its parallelism support. In the following sections, we summarise the distinguishing strands of investigation explored in OANTS, which utilise asynchronous modes of execution.

- The integration of reasoning systems aspect is portable to other systems. But, the command suggestion mechanism is highly dependent on the proof object/proof data structure of the OMEGA system and thus does not allow for easy portability
- Use of agent based mechanisms for the interaction and orchestration of heterogeneous reasoning systems
- The central proof object allows for translations of contributions from external reasoners into it.

2.2.3.1 Flexible integration of heterogeneous reasoning systems

The heterogeneous reasoning systems addressed include: Higher-order and first-order, model generators and computer algebra systems. It uses the MathWeb software bus [Zimmer and Dennis, 2002] primarily for distribution and communication. Some key distinguishing aspects of this strand of the project are:

- One of the key features is the use of a *central proof object*. This is used for exchanging information from and with the main proof and the external reasoners.

- The system uses a declarative framework aimed to allow for integration of reasoners in a customisable and resource adaptive manner.
- The co-operation between two integrated systems has been realised via an inference rule.
- A concurrent hierarchical blackboard architecture is used for orchestrating co-operation between the various agents. Problems and sub-problems are posted to the blackboard from where they can be picked up by an external reasoner which can then contribute to the overall solution either by solving the problem that it has picked up or generating sub-problems for the same.
- The idea of using a prover and counter-example generator has been explored, for instance using the automatic first-order prover, *Otter* [McCune, 1994].
- Experiments have been reported converting first-order problems into higher-order and using a higher-order and first-order prover collaboratively outperforming first-order provers for some instances.
- Allows for suspend-resume functionality on a higher-level for resource optimisation: E.g., when a proof state has only first-order goals, the agents for higher-order rules are switched off.

2.2.3.2 Command suggestion mechanism within the OMEGA system

The command suggestion mechanism within the OMEGA system, has been realised by employing an agent-oriented approach incorporating concurrent consideration of the various possible next steps followed by a weighted analysis of the same using goal-directed heuristics. Originally developed to support the user in interactive theorem proving by searching for possible next proof steps during user interaction, these suggestions are computed by inference parameters extracted from the proof state which inform the search for applicable inference rules.

- The individual agents are of two types: *command* agents and *suggestion* agents. The *command* agents post arguments to the argument-blackboard, triggering *suggestion* agents to post possible suggestions to the suggestion-blackboard. Agents suggest arguments of inference rules and they are assessed by an independent agent on the basis of heuristics. The suggestion agents autonomously

search for suggestions, as background processes. The suggestion agents cooperate by exchanging results via a blackboard architecture. The ranked results are shown to the user as they are computed, thus preventing the potential bottleneck of long user waiting times. The background processes allow for utilisation of idle resources and application of resource-adaptive strategies.

- The approach has aimed to capitalise on the implicit information (typically relating the arguments of the rules: premises, conclusions and additional parameters) present in rules and tactics in a natural deduction setting. Each inference rule has its own associated agent society and its own associated blackboard. Each external reasoner is also encapsulated by an agent.

- In the context of OANTS and the heterogeneous setup, the suggestions can include calling external reasoners apart from the routine ones: application of tactics, specific calculus rules and proof methods. To be precise, using the information from the proof state, the applicability of the rules, tactics etc are tested and the appropriate parameter instantiations are suggested for the same.

- The system can work in two modes: presenting the suggestions to the user, leaving the ultimate decision to the user or in an automatic mode, where the system makes the choice and stores the others for possible backtracking.

2.2.3.3 Exploration of multiple strategies

OANTS has also been used within the multi strategy proof planner MULTI [Melis and Meier, 2000] in the following ways:

- To determine the applicability of proof planning methods in the context of interactive proof planning.

- To check for applicable theorems from the mathematical knowledge base.

2.3 Work partitioning approaches used in fully automatic theorem provers

2.3.1 TEAMWORK

[Denzinger and Kronenburg, 1996] proposes the *teamwork* approach, to tackle the difficult problem of work partitioning for automatic theorem proving. It is advocated by the authors as a useful technique for scenarios where the description of the task shows no obvious ways of distribution. It is inspired by the team dynamics of a modern organisation, in particular, where the teams can be reconfigured. It uses techniques from the AI *planning* domain. Three categories of computational components are introduced: *expert, referee* and *supervisor*. It is a hierarchical structure where each expert reports to a referee and the referee reports to the supervisor who is responsible for steering the subsequent processing. An iterative process is specified as follows:

- Experts are allocated individual tasks during the *work phase* and report to its referee upon completing the task
- The referee produces a *report* for each of its experts and chooses which of the results may be of interest to other experts and reports them to the *supervisor*
- In the next phase (referred as *team meeting*), the *supervisor* aggregates all the information that it has and evaluates the performance of the individual experts (referred as *short term memory*) and also augments its knowledge about the referees, in terms of their dependencies and incompatibilities (referred as *long term memory*). Based on the knowledge that it has, the supervisor performs the *reactive planning* task of choosing the experts for the next round and their resource allocations. It also chooses the results that will benefit the majority of the experts and adds it to the problem instance for the next round.

Mechanisms are proposed for accomplishing the steps involved in this iterative process: judgements made by the referees; information used by the supervisor to make the decisions of devising a new plan, revising a plan and allocation of resources.

The teamwork approach is fundamentally a competitive approach. By using a reactive planning architecture to devise and revise a plan, it addresses the problem of not being able to effectively partition the work apriori for a given problem. By using multiple

experts working on the same problem, it allows for the same problem to be tackled using different approaches. However, there is a level of redundancy, as each expert holds a copy of the problem. Also, there is no explicit knowledge-sharing or co-operation between the processes. The authors claim to facilitate implicit co-operation as after each round, only the results that will benefit the majority of the experts go to the next round. The work reports results where the whole system fares better than the individual experts.

The *teamwork* approach was developed initially for the domain of equational deduction by completion [Denzinger and Kronenburg, 1996]. It has subsequently been used to parallelise strategies based on the unfailing completion procedure using a combination of the teamwork approach and the PaReDuX system [Avenhaus et al., 2002], a strategy-compliant parallel implementation of the unfailing completion method. The approach has also been used in the **TECHS** system [Fuchs and Denzinger, 1997], where it is used to engineer a heterogeneous reasoning system.

2.3.2 Nagging: NAGSAT, DALI

[Segre et al., 2002; Sturgill and Segre, 1997] propose a generic parallel search-pruning technique called *nagging*, in which asynchronous solvers work on different reformulations of the same problem or sub-problems. This is aimed at exploiting a given solver's sensitivity to the problem's formulation. E.g., an alternative reformulation of the N-queens problem can be a 90-degree board rotation. The technique adopts a master-worker (a.k.a master-slave) approach. The master carries out a sequential search. A problem transformation function is specified for each worker to map search trees to alternate search trees. The workers work on the alternative formulation (using its problem transformation function) of a sub-space of the search space that the master is working on.

The possible scenarios of interaction between the master and worker are as follows: (i) If the master backtracks beyond the sub-tree given to the worker (thus rendering the worker's work redundant), then, it issues a call to the worker to quit and the worker becomes idle and goes into the loop to request for more work from the master. (ii) If the worker completes before the master, then it communicates its results to the master and depending on the result, the master uses it either to complete the problem or to prune its own search space and continue working. *Nagging* shows real benefits when

(ii) happens often and (i) happens rarely.

The technique is designed to be inherently fault-tolerant and scalable. It does not require explicit load balancing as whenever a worker becomes idle, it goes and fetches work. A serious limitation of this work is that there is no information sharing between the workers. Furthermore, it crucially hinges on the availability of effective problem reformulation techniques, which in itself requires highly tuned heuristics. The technique has been implemented for a SAT solver [Forman and Segre, 2002] and for a resolution style first-order prover [Sturgill and Segre, 1997].

2.3.3 Other systems

The **DARES** (Distributed automated reasoning system) [Intosh et al., 1991] applies ideas from the distributed problem solving domain to theorem proving. It is based on resolution style automated theorem proving. The objective of this work is to come up with a co-operative problem solving strategy that works by using independent agents working on a problem with the caveat that no agent has sufficient knowledge to solve the problem. The solution proposed aims to deliver a co-operative strategy where each agent works on its own incomplete knowledge and uses heuristics for co-operation. The co-operation is in the form of requesting other agents for information, the decision to make the request being determined by its own assessment of its current state.

[Fisher, 1997; Fisher and Ghidini, 2002] discuss early ideas on a computation model for concurrent theorem proving using asynchronous, autonomously executing objects (referred as *agents* in this work). It is based on the notions of broadcast message passing and grouping the agents to minimise communication and structure the agent space. In the context of theorem proving, formulae are distributed to the agents and an appropriate logical deduction mechanism is encapsulated within the execution machinery of the agent. The agents use *broadcast* message passing for communication and each agent *listens* to the messages being broadcast and takes appropriate action. However, the focus of the work is to apply the ideas and the related computational model to complex distributed systems rather than utilise concurrent programming techniques to engineer better theorem provers. Moreover, the work does not include details on system implementations and/or empirical results for a concurrent theorem prover. No further work has been done applying these ideas to engineer better theorem provers[4],

[4]Personal email communication with the author

though there is published research available on application of the ideas to multiagent systems [Fisher, 2004].

2.4 Parallel functional programming languages

It is well recognised that the functional programming languages are a good substrate for implementing concurrency. In recent years, many functional programming languages with concurrency support have emerged. In this section, we provide a summary of some of the advantages of using a functional programming language to implement a concurrent system and enumerate some key concurrent functional programming languages. More details are provided later in the thesis in §5.5.1.

Some of the key advantages of functional programming languages are [5]:

- Immutable state
- Lack of side effects
- Referential transparency
- Allows for composition
- Ease of synchronisation, one of the biggest challenges faced by a programmer using concurrent techniques. Many imperative languages use *explicit synchronisation*, i.e. the mechanisms of synchronisation have to be completely handled by the programmer and require careful use of locks, semaphores etc. One of the established techniques that circumvents the need to use these devices is that of *implicit data flow synchronisation* (explained in detail in §5.5.2.1). This technique fits naturally into the declarative concurrency paradigm and hence a functional programming language is well placed to support this.
- A functional programming language equipped with concurrency support provides the perfect setting for development of *concurrent programming abstractions* as higher-order programming constructs that can be composed and reused.

Some functional languages that provide concurrency support are:

[5] Some of these apply only for *pure* functional programming languages

Erlang Erlang [Armstrong, 1997, 2007] has been used in real-time telecommunications applications at the Ericsson laboratories, Sweden. Its computational model treats processes as black boxes with message-passing as the sole form of communication. The emphasis is on robustness and fault-tolerance, driven by the target domain of real-time applications. However, it does not have support for type inference.

Haskell Haskell is a pure functional programming language and various libraries have been developed to provide support for parallel programming [Jones and Singh, 2008]

Scala Integrates features of object-oriented languages and functional programming languages and uses static typing [Odersky, 2004]

F# F# [Syme et al., 2007] provides language-integrated support for asynchronous functional programming with a focus on reactive event-driven programming

OCaml *OCamlMPI* [Leroy, 2003], is an implementation of bindings for OCaml (a functional programming language [Leroy, 1996]), based on the message-passing interface standard (MPI). MPI bindings allow for restricted forms of programming models. In particular, the multithreaded model is not possible with MPI bindings

Alice ML Alice ML [Rossberg et al., 2006] is a standard ML based language with support for concurrency and distribution. It provides static typing while allowing for dynamic type checking of higher-order modules loaded at runtime. This is the implementation language used in this work and is described in detail in §5.6 and Appendix §A 2

PolyML Provides support via libraries for a small selection of asynchronous programming features like *futures*. The focus is to use multicore machines using native threads [Matthews, 2010]. It does not provide support for distribution.

PolyML vs Alice ML In PolyML, support for concurrent programming is not very developer friendly (compared to e.g. Alice ML). It is still fairly primitive and has only a very limited set of features. This can prove to be a serious limitation even to be able to develop modest experiments to use these features for proof checking. The current support provides an ML view on the original C versions of the well known *Posix Threads (or pthreads)* library using the following features:

Encapsulation of a concurrent computation The *fork* operator creates a new thread and executes the given computation but cannot give a return value. Also, there is no *join* operation. The authors state that to simulate a return value, side-effects will need to be used together with appropriate synchronisation. Alice ML provides concurrent computations (encapsulated by the *thread* structure) as first class values. This feature together with the powerful support for *implicit (dataflow) synchronisation* (for more details, the reader is referred to §5.6, §5.5.2.1), allows for asynchronous computations to be passed around as *futures*, which stand for the pending computations.

Dataflow synchronisation Unlike Alice ML, there is no support for *implicit (dataflow) synchronisation* (see §5.6, for more on the support provided by Alice ML). The work has made an attempt to wrap up the *pthreads* based synchronisation primitives (mutex, condition variable). The authors state that this is a higher-order representation of *conditional critical section*. However, from the details described in the paper, the operations provided are fairly restrictive and it requires the programmer to handle many of the synchronisation relation operations: e.g., consider the key synchronisation primitive called *guarded_access*; this has to be supplied with an explicit guarding predicate and a state update function; a change in state is *broadcast* to the waiting threads; though the broadcast operation is done automatically, the waiting threads have to take the responsibility for establishing some semantic conditions for sychronisation; the primitive cannot make distinctions between state changes while signalling; furthermore, the *broadcast* operation is a source of bottleneck, when the number of dependent processes are large.

2.5 Conclusions

In this chapter, we discussed some key directions in which research has been pursued to address parallelisation of theorem proving, focussing on some prominent representative systems, most relevant to the work discussed in this thesis. The discussion highlights the diversity of the theorem proving flavours tackled and the parallelisation techniques employed. Among the systems discussed, most of them have attempted to use effective work partitioning and load balancing for optimal utilisation of re-

sources. Other technologies adopted are: agent based methodologies (§2.2.3); using asynchronous solvers (§2.3.2, §2.3.1); using techniques from other fields like those employed by the Teamwork project; using asynchronous proof attempts on multiple reformulations of the problem. Some key issues that emerge as important for effective application of concurrent and distributed techniques for theorem proving are:

1. Search space partitioning
2. Dynamic load balancing
3. Effective information sharing
4. Identifying and addressing sources of bottlenecks
5. Overheads: Scheduling and locking/unlocking are known to be two main overheads affecting parallel implementations. In particular, when the individual subproblems created as a result of work partitioning are small, the cost of creating a thread and allocating a task to a thread can be many orders of magnitude higher than the work performed by the computation. In the case of shared memory, locking/unlocking account for a significant part of the overhead.
6. Scalability, i.e. the more processors there are, the faster the computation is performed (i.e. the faster the solution is found). Most of the parallel implementations imply a proportional increase in communication overheads with an increase in the number of processors. This becomes an inhibiting factor for scalability.
7. Evaluation difficulties: in particular, given the sensitivity of distributed systems to the effectiveness of a particular implementation, it becomes very hard to make a uniform empirical evaluation. Another related issue is that of evaluation of a particular implementation vs evaluation of the techniques employed.

The work reviewed in this chapter exhibit the diversity in focus areas of system development which in turn, influence the design decisions. One possible classification of the focus areas is as follows:

Architecture oriented Optimal utilisation of machine architectures and hence devising techniques to address their strengths and weaknesses, e.g., utilisation of idle resources in a distributed network of computers, like grids. For such a scenario, fault-tolerance capabilities and optimal work stealing techniques become very crucial for the success of the system. Related work discussed earlier are PSATO,

GRIDSAT, and metaPRL. A primary concern for these systems has been to address the scenario where workstations fail so as to enable productive use of work done till that point. Another objective has been that it should not cause bottlenecks and that it should not compromise the soundness and consistency of the system.

The field of SAT (which has seen a huge surge in published research on parallelisation efforts in recent years) provides a good illustration of the issue of architecture dependency and how the parallelisation efforts invariably are oriented towards making the most of and/or circumventing problems posed by the dominant architectures of the day. One of the earliest published work in parallel SAT was targeted at *transputers* [Böhm and Speckenmeyer, 1996]. A more recent work, separated by a decade from this is tailored towards the grid [Hyvärinen et al., 2008a] and thus focuses on the utilisation of idle-resources and adopts techniques for making judgements on the work required by using techniques from the research on distribution of solutions.

Application oriented The various systems have tried to achieve different objectives related to the particular flavour of system, using parallelisation and some of these are:

- SAT: To improve the tractability threshold which in turn, includes space and time. However, most works focus on improving the time taken to solve a problem.
- Portfolio based systems use characterisations of strengths of particular solvers with respect to problem classes. A distributed setup is used to run multiple solvers, matching the solvers with the problems.
- Heterogeneous systems aim to leverage on the strengths of different reasoning systems. The OANTS system (§2.2.3) discusses how the application domain of mathematical formalisations stands to gain from a heterogeneous approach employing distributed architectures. E.g., a proof attempt in a higher-order formalisation can generate problems that are very appropriate to be tackled by a first-order prover or a SAT solver.
- Exploit the effect of alternate formulations of the same problem by running asynchronous solvers on the different formulations. This has been investi-

gated in the context of SAT and automatic first-order proving in the work on the *nagging* technique e.g. as discussed in §2.3.2.

Theorem proving problems come from a variety of domains and they vary vastly in their problem structure, hardness and solution distribution. A one-solution-fits all approach is unlikely to work as each problem class and/or problems may stand to benefit by application of different concurrent techniques. Thus, the ease of prototyping and experimentation is of crucial importance for the effective investigation of the scope for applying concurrent techniques to theorem proving scenarios and to assess their efficacy. An iterative developmental life cycle is required addressing the following stages: implementation/prototyping, empirical studies, analysis and refinement of the system. However, the experimentation phase can often be stifled by the difficulties of concurrent programming which is notoriously error prone and difficult to program. Thus, it will be hugely beneficial to provide a prototypical system for the theorem prover under consideration such that it provides the building blocks and allows the user to build on them to quickly prototype new techniques, conduct experiments and carry out empirical analysis on the same.

Considering the various systems reviewed in this chapter, an almost uniform picture that emerges is the limited scope for portability and lack of incremental development. Given the changing nature of the architectures today, the issue of architecture dependency highlighted earlier is very relevant. This further accentuates the importance of producing portable implementations. On an implementation level, most of the systems reviewed in this chapter have used API based approaches which are not exactly conducive to portability. Also, there is little cross pollination of techniques employed, e.g. from one theorem proving flavour to another, or even within the same flavour, in many cases.

Another important issue is that of empirical evaluation. As discussed in [Bonacina, 1999], empirical evaluations conducted in the field of parallel theorem proving are not always indicative of the true potential of the implemented techniques. Because, as is imperative for empirical evaluations, they tend to be done for specific implementations rather than the strategies implemented. This speaks further for a flexible framework that allows for an effective *isolation* of *design* and *implementation*.

Use of the well established software engineering practice of effective programming abstractions of the concurrent techniques can help achieve this as well as aiding portability of the same. Use of *domain specific programming abstractions* for application of concurrent techniques has been advocated by leading experts in the field of concurrent programming as well [Asanovic et al., 2006] and has been adopted by many application domains. However, there has been no work towards producing concurrent programming abstractions that will be widely applicable to various theorem proving scenarios. This is in contrast to approaches adopted by other fields that have used concurrency and parallelism to build better applications. E.g., image processing [Falcou, 2009] uses the notion of algorithmic *skeletons* [Cole, 1991] to address this need. Further discussion on this topic can be found later in the thesis, in chapter 5.

Thus, the development of systems that allow for rapid prototyping of and experimentation with, novel proof search procedures merits serious investigation. The availability of the same can greatly help the development of effective application of concurrent and distributed techniques to theorem proving. In particular, it is worth exploring the use of programming abstractions for implementing the concurrent techniques as it can help effective isolation of *design* and *implementation* and promote: portability, incremental development and reuse of the abstractions across various theorem proving scenarios.

SAT solving and LCF style theorem proving are representative of two diverse schools of theorem proving. Among other things, SAT represents the style of brute-force search with little scope for human intervention and the LCF style is representative of interactive style of theorem proving and a style of reasoning closer to the way humans reason. Thus, these two are good candidates for testing the utility of an experimental workbench, to explore the scope and efficacy of using previously unexplored or little explored concurrent and distributed techniques to implement novel search procedures in the respective contexts of SAT and LCF style. An LCF style first-order prover can prove to be a good candidate to base a prototype on to apply concurrent techniques. In this chapter, we reviewed systems addressing parallelisation for these two flavours.

We summarised the state of parallel SAT and identified some possibilities for exploration of different directions in §2.1.11. Among the LCF style provers, the dominant research direction has been the use of heterogeneous provers, an example of which is the OANTS system discussed in §2.2.3. The metaPRL project adopts a different direction (discussed in §2.2.1) and addresses the topic of using idle workstations to

implicitly parallelise tactic application in a predefined way. However, it does not provide for information sharing and it does not provide the scope for the user to build their own concurrent techniques. LCF provers are ideal vehicles for developing *sound, programmable* extensions that incorporate concurrent and distributed techniques. They also provide scope for ML level user interaction. Thus, using a functional programming language with concurrency support to address this potential merits serious investigation. Giving the user the flexibility to develop their own extensions can greatly promote the possibilities of prototyping of and experimentation with novel proof search procedures that apply concurrent and distributed techniques.

In the next chapter, we present a concise statement of the hypothesis of this project and give an overview of how we have addressed the developmental aspects via the two case studies of SAT and LCF style prover and development of the respective prototypes. The SAT case study explores the opportunities identified in §2.1.11 and the LCF style prover enables *sound* and *programmable* extensions that in turn, can be used to develop novel proof search procedures.

Chapter 3

Hypotheses and case studies

3.1 Hypotheses

In this section, we state the hypotheses of this work. These are explained in detail in the next section, which includes the rationale for our choice of case studies.

Developmental level hypothesis

> Using a functional programming language with language-based (as opposed to API based) support for concurrency and distribution, enables easy prototyping of applications of concurrent and distributed techniques to theorem proving. Use of programming abstractions, to implement the concurrency techniques aids portability, promotes incremental development and allows for isolation of design and implementation.

The utility of the developmental approach described above, is illustrated via proof-of-concept prototypes of application of concurrent techniques to address two diverse case studies of theorem proving: the propositional satisfiability problem (SAT) and LCF style (first-order) theorem proving. Furthermore, the individual case studies, address the scope and utility of applying concurrent techniques in specific ways, by exploiting previously unexplored parallelisation opportunities within the case-studies, as described below.

Object level hypothesis

1. For the propositional satisfiability problem (SAT), use of an asynchronous mode of execution enables the development of two novel approaches to SAT:
   (a) A hybrid solver using an asynchronous combination of two distinct SAT approaches: the DPLL [Davis et al., 1962] and Stalmarck [Sheeran and Stalmarck, 2000] algorithms. In comparison to the stand alone DPLL solver, the hybrid solver performs better for some problem cases and does not show significant slowdown for other cases examined.
   (b) As an exploratory research effort, a novel algorithm has been developed by applying concurrent techniques to the Stalmarck algorithm. The new algorithm is well placed to utilise large scale parallel processing capabilities and demonstrates a novel form of work-partitioning approach for SAT.
2. A multilayered approach to application of concurrent techniques to an LCF style first-order prover, using concurrent LCF-style tacticals, realised via programming abstractions enables:
   (a) Programmable extensions (to the prover), incorporating concurrent programming techniques, retaining the soundness guarantees
   (b) Easy prototyping and evaluation of novel proof search techniques, applying concurrent programming techniques, that can be tailored to a given theorem proving scenario
   (c) The novel proof search procedures use concurrent approaches to deal with theorem proving tasks and in the process, address some of the shortcomings of their sequential counterparts and fare better in some test cases.

3.2 Our approach and choice of case studies

In the last section, we set out the hypotheses of this thesis. In this section, we elaborate on the same, with a brief outline of how they have been addressed, in this thesis. The rationale for choosing the case studies is also explained.

Developmental level A prescriptive analysis of the implementation aspects, as outlined in §3.1 above, is discussed, in detail, in chapter 5). In concrete terms, here is how we have realised the same, in this project:

Choice of platform Alice ML [Rossberg et al., 2006], a functional programming language with rich, lightweight, language-based (as opposed to API-based) support for concurrency and distribution has been used to implement the concurrent and distributed techniques. The rationale for this choice is covered in detail in §5.5 and §5.6.

Use of programming abstractions Programming abstractions have been developed, for the concurrent techniques implemented. The abstractions have been developed as higher-order functions in Alice ML, in a way that promotes reusability, portability and incremental development and allows for separation of design and implementation. This aspect is covered in detail in §5.4.1.

Object level The desirable developmental methodological criteria, gathered from our analysis, have been applied to implement prototypes of application of concurrent techniques to two diverse case-studies of theorem proving flavours: SAT and LCF style first-order theorem proving. The applications of concurrent techniques considered, aim to exploit previously unexplored parallelisation opportunities and are described respectively in chapter 6 and chapter 7. In brief, they are as follows:

SAT

- Implementation of a hybrid approach to SAT using asynchronous SAT solvers, based on combining the depth-first approach based DPLL algorithm [Davis et al., 1962] with the breadth-first approach based Stalmarck's algorithm [Sheeran and Stalmarck, 2000]
- Implementation of a novel distributed algorithm based on Stalmarck's algorithm for SAT [Sheeran and Stalmarck, 2000]
- Development of programming abstractions for the techniques employed
- Evaluation of the implementations using standard benchmark problems

LCF style prover

- Development of a multilayered approach for developing programmable extensions (to an LCF prover), such that the extensions incorporate

concurrent and distributed techniques and retain the soundness guarantees of the LCF prover. In particular, the multilayered approach we have developed is as follows: Use programming abstractions for the concurrent techniques and use them to develop concurrent tacticals and use them in turn, for developing novel proof search procedures

- Use a LCF style, first-order prover, to develop a proof-of-concept prototype, for this multilayered approach
- Evaluation of the implementation, assessing their scope of addressing the limitations of their sequential counterparts

The case-studies serve the following purposes:

- They help us to understand the efficacy of our approach to implementation, in terms of ease of prototyping and experimentation and portability.
- They help us to understand the performance gains/losses made by exploiting the particular parallelisation opportunities. Performance metrics include

  Speed, size of search space In comparison to their sequential counterparts in the average case scenario

  Scope, Success rates Can handle complexity and size better and/or can handle problems that are not tractable by the sequential counterparts

Chapter 4

Background

In this chapter, we provide details deemed relevant for the purpose of understanding work discussed in this thesis and an enumeration of notations and terminology used in this thesis. Definitions of a purely technical nature and/or definitions that are not explicitly used, but still relevant to the thesis, are provided in the glossary accompanying this thesis. In this thesis, we use Alice ML [Rossberg et al., 2006] syntax[1], to describe code fragments[2]. Topics addressed in this chapter include the following:

- Propositional logic, in §4.2.3 and first-order logic, in §4.3
- Propositional satisfiability (SAT) solvers, in §4.5, relevant for understanding the material discussed in chapter 6
- The prototype first-order theorem prover discussed in chapter 7 is an LCF style prover and uses sequent calculus. To this end, §4.4 provides a general overview of theorem proving, covering natural deduction, sequent calculus and the LCF style of theorem proving.
- §4.7 provides background material that is specific to first-order theorem proving and addresses unification, meta-variables and an enumeration of the sequent calculus rules for first-order logic.
- For a broader introduction to theorem proving and/or details on specific aspects, the following sources are recommended: [Huth and Ryan, 2004], [Harrison,

[1] which in turn, is based on standard ML (SML) [Milner et al., 1997]

[2] Many definitions covered in this chapter include recursively defined structures and functions and we use code-fragments as an aid to describe these, along with verbal descriptions.

2009], [Robinson and Voronkov, 2001]. Many of the definitions provided in this chapter are sourced from the first two texts.

- Relevant information on parallel, concurrent, distributed programming is provided in §4.8. Appendix §A 1 provides some more details on this topic as does chapter 5. [Andrews, 2000] is a recommended source for more on this topic.

4.1 Formal logic: basics

Logic is widely understood as the study of formal (symbolic) systems of reasoning and of methods of attaching meaning to them. In formal logic, a clear distinction is maintained between the formal (symbolic) expressions and what they stand for.

Syntax of a logic sets out a precisely defined language that provides the building blocks for the language (giving its alphabet and grammar) and the rules for a well-formed statement (often referred to in the literature as a *well-formed formula (wff)*). In this thesis, we use just *formula* to refer to a wff.

Semantics is concerned with the meaning of these formal (symbolic) expressions.

Interpretation maps expressions to their meanings, thus connecting the syntax and semantics of the given logic.

In the next two sections, we describe the syntax and semantics of propositional logic and first-order logic and associated terminology. Also included are descriptions of the notions of validity, tautology and satisfiability for the two logics.

4.2 Propositional logic

In propositional logic, formulas are intended to represent propositions, i.e. assertions that may be considered true or false (often referred to as *truth-values*). In the rest of this section, we describe the syntax and semantics of propositional logic and describe related definitions and terminology used in this thesis.

4.2.1 Syntax and semantics

Syntax Formulas in propositional logic are built using the following:

Constants 'True' ($\top$), 'False' ($\bot$) [3]

Atoms Atomic propositions, also referred to as *propositional variables* or just *variables*.

Logical connectives A logical connective is an operator that takes a fixed number (referred to as *arity*) of formulas as arguments and gives a compound formula as the result. Formulas in propositional logic are built using the following connectives, given below in the descending order of precedence, with examples illustrating their usage. For each connective, the symbols used to denote them are also given[4].

Negation, $\neg$, *Not* : $\neg p$, where p is a variable

Conjunction, $\wedge$, *And* : $p \wedge q$, where p, q are variables

Disjunction, $\vee$, *Or* : $p \vee q$, where p, q are variables

Implication, $\rightarrow$, *Imp* : $p \rightarrow q$, where p, q are variables

Double-implication, $\leftrightarrow$, *Iff* : $p \leftrightarrow q$, where p, q are variables

Propositional formula A propositional formula ϕ is defined over a set of propositional variables, $x_1, x_2, \ldots, x_k$, using the standard propositional connectives, $\neg$, $\vee$, $\wedge$, $\rightarrow$, $\leftrightarrow$. Listing 4.1 gives a datatype definition in Alice ML, for a *well-formed formula* (*wff*) in propositional logic. In Backus Naur Form, the definition of a propositional formula can be given as

$$\phi ::= \bot \mid \top \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid \phi \leftrightarrow \phi$$

where p stands for any propositional variable.

Notation In this thesis, we use lower case and upper case alphabets to denote variables and formulas respectively

[3] We use the capitalised words to refer to the constants

[4] For each connective, the abbreviated English word is used in code fragments and the symbol is used in infix formulas.

Listing 4.1: Datatype definition in Alice ML, for wff in propositional logic

```
datatype ('a) formula = False
                      | True
                      | Atom of 'a
                      | Not of ('a) formula
                      | And of ('a) formula * ('a) formula
                      | Or of ('a) formula * ('a) formula
                      | Imp of ('a) formula * ('a) formula
                      | Iff of ('a) formula * ('a) formula
```

Semantics The semantics of propositional logic is captured via the following definitions:

Valuation determines the assignment of truth-values to the atoms. It is a function from the set of atoms to the set of truth-values.

Truth-table, meaning of connectives The semantics of logical connectives can be explained using *truth-tables*[5]. *Truth-tables* (as used in propositional logic) are used to compute the truth-value of a given propositional formula, for each combination of truth-values taken by its constituent variables. Thus, if a given formula F has n propositional variables there will be 2^n rows (to account for the 2^n possible combinations of truth-values of the n variables) and $n+1$ columns (to account for the n variables and F) in the truth-table. An example is provided in Table 4.1.

p	q	p ∧ q
⊤	⊤	⊤
⊤	⊥	⊥
⊥	⊤	⊥
⊥	⊥	⊥

Table 4.1: Truth-table for conjunction of two variables

Truth-value of a formula Since propositional formulas are intended to represent assertions that may be true or false, the ultimate meaning of a formula is just one of the two *truth-values*, 'True' or 'False' and it depends on the truth-values assigned to the atomic propositions and the constants and connectives present in the formula.

[5]More on this line of explanation can be found in one of the references provided earlier

Given a formula F and a valuation v, the overall truth-value of the formula can be computed by the recursively defined function, *eval*, given in Listing 4.2. This function also clarifies the semantics of the logical connectives mentioned above.

Listing 4.2: Truth-value of a propositional logic formula, *F*, for a valuation, *v*

```
fun eval F v =
case F of
 False => false
| True => true
| Atom(x) => v(x)
| Not(p) => not(eval p v)
| And(p,q) => (eval p v) andalso (eval q v)
| Or(p,q) => (eval p v) orelse (eval q v)
| Imp(p,q) => not(eval p v) or (eval q v)
| Iff(p,q) => (eval p v) = (eval q v);
```

4.2.2 Validity, satisfiability and tautology

We say that a valuation v *satisfies* a formula F if

$$eval\ F\ v\ = \top$$

A formula is said to be:

- a *tautology* or logically *valid*, if it is satisfied by *all* valuations
- *satisfiable*, if it is satisfied by *some* valuation(s)
- *unsatisfiable* or a *contradiction*, if *no* valuation satisfies it.

Some related observations:

- A tautology is also satisfiable.
- A formula is unsatisfiable precisely if it not satisfiable
- For a given formula F, for any valuation, v,

$$eval\ (\neg F)\ v\ is\ false\ iff\ \ eval\ F\ v\ is\ true$$

  So, F is a tautology if and only if $\neg F$ is unsatisfiable

- Intuitively speaking,
  - tautologies are 'always true'
  - satisfiable formulas are 'sometimes (but possibly not always) true'
  - contradictions are 'always false'

4.2.3 More definitions and notations

Literal A literal is a variable, v or the negation of a variable. We use $\neg v$ to denote the negation of the variable v.

Clause A clause is a disjunction of literals. It can be written as

$$l_1 \vee l_2 \vee \ldots \vee l_n,$$

where each l_i is a literal. It follows trivially from the definition that for a clause to be true, at least one of the literals has to be true and it is false if all the literals are false.

Empty clause, unsatisfiability for a clause An empty clause, i.e. a clause with no literals is taken to be trivially unsatisfiable. A clause can thus be unsatisfiable either when it has no literals or when all the literals in the clause take the value *false*.

Unit clause A clause is said to be a unit clause, if it contains exactly one literal.

Conjunctive normal form (CNF) A propositional formula is said to be in conjunctive normal form (CNF), if it is a conjunction of clauses. Here are some more definitions related to CNF that are used later in this thesis, in chapter 6.

3-CNF When each conjunct contains a disjunction of *at most three* literals, the formula is said to be in *3-CNF*.

Conversion to CNF Given an arbitrary boolean formula F, there exists a polynomial algorithm to convert it to a CNF formula, F', such that it is *equisatisfiable*, i.e. F' is satisfiable if and only if F is [Tseitin, 1968].

CNF and satisfiability It follows from the definition that a given CNF formula is satisfiable iff all its clauses are satisfiable.

Pure literal Used mostly in the context of a CNF representation, a literal is said to be pure (in the context of the given formula) if its negation does not occur in the formula.

Empty problem An empty CNF problem, i.e a CNF problem with no clauses is valid.

Set representation In the material discussed in this thesis, we represent a SAT problem in CNF, as a *set* of clauses and a clause as a *set* of literals.

SAT Given a propositional formula, the problem of finding whether there exists a variable assignment such that the formula evaluates to true is called the propositional satisfiability problem, also referred to as boolean satisfiability problem and is abbreviated as *SAT*.

Tautology checking As defined earlier, a given formula is a tautology if its negation is unsatisfiable and it is not a tautology if the negation is satisfiable. Thus, the problem of finding if a given propositional formula F is a tautology is equivalent to finding if $\neg F$ is unsatisfiable.

4.3 First-order logic

Propositional logic allows us to build formulas only from propositional variables. First-order (predicate) logic extends propositional logic by accommodating the following (described in detail below):

- Variables refer to individual entities, rather than truth values
- Propositions can be built from non-propositional (domain) variables and constants using *functions* and *predicates*
- Quantifiers, *universal* and *existential* : $\forall, \exists$
- *Bound* variables: non-propositional variables can be *bound* with *quantifiers*

This section describes relevant background related to first-order logic, useful for understanding material discussed in this thesis, in particular chapter 7.

4.3.1 Syntax and semantics

Syntax The following notions describe the syntax of first-order logic

Vocabulary A first-order logic vocabulary consists of three sets. A set :

- $\mathcal{P}$, of predicate symbols, each with its associated *arity*, i.e. the number of arguments it expects
- $\mathcal{F}$, of function symbols, each with its associated *arity*, i.e. the number of arguments it expects
- $\mathcal{C}$, of constant symbols. Constants can be interpreted as 0-arity functions and so, the set of constant symbols can be subsumed in the set of function symbols. Thus, in most cases, the set of constant symbols is not specified explicitly in the vocabulary.

Variable is a place-holder for any, or some, unspecified objects/concrete values.

Term is used to refer to an object that we are talking about and terms can be: variables, constants and functions applied to those. In pseudo Backus Naur form, we may write a term, t, as follows:

$$t ::= x \mid c \mid f(t, \ldots, t)$$

where x ranges over `var`, a set of variables, c over 0-arity function symbols in $\mathcal{F}$, and f over those elements of $\mathcal{F}$ with arity $n > 0$.

It is important to note that

- the first building blocks of terms are constants and variables
- the notion of terms is dependent on the set $\mathcal{F}$. If it is changed, the set of terms also changes. The same holds true for the set of formulas (defined below), when $\mathcal{F}$ is changed.

Predicate takes a fixed number (referred to as *arity*) of terms as arguments. It evaluates to a truth-value, when its arguments evaluate to domain elements and a valuation function for the variables is given.

Function takes a fixed number (referred to as *arity*) of terms as arguments. It is a term and evaluates to an element of the domain, when its arguments do.

Formula In pseudo Backus Naur form, a first-order logic formula, P, is as follows:

$$P := P(t_1, t_2, \ldots, t_n) \mid \neg P \mid P \wedge P \mid P \vee P \mid$$

$$P \rightarrow P \mid P \leftrightarrow P \mid \forall x P \mid \exists x P$$

where $P \in \mathcal{P}$ is a predicate symbol of arity $n \geq 1$, t_i are terms over $\mathcal{F}$ and x is a variable.

Connectives are operators that takes a fixed number (referred to as *arity*) of formulae as arguments giving a compound formula as result; the compound result has a truth value determined by the connective and the truth-values of the arguments. Classical first-order logic without equality builds on propositional logic with the following additional constructs called *quantifiers*: $\forall, \exists$, that are used with variables and terms.

Quantifiers The formula, $\forall x$ P, where x is a variable and P any formula, means intuitively, 'for *all* values of x, P is true'. For this reason, $\forall$ is referred to as the *universal quantifier*. The analogous formula $\exists x$. P, means intuitively, 'there exists an x such that P is true', i.e. 'P is true for *some* value(s) of x'. For this reason, $\exists$ is referred to as the *existential quantifier*. In the formulas, $\forall x$ P and $\exists x$. P, P is referred to as the *scope* of the quantifier. It is worth observing here that in first-order logic, quantifiers cannot be applied to functions or predicates. Logics where quantification over functions and predicates is permitted are said to be *second-order* or *higher-order*.

Bound variables, Free variables The quantifier is said to *bind* instances of x within its scope and these variable(s) are said to be *bound*. It is useful to note that renaming the *bound* variables does not affect the meaning of a formula. Instances of variables that are not within the scope of a quantifier are called *free* variables. Intuitively speaking, a bound variable is just a placeholder referring back to the corresponding binding operation, rather than an independent variable in the usual sense.

Signature, language When we talk of a *signature* of first-order logic, we refer to the pair of sets, of functions and predicates, both as name-arity pairs and the corresponding *language* as the sets of terms and formulas that can be built using only functions and predicates appearing in that signature (but

any variables)[6].

Terms vs Formulas In first-order logic, a syntactic distinction is made between *formulas* and *terms*. Formulas are intended to be *true* or *false*. Terms stand for objects in the domain of discourse and are in turn, built from variables using functions.

Notation

- We use lower case letters for variables and arity and upper case letter for all other symbols.
- The order of precedence of symbols in a formula is as follows:
  - $\neg$, $\forall y$ and $\exists y$ bind most tightly
  - $\vee$ and $\wedge$
  - $\rightarrow$, $\leftrightarrow$

Substitution Given a variable x, a term t and a formula ϕ, a substitution, $\phi[t/x]$, is defined to be the formula obtained by replacing each free occurrence of the variable x in ϕ, with t.

More concretely, a substitution is a finite set of replacements $[t_1/x_1, \ldots, t_k/x_k]$ (a function from variables to terms), where $x_1, \ldots, x_k$ are distinct variables and $t_1, \ldots, t_k$ are terms.

The finite set $x_1, \ldots, x_k$ is called the *domain* of the substitution. A given substitution, ϕ, can be defined to apply over arbitrary terms and formulae, by defining xϕ = x if x not in domain ϕ.

A given substitution ϕ can be extended to accommodate terms, constants and literals as well by augmenting the definition with $x\phi = x \forall x \notin domain(\phi)$. A pair t_i/x_i is called a *binding* for x_i. The extension, composition and equality operators are defined in a natural way.

Substitution and free variables While performing the substitution $\phi[t/x]$, the term t may contain a variable y, such that the occurrences of x in ϕ are under the scope of $\exists y$ or $\forall y$ in ϕ. In such cases, as a result of the substitution, the value y, which might have been fixed by a concrete context, gets caught in

[6]The exact formal definitions of *language* and *signature* vary in the literature. The key objective though is that the concept of a term or formula being in a restricted language is clear

the scope of a quantifier. So, we use a *capture-avoiding* substitution, where such bound variables are renamed, before carrying out the subsitution.

Semantics As with a propositional formula, the meaning of a first-order formula is defined recursively and depends on, and varies with, the actual choice of values and the meaning of the predicate and function symbols involved. To describe the notion of semantics for first-order logic, we require the following definitions

Interpretation, valuation In first-order logic, the variables, function symbols and predicate symbols all need to be interpreted. It is customary to separate these concerns, and define the meaning of a term or formula with respect to both an *interpretation*, which specifies the interpretation of the function and predicate symbols, and a *valuation*, which specifies the meanings of variables. Mathematically, an *interpretation* M consists of the following three parts:

Domain A nonempty set D called the *domain* of the interpretation. The intention is that all terms have values in D.

Interpretation of functions A mapping of each *n-ary* function symbol f to a function $f_M : D^n \to D$.

Interpretation of predicates A mapping of each *n-ary* predicate symbol P to a boolean function $P_M : D^n \to \{false, true\}$. Equivalently, we can think of the interpretation as a subset $P_M \subseteq D^n$.

Value of a term The value of a term in a particular interpretation M and valuation v is defined by recursion, taking note of how all variables are interpreted by v and function symbols by M:

$$\texttt{termval}\ M\ v\ x = v(x),$$
$$\texttt{termval}\ M\ v(f(t_1, \ldots, t_n)) = f_M(\texttt{termval}\ M\ v\ t_1, \ldots\ \texttt{termval}\ M\ v\ t_n)$$

4.3.2 Satisfiability, logical equivalence, validity

Whether a formula `holds` (i.e. has the value 'true') in a particular interpretation M and valuation v is defined by recursion and mostly follows the pattern described earlier for propositional logic. The definitions are given below. The main added complexity is specifying the meaning of the quantifiers. We intend that $forall.x\ P(x)$ should hold in

a particular interpretation M and valuation v, precisely if the body *P(x)* is true for *any* interpretation of the variable x, i.e. if we modify the effect of the valuation v on x in any way at all.

$$\texttt{holds}\ M\ v \bot = false$$

$$\texttt{holds}\ M\ v \top = true$$

$$\texttt{holds}\ M\ v\ (R(t_1, \ldots, t_n)) = R_M(\texttt{termval}\ M\ v\ t_1, \ldots, \texttt{termval}\ M\ v\ t_n)$$

$$\texttt{holds}\ M\ v(\neg p) = \neg(\ \texttt{holds}\ M\ v\ p)$$

$$\texttt{holds}\ M\ v(p \wedge q) = (\ \texttt{holds}\ M\ v\ p)\ and\ (\ \texttt{holds}\ M\ v\ q)$$

$$\texttt{holds}\ M\ v(p \vee q) = (\ \texttt{holds}\ M\ v\ p)\ or\ (\ \texttt{holds}\ M\ v\ q)$$

$$\texttt{holds}\ M\ v(p \rightarrow q) = (not\ (\texttt{holds}\ M\ v\ p\ ))\ or\ (\ \texttt{holds}\ M\ v\ q)$$

$$\texttt{holds}\ M\ v(p \leftrightarrow q) = (\texttt{holds}\ M\ v\ p = \texttt{holds}\ M\ v\ q)$$

$$\texttt{holds}\ M\ v(\forall.x\ p) = for\ all\ a \in D, (\texttt{holds}\ M((x \mapsto a)v)p)$$

$$\texttt{holds}\ M\ v(\exists.x\ p) = for\ some\ a \in D, (\texttt{holds}\ M((x \mapsto a)v)p)$$

Validity, logical equivalence By analogy with propositional logic, a first-order formula is said to be *logically valid* if it holds in all interpretations and *and* all valuations. If $p \leftrightarrow q$ is logically valid, we say that p and q are *logically equivalent*.

Satisfiability We say that an interpretation M *satisfies* a formula P, or simply that P `holds` in M, if *for all valuations* v, we have $\texttt{holds}\ M\ v\ p = \texttt{True}$. Similarly, we say that M satisfies a *set* of formulas, or that S `holds` in M, if it satisfies each formula in the set. We say that a first-order formula or set of first-order formulas is *satisfiable* if there is *some* interpretation that satisfies it.

Model An interpretation that satisfies a set of formulas Γ is said to be a *model* of Γ. The notation $\Gamma \models P$ means 'P holds in all models of Γ'. When Γ is the empty set, we just write $\models P$

4.4 Theorem proving

We use the term *automated/mechansied reasoning systems* with the following interpretation: (i) *reasoning* is understood as formal deductive inference as practiced in formal

logic (ii) the term *automated/mechansied systems* is used to broadly include classes of software systems that are capable of performing the *reasoning* without or with (partial or step-by-step) human intervention. We use the terms *automatic/fully automatic* and *interactive* systems to refer to the two classes respectively. A theorem proving problem is typically specified in a given *logic*, say, L, as follows:

Given a set of axioms (assumptions), A and a conjecture (goal) G to prove,
is there a proof in L of G from the given axioms, A?
where A, G are specified in the given logic.

For a typical problem scenario, this translates to:

- the assumptions capture all the relevant available information
- the conjecture expresses the question being asked

The problem is given to an automated reasoning system to work on until it arrives at an answer or until it runs out of resources or the execution is terminated by the user.

4.4.1 Inference system

An *inference/deduction system* is a mechanism that allows for the construction of valid logical statements from other valid ones by *purely syntactic means*. An *inference rule* gives a method of deriving valid formulas (conclusions), from a set of given formulas (premises), by purely syntactic means, i.e. without using any semantic information.

A proof calculus is the formalisation of the deductive machinery of choice. A given automated reasoning system implements a specific deductive machinery via a particular proof calculus. The inference rules that are part of the proof calculus are called *basic* inference rules in contrast to the rules that can be derived, which are referred as *derived* rules. The emphasis is on the use of purely *syntactic means*, i.e. based purely on the *form*, hence the alternative name *formal rules/systems*.

The choice of a proof system depends, amongst other things on: the logic, the application domain, the intended mode of operation of the system (automatic, interactive etc). Natural deduction based systems, sequent calculus, axiomatic systems and tableaux systems are examples of inference systems. We describe the natural deduction system and the sequent calculus in §4.4.2, as these are relevant to the material discussed in the first-order theorem proving case study discussed later in the thesis.

4.4.2 Natural deduction

Natural deduction is a style of inference that captures the reasoning patterns used by humans, more closely than axiomatic systems, hence the qualification *natural*. It consists of rules for introducing and eliminating each of the logical connectives and quantifiers. Despite the *natural* tag, the deduction still is a formal system as in: it allows us to manipulate formulae and derive conclusions by purely syntactic means, regardless of their meaning.

As an example, consider the following, for the case of propositional logic: suppose that, by assuming P is true, Q can be shown to be true, by virtue of some intervening proof steps. Then, by making a semantic argument using the truth table for the connective $\rightarrow$, we can conclude that $P \rightarrow Q$ holds. This conclusion does not depend on any assumption. The assumption of P being true was made within the proof and was *discharged* in the process of going from Q to $P \rightarrow Q$. This is an illustration of a method for *introducing* the connective $\rightarrow$ and implicitly generating a new formula. Similar arguments follow for *eliminating* the connectives from a formula. E.g., the elimination rule for $\rightarrow$ captures the well known *modus ponens*. It says that if you know $P \rightarrow Q$ and you know that P is true, then Q holds. Similar such rules can be formulated for other connectives.

4.4.3 Sequent calculus

In natural deduction, proofs are constructed by fitting the rules together, in the form of a tree. As in ordinary reasoning, temporary assumptions may be made, in the course of the proof and then discharged by incorporating them into the conclusion. The proof-tree form of proofs in the natural deduction system, in their crude form, do not lend themselves well to reasoning about them and/or to incorporate them in a software system etc.. *Sequent calculus* addresses this well. It is a less pictorial and more algebraic formulation of natural deduction in which the role of assumptions is made more explicit. It provides a means of reasoning about proofs and axiomatising deduction.

Natural deduction and sequent calculus, by virtue of capturing the behaviour of the logical connectives (independent of the logic), gives us the opportunity to generate different logics by varying the rules. This has led to their use in the engineering of theorem provers aimed at providing a generic theorem prover approach [Paulson, 1989].

Definitions

Sequent Though the rules of sequent calculus affect logic formulae, the objects of manipulation are not logic formulae, but *sequents*. A *sequent* is of the form: $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi_1, \psi_2, \ldots, \psi_m$, where:

- $\phi_1, \phi_2, \ldots, \phi_n$ and $\psi_1, \psi_2, \ldots, \psi_m$ are lists of formulae.
- A formula appearing by itself on either side of the turnstile symbol denotes a singleton set.
- For a given interpretation, if the sequent holds, it means the following:

  If all ϕ_is are true, then, at least one of the ψ_js is true.

Premises, conclusion For convenience, a sequent is often represented as

$$\Gamma \vdash \Delta,$$

where both Γ and Δ are (possibly empty) sets of formulae. Γ is the sequent's *antecedent/premises* and Δ its *succedent/conclusion*. A special case is

$$\vdash \psi$$

, which has the same meaning as ψ.

Role of Sequent It is useful to note that a sequent is not a formula and the symbol $\vdash$ (known as *turnstile* is not a connective. Furthermore, the sequent captures the *intention* of being able to apply inference rules to the premises, repeating the process if necessary, to eventually obtain the conclusion.

Valid sequent A *valid* sequent is one that is true under every interpretation. Thus, referring to the above point on the role of a sequent, we can say that a valid sequent gives the *intention* of the status of *certainty*.

Basic sequent A sequent is called *basic* if both sides share a common formula. Such sequents are clearly valid.

Left rules, right rules Sequent calculus rules come in pairs, to introduce each connective on the left or right of the $\vdash$ symbol. For first-order logic, there are *:left* and *:right* for each connective and quantifier. The sequent calculus rules for classical first-order logic without equality are given in Table 4.3.

4.4.4 Backward proof and sequent calculus

Though the inference rules given in Table 4.3 render themselves to forward-reasoning at first glance, such a usage to find a proof for a given conjecture entails enumeration of all the possible derivations using the given premises. This approach is used in *tableaux* based methods and is not addressed in this work. An alternative approach is to find a proof using a backward style of proving, often referred to as *refinement* or *backward proof*:

- Start from the initial goal, i.e. the given sequent that is to be proved. At this stage, this is the root of the proof tree and its only leaf is this goal
- Apply a sequent rule to one of the leaves. Here, the leaf (goal) plays the succedent and the application of the rule generates sub-goals, which are in turn, the antecedents of the applied rule. Thus, the leaf is now transformed into a branch node with one or more leaves (sub-goals).
- The above step is performed recursively until all the leaves are basic sequents (success) or when no rules can be applied to a leaf any more (failure).
- For propositional logic, this procedure must terminate, though this is not the case for first-order logic.

4.4.5 Interactive theorem proving

In interactive theorem proving systems, the human user guides the proof process, with the possible assistance of the machine (possibly to do some of the tedious/mundane bits or to marshal the power of automation using encodings of specific proof search procedures and heuristics), while the system still ensures that no mistakes are made, i.e. that the proof produced eventually, is *sound*.

The interactive aspect naturally fed the need for *programmability* of the theorem prover: the user should be able to extend the built-in automation as much as desired, while still being able to allow only extensions that are sound. In the next section, we describe LCF (Logic for Computable Functions), which started as a system that addressed the dual needs of interactive aspects and programmability and has gone on to become one of the most influential foundations of interactive theorem proving. It has formed the basis

for many successful interactive theorem provers, e.g. Isabelle [Nipkow et al., 2002], Nuprl [Constable et al., 1986]. In this work, we use the terms *LCF based approaches, LCF style provers* to refer to such systems and use the following terms synonymously: programmable theorem provers, tactic-based theorem provers and LCF-style theorem provers.

4.4.6 LCF

In the LCF approach,

- The commands are embodied in a language that has an expressive *functional* subset [7].
- Each inference rule of the logic is expressed as an ML function, which has as its result a value of the special *abstract type* (say, `thm`). This special abstract type, which stands for proved theorems in the implementation language, is in fact one of the key LCF ideas.
- The only constructors of the abstract type `thm` correspond to approved inference rules. This ensures that anything of type `thm`, must by construction, have been *proved* rather than simply *asserted*.

However, the user is given full access to the implementation language and can use any programming techniques of the implementation language to engineer more sophisticated ways of orchestrating the basic inference rules. As `thm` is an *abstract type* with specific constructors as discussed above, any result of type `thm`, in which ever way it was arrived at, must ultimately have been produced by correct application of the primitive rules. This holds no matter how complex the means of arriving at that was. Thus, it allows for both **programmability** of the prover as well as guaranteeing the **soundness** of the programmed extensions. In practice, the implementation language for most interactive theorem provers is usually a flavour of ML (Meta language).

LCF style provers use a predominantly goal-directed, backward-chaining style of proof (§4.4.4). The notion of *tactics* helps to realise this in an efficient manner. Tactics,

[7] LCF and the functional programming language ML (Meta Language) are very closely related, with the latter having had its genesis in the development of (Edinburgh) LCF; ML was the precursor to Standard ML (SML)

are essentially the rules of inference, with intended usage in the backwards direction, equipped with extra book-keeping mechanisms. Using tactics, we can *formalise* the idea of working backwards from a goal to (possibly simpler) sub-goals. This equips us with the tools to program some general purpose problem solving strategies.

Another feature of LCF is the following: when a rule gets used (in the backward style), giving a list of sub-goals, the *justification*, the reason why it was a legitimate step (i.e. the name of the inference rule), has to be kept track of. In LCF, this is taken care of by *tactics*. Tactics are thus functions which encapsulate an inference rule and maps a goal to a list of sub-goals while maintaining the *justification*.

Thus, a typical step in an LCF prover will involve: finding a tactic (rule) whose conclusion can be made to match the goal (sub-goal) and read off the premises of the rule thus found to give the sub-goals. Keep using this basic strategy until all sub-goals reduce to axioms or previously proved theorems. The challenge that this introduces is that at each step there will be many matches (of tactics). So, an efficient search mechanism will be required to make sure that we try all possibilities. *Tacticals* provide the tools to address this aspect.

Tacticals are encapsulation of control structures, for applying the tactics in various ways (sequencing, conditional operation, repetition etc).

LCF systems have a kernel, which consists mechanisms to apply the basic inference rules. All other proof rules are defined in terms of these rules. Thus, it suffices to just trust the small kernel. This is a very desirable feature, particularly for prototyping of and experimentation with sophisticated techniques. We have used one such system for first-order logic, in our case study discussed in chapter 7.

4.5 SAT solvers: some relevant background

The propositional satisfiability problem, often abbreviated as SAT, was the first problem to be shown as being NP-complete [Cook, 1971] and thus is of significant theoretical importance. Despite its NP-complete status, many industry-standard SAT solvers

have been developed that have been used to tackle real world problem instances of up to a million variables.

SAT solvers are being used increasingly in a wide range of application domains. Recent advances have pushed the tractability threshold of industry-standard SAT solvers both in terms of problem-size (number of variables) and complexity. The electronic design automation (EDA) industry has increasingly adopted SAT engines for a wide variety of testing and verification tools like automatic test pattern generators, equivalence checkers, property checkers. SAT is also increasingly being used for software verification and debugging. Outwith the hardware and software verification community, SAT has also been used widely for other domains like: configuration management such as resolving software package dependencies and checking consistency of technical documentation [Sinz et al., 2006].

Key propositional logic related definitions and notations used in this thesis were provided in §4.2.3. In this section, we provide the following aspects of SAT related background that are particularly relevant to the work described in this thesis:

- Detailed descriptions of the DPLL and Stalmarck algorithm
- Overview of some key techniques used in state-of-the-art DPLL-based SAT solvers

For more details about SAT related background and recent advances, the reader is referred to [Biere et al., 2009]. [Harrison, 2009] is a good reference for general background on SAT and details on the workings of the Stalmarck algorithm, in particular.

4.5.1 SAT algorithms: an overview

Algorithms for SAT can be broadly classified as below.

Complete algorithms These algorithms can prove both satisfiability and unsatisfiability. Some complete SAT algorithms are:

- Resolution based algorithms: DP [Davis and Putnam, 1960], DPLL [Davis et al., 1962]
- Stalmarck's method [Sheeran and Stalmarck, 1998, 2000]
- Recursive learning [Kunz and Pradhan, 1994]

- Algorithms based on Binary Decision Diagrams (BDDs) [Drechsler and Becker, 1998]

Incomplete algorithms These algorithms cannot prove unsatisfiability. Some of these algorithms apply probabilistic techniques to solve the SAT problem and some consider the SAT CNF problem as a discrete optimisation problem of maximising the number of satisfied clauses. Examples of algorithms in this category are:

- Local search [Selman et al., 1996]
- Randomised restarts [Gomes et al., 1998]
- Simulated annealing [Kirkpatrick et al., 1983; Spears, 1993]
- Hill climbing [Gent and Walsh, 1993]

In the work described in this thesis, we consider parallelisation for only the complete category. We describe the DPLL and Stalmarck algorithms in detail in the following sections.

4.5.2 DPLL

In this section, we describe the DPLL algorithm [Davis et al., 1962] and provide an overview of key techniques used in modern DPLL-based SAT solvers.

4.5.2.1 The DPLL algorithm

The long established and popular DPLL algorithm [Davis et al., 1962], follows a depth first search approach. It uses branching, *unit clause propagation* and *pure literal* detection. An informal description of the algorithm is given below. A code fragment describing a functional implementation of a recursive version of this algorithm is given later in the thesis in Listing 6.2.

1. Branch Given a CNF formula, the algorithm heuristically selects an unassigned variable and assigns it either *true* or *false*. This is referred in the literature (synonymously) by any of the following terms: *case-split, branching, decision-point*.

2. Apply inference rules The solver then tries to deduce the consequences of the assignment made using the following inference rules:

Unit clause rule Let C be a unit clause consisting only the literal, v. Obviously, C is true iff v is true. In the context of the CNF representation, this becomes a powerful inference rule to apply to reduce the problem, during the search for a satisfiable assignment. Because, for the CNF formula to be true, every clause has to be true, including the unit clauses and this in turn, implies that the sole variable in the clause has to be true. This is added to the assignment and applied throughout the problem which can in turn, reduce the problem further. Iterated exhaustive application of the unit clause rule (i.e. until it can no longer be applied) is performed. This is referred to as *unit propagation*. This is the key inference rule for the DPLL algorithm. As evident from the description, the unit clause rule and consequently, the DPLL algorithm relies crucially on the CNF representation.

Pure literal rule For the purpose of finding a satisfiability assignment for a CNF formula, pure literals can be assigned the value True and the clause of occurrence (which is now true) can be dropped from the problem. This is also used in an iterative, exhaustive manner, but has been dropped out of most modern SAT solvers as it is observed to slow down the algorithm and the benefits of its use are not sufficient enough, to justify its use.

3. Satisfying assignment found/Backtrack After applying the inference rules, the algorithm can reach a state with the following 3 possibilities:

SAT The problem is empty, i.e. all clauses have been satisfied. The algorithm terminates with the answer *SAT* with the current assignment as a possible satisfying assignment.

Conflict, Backtracking When the algorithm encounters an empty clause, i.e. the problem has been rendered unsatisfiable by the current assignment, a *conflict* is said to have occurred. Occurrence of a conflict means that a satisfying assignment cannot be reached by using the current assignment. So, the algorithm *backtracks*, to try a different branch value for the most recent decision level. If both branches have been explored at that level, it backtracks to the earlier decision level and continues applying the inference rules, i.e. applies Step-2 as above. If there are no more variables to branch on and/or no more decision levels to backtrack to, it means that the entire search space has been explored without finding a satisfying assign-

ment. So, the algorithm terminates with the answer *UNSAT*. It is useful to observe here that the original algorithm thus incorporates what is now termed *chronological* backtracking, relying only on the nesting level of the tree.

It is worth observing here that from an implementation point of view, there are significant number of state related operations that happens here, as the algorithm has to throw away the current problem state and use the assignment at the level to which it has backtracked, along with the original problem instance.

Unknown If the application of the inference rules did not lead to either SAT or conflict, the algorithm continues with Step-1, i.e. branching.

4.5.3 Stalmarck's algorithm for SAT

Stalmarck's algorithm [Sheeran and Stalmarck, 2000; Stalmarck, 1992; Stalmarck and Saflund, 1990a] is an algorithm for checking if an arbitrary propositional formula (not necessarily in CNF) is a tautology or not. For the case of SAT, one can equivalently check if the negation of the given formula is a tautology.

Stalmarck's method is a proof procedure for classical propositional logic and has been implemented in a suite of commercial tools called *NP-Tools*, engineered by the company *Prover Technology* (www.prover.com). This suite has been successfully used in real world industrial verification projects containing millions of sub-formulas in the areas of telecom service specification analysis, analysis of railway interlocking software, analysis of programmable controllers and analysis of aircraft systems [Borälv, 1997]. Furthermore, Stalmarck's method has been found to perform better than BDD based methods and the *Otter* prover [McCune, 1994] for some classes of real world verification problems [Groote et al., 1995]. The implementation related aspect of representing a propositional logic formula as a set of triplets, which plays a pivotal role in the Stalmarck procedure, is covered by a patent [Stalmarck, 1992].

Definitions

For a given formula *X*, let *S(X)* be the set containing all subformulas of *X*, including True ($\top$), False ($\bot$) and the complements of subformulas of X. Then, a *formula relation* on X is defined as an equivalence relation with domain, *S(X)*, with the following additional qualifications and notations:

- $A \sim B$ means that A and B are in the same equivalence class and must have the same truth value
- If $A \sim B$, then $\neg A \sim \neg B$
- $A \nsim B$ is encoded as $\neg A \sim B$, thus allowing for encoding of both equalities and inequalities between subformulas
- $R(A \equiv B)$ refers to the least formula relation containing R and relating A and B; $A \equiv B$ is referred to as an *association*.
- X^+ refers to the identity relation on S(X), placing each element of S(X) in its own equivalence class; $X^\top$ refers to $X^+(X \equiv \top)$; $X^\bot$ is defined in a similar way Note that $X^\top$ constitutes a *partial valuation* and plays an important part in the algorithm
- If a formula and its complement are in the same equivalence class, it signals an *explicit contradiction*
- Union and intersection of the equivalence classes are defined in the standard way.
- These equivalence classes are of particular interest when (i) $\top$ is a member (ii) $\bot$ is a member.

Notion of triplets

The algorithm uses a data structure called *triplets* to represent compound formulas. This is explained via the definitions and example below.

- A *triplet* (x, y, z), for a connective $\oplus$, is an abbreviation for

  $$x \leftrightarrow y \oplus z$$

  where $\oplus$ can be any boolean connective, and the variable x represents a subformula of the original formula.

- Any arbitrary propositional formula can be reduced to a set of triplets by introducing new variables (when needed) to stand for subformulas. For the purpose of this thesis, we make a distinction between these newly introduced variables and the variables present in the given formula by referring to them as *triplet* variables and *original* variables respectively. An example is provided in Example 4.1.

Example 4.1 Triplets

The formula $p \rightarrow q \rightarrow p$ gives the following triplets, for the connective, $\rightarrow$:
(b1, q, p) (b2, p, b1)
where p, q are the *original* variables and b1 and b2 are the *triplet* variables, with b1 standing for the subformula $q \rightarrow p$ and b2 for the entire formula

- When a given *triplet* (x, y, z) is *explicitly contradictory* i.e. it signals a contradictory propositional formula when expressed as $x \leftrightarrow y \oplus z$, it is said to be a *terminal* triplet. E.g., the triplet $(\top, \top, \bot)$, is a terminal triplet for the $\rightarrow$ connective.

  For the sake of convenience, we adopt the convention that for the connective $\rightarrow$, the only *terminal* triplets are $(\top, \top, \bot)$**,** $(\bot, x, \top)$**,** $(\bot, \bot, x)$. It is easy to see that any explicitly contradictory triplet is equivalent to one of these forms.

The *triplet* representation plays an important part in the algorithm. The algorithms works by assigning truth values to the triplets (i.e. to the subformulas) and deriving the consequences by using the inference rules and recording equivalences between the triplets (subformulas). Thus, it serves as a shorthand notation to capture (sub)formula relations.

Simple rules

For each connective, a set of *simple rules*, also referred to as *trigger rules* are defined, using the notion of triplets. Intuitively, for a given triplet (p, q, r), a simple rule captures the obvious deductions when p is equivalent to another formula, including True ($\top$) and False ($\bot$). The unifying pattern for the rules is the following:

If all the preconditions hold, then the conclusions must hold.

The simple rules for the connective $\wedge$ are given in Table 4.2. Similar rules apply for other propositional connectives as well.

If ...	**Then ...**
$p = \neg q$	$q = \top$ and $r = \bot$
$p = \neg r$	$q = \bot$ and $r = \top$
$q = r$	$p = r$
$q = \neg r$	$p = \bot$
$p = \top$	$q = \top$ and $r = \top$
$q = \top$	$p = r$
$q = \bot$	$p = \bot$
$r = \top$	$p = q$
$r = \bot$	$p = \bot$

Table 4.2: Stalmarck trigger rules for the connective $\wedge$, i.e. for the formula, $p \leftrightarrow q \wedge r$

Applying a *simple rule* to a set of triplets gives a new set of triplets obtained by substituting the newly derived variable instantiations if any. A small example illustrating this is given below (see Example 4.2). The *simple/trigger rules*, along with the *dilemma rule* described below (§4.5.3) provides a complete proof system for classical propositional logic [Sheeran and Stalmarck, 2000].

Example 4.2 Application of simple rules

Referring to Example 4.1 above, let us assume $b2$ (which corresponds to the entire formula $p \rightarrow q \rightarrow p$) to be *False* and apply the simple trigger rules.
For the triplet $(b2, p, b1)$, if b2 = $\bot$ then $p = \top$ and $b1 = \bot$. Substituting this newly derived information to the triplet (b1, q, p) gives ($\bot$, q, $\top$), which is a *terminal* triplet.

Thus, we started with the assumption that $p \rightarrow q \rightarrow p$ is false and we have derived a terminal triplet, i.e. a contradiction. So, we conclude that the formula is valid.

Using the equivalence classes

The equivalence classes on subformulas defined earlier is used in the following ways, in the context of the Stalmarck procedure:

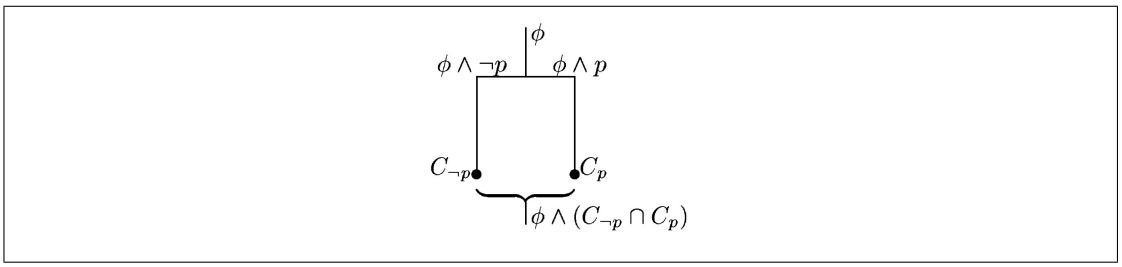


Figure 4.1: Branch-merge rule: applies a case-split and garners conclusions from the two branches

- It extends the scope of possible derivations: instead of just deriving some formulas to be true or false, one can also derive the knowledge that certain (sub)formulas are equivalent, i.e. certain sets of formulas have the same truth value. This feature gives more power to the *simple* rules and consequently the proof procedure.

- From the point of view of a refutation procedure, the derivation of a contradictory formula relation from $X^{\top}$, constitutes a refutation of the formula X. This notion can be extended to tautology checking by attempting to refute $X^{\bot}$.

- The equivalence classes also play a pivotal role in the algorithmic description of the dilemma rule as defined below.

- If an *explicit contradiction* (see §4.5.3 for definition) has been derived in the course of a derivation, the relation can be deemed to be equivalent to that with a single equivalence class and the derivation can be stopped.

A desirable by-product of the use of (sub)formula relations is the potential scope to gainfully exploit the implicit structural information present in many real world SAT instances. As discussed in §2.1.1, DPLL-based SAT solvers do not fare well in this aspect of utilisation of implicit structural information [Thiffault et al., 2004].

R
R1: the derivation $R(A \equiv B)$
R2: the derivation $R(A \equiv \neg B)$
$R_1 \sqcap R_2$

Figure 4.2: Dilemma rule, a branch-merge rule, implemented using equivalences, $\sqcap$ denotes intersection

Dilemma rule

The *dilemma rule* (Figure 4.2) is a *branch and merge* rule (Figure 4.1). For a given formula relation R, application of the dilemma rule involves the following steps:

- Choose A and B from different (and non-complementary) equivalent classes in R.
- Obtain the derivations R_1 and R_2, obtained by exhaustive application of the simple rules to the two independent branches: $R(A \equiv B)$ and $R(A \equiv \neg B)$ respectively.
- Extract $R_1 \sqcap R_2$, i.e. the conclusions that are common to both branches (*merge* operation) with $\sqcap$ defined as below
- $R_1 \sqcap R_2$ is defined as:
  - R_2 if R_1 is *explicitly contradictory*
  - R_1 if R_2 is *explicitly contradictory*
  - $R_1 \cap R_2$ otherwise, where $\cap$ is understood as set intersection
- Choosing $B \equiv \top$ gives the case of the two branches being $A \equiv \top$ and $A \equiv \bot$, i.e. one where some propositional variable is assumed to be true and one where it is assumed to be false
- Thus, intuitively, the dilemma rule can be understood as: any information that holds for **both** the truth values of a propositional formula, i.e. when x is true **and** when x is false, must hold independent of the value of x, i.e. it is (universally) consistent information. Thus, the knowledge derived in the course of the Stalmarck algorithm can be used by another algorithm applied to the same problem. This is a valuable feature which we use in our hybrid SAT solver which we describe in chapter 6.

Saturation procedure

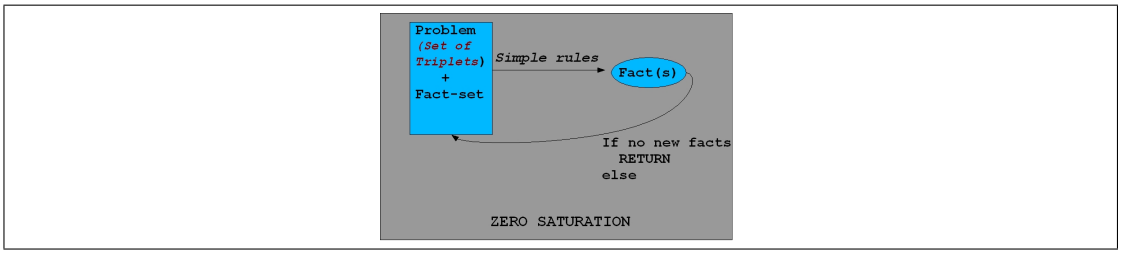


Figure 4.3: 0-saturation procedure of Stalmarck's algorithm

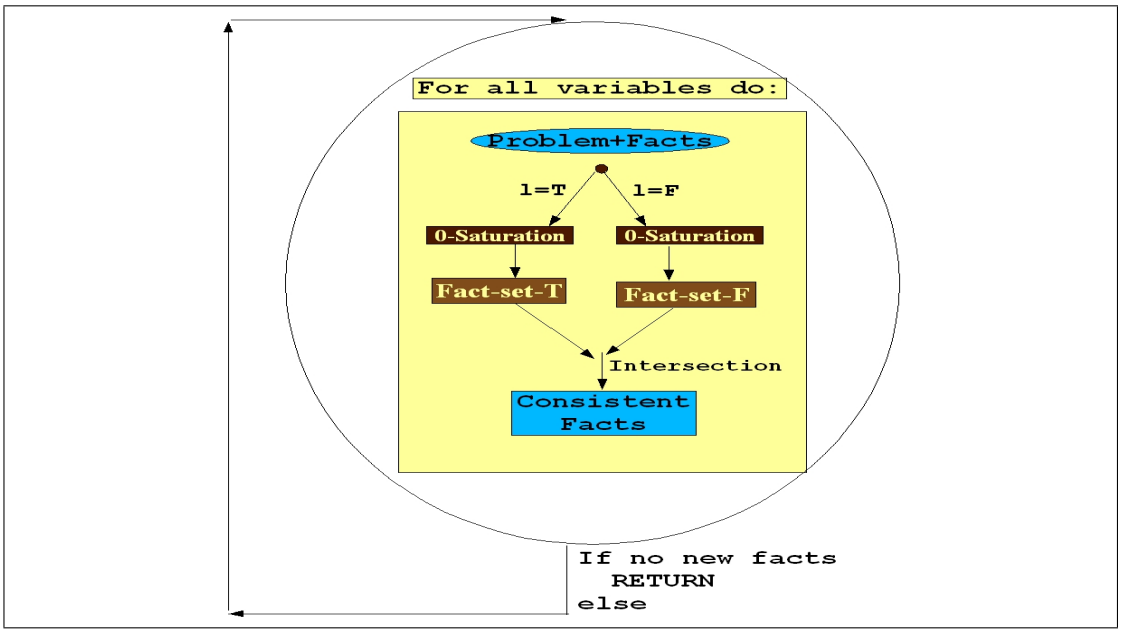


Figure 4.4: 1-saturation procedure of Stalmarck's algorithm

Listing 4.3: Recursive saturation procedure for Stalmarck's algorithm

```
saturate(R,k+1) = repeat
 L := Sub(R);  R' := R
 for each l in L
 do
   R1 = saturate(R(l equiv FALSE),k)
   R2 = saturate(R(l equiv TRUE) ,k)
   if contradictory(R1) and contradictory(R2)
   then return R1 union R2
   else if contradictory(R1)
   then R = R2
   else if contradictory(R2)
   then R = R1
   else R = R1 intersect R2
until R' = R
return R
```

In the Stalmarck algorithm, the proof system consisting of the simple rules and the dilemma rule is embedded in a *saturation* framework, i.e. exhaustive application of the rules until no more new information (no more new equivalences) can be derived. This allows for the following valuable feature: recursive learning and incorporation of information gathered.

Given an equivalence relation, i.e. a set of equivalences between (sub) formulas, *0- saturation* tries to derive as many new equivalences as possible, by *exhaustively* applying the simple rules and using the properties of symmetry, transitivity and involution of negation where applicable.

In practice, 0- saturation starts with an equation (between two triplets), applies a related simple rule and derives the consequences (which are in the form of equations themselves). It continues to apply the simple rules on those triplets whose variables were affected by the consequences of the earlier application(s). The process continues until no further simple rules can be applied. It augments the equivalence relation with the newly derived consequences. Example 4.2 provides a simple example illustrating 0- saturation. Listing 4.3 gives the pseudocode for *k+1-saturation*, defined in terms of branching and *k-saturation*.

4.6 Relevant key characteristics of Stalmarck's algorithm

Some of the key strengths of the method that have contributed to its success in the hardware domain and other industrial applications [Borälv, 1997] are as follows:

- Ability to exploit the structure of the given formula via *(sub)formula relations*, a key benefit compared to the CNF based methods like DPLL where the CNF conversion often entails loss of (implicit) structural information. E.g., it is known to fare much better than DPLL for Urquhart problems [Urquhart, 1987] and pigeon hole problems [Haken, 1985]. Both these classes of problems are easy for a human to solve because of the inherent structure, but yet, they have been proved hard for DPLL-based solvers.

- The recursive *saturation* algorithm which allows for continuous gathering of information in the form of formula relations. This has enabled the algorithm to efficiently search for shallow sub-formula proofs and this in turn, has turned to be an efficient strategy to tackle many industrial problems.

- The *saturation* aspect further *distinguishes* the algorithm from both *breadth-first-search and iterative deepening* [Sheeran and Stalmarck, 2000].
- Intuitively, the dilemma rule can be understood as: any information that holds for **both** the truth values of a propositional formula (when x is True and when x is false) must hold independent of the value of x, i.e. it is (universally) consistent information. Thus, the knowledge derived in the process of the Stalmarck algorithm can be shared by a different algorithm applied to the same problem. This is a valuable feature which we use in our hybrid SAT solver.
- The *learning* mechanism used in the Stalmarck algorithm has the following key advantages compared to the popular conflict driven clause learning (CDCL) [Marques-Silva et al., 1996] based techniques which are DPLL-based.
  - Stalmarck's algorithm learns by spanning the search tree in a breadth-first fashion whereas the DPLL-based CDCL techniques are restricted to the depth-first search space exploration. This makes it an ideal candidate to be used as a *complementary* learning mechanism with a DPLL-based solver.
  - The above mentioned point about loss of structural information applies to the CDCL techniques as well as they are DPLL-based and Stalmarck's algorithm fares better on this aspect.
- The method is more sensitive to the *hardness degree* of a formula (see §4.5.3, [Sheeran and Stalmarck, 2000]) than to its size in terms of number of variables or connectives. This makes it a good choice for application for real-world problems of a large scale as well.

Hardness criteria

Stalmarck's algorithm has an associated notion of proof hardness based on a novel proof-theoretic notion of proof depth which translates to minimum number of nested instances of the branch/merge rule required in any proof of a problem (formula). Roughly speaking, a formula's satisfiability is decidable by n-saturation, if it is decidable by the primitive rules and at most n-deep nesting of case-splits. A formula decidable by n-saturation is said to be n-easy, and if it is decidable by n-saturation but not $(n\text{-}1)$-saturation, it is said to be n-hard. For more details, the reader is referred to [Sheeran and Stalmarck, 2000; Stalmarck, 1994; Stalmarck and Saflund, 1990b].

The notion of proof hardness is of interest to us in this thesis for the following reason: With respect to this notion, the Stalmarck procedure is exponential in the hardness of the formula, but polynomial in the size of the formula, assuming a maximum degree of hardness [Stalmarck, 1994]. Thus, the method is much more sensitive to the hardness degree of a formula than to its size, in terms of the number of variables or connectives. Problems encountered in many real world applications have been found to have low degrees of hardness, typically less than 2 [Borälv, 1997].

4.7 First-order theorem proving: some relevant background

Definitions related to first-order logic were provided in §4.3. In this section, we provide background material related to relevant logical inference methods for first-order logic. For the purpose of the prototype prover discussed in this thesis (chapter 7), we consider classical first-order logic without equality and the material discussed in the rest of this section is to be taken in this context.

4.7.1 Unification

As described in §4.3, a *substitution* in first-order logic allows for free occurrences of variables in a formula to be replaced by terms[8]. The process of finding substitutions that make different logical expressions identical, is called *unification* and is a key component of all first-order inference algorithms.

Informally speaking, a unification algorithm gives a syntactic procedure for deciding on appropriate instantiations to make terms match up correctly when it is possible to do so and reports failure, when otherwise. An often cited analogy is that of solving a system of simultaneous equations in ordinary algebra. Just as a set of equations may not have a solution, so may a unification problem. A code-fragment is provided in Listing 4.4, describing a particular unification algorithm for *classical first-order logic without equality*. This has been used in the prototypical first-order prover discussed later in chapter 7.

[8]Given a variable x, a term t and a formula ϕ, a substitution, $\phi[t/x]$, is defined to be the formula obtained by replacing each free occurrence of variable x in ϕ with t.

Listing 4.4: Unification algorithm for first-order logic. The algorithm works by comparing the structures of the inputs, element by element. The substitution *mu* is built up along the way and is used to make sure that later comparisons are consistent with bindings that were established earlier.

```
Unify(x, y) = Unify_internal(x, y, [])

Unify_internal (x, y, mu)
 If (mu = Failure) then return Failure
 If (x=y) then return mu
 If (Is_a_variable(x)) then return Unify_variable(x, y, mu)
 If (Is_a_variable(y)) then return Unify_variable(y, x, mu)
 If (Is_a_compound(x)) and (is_a_compound(y)) then
 return Unify_internal( args(x), args(y), Unify_internal(op(x),op(y),mu )
 If (Is_a_list(x)) and (Is_a_list(y)) then
 return Unify_internal( tail(x), tail(y), Unify_internal(head(x),head(y),mu)
otherwise return Failure


Unify_variable(var, x, mu)
 If (a substitution value/var is in mu) then
 return Unify_internal(value, x, mu)
 If (a substitution value/x is in mu) then
 return Unify_internal(var, value, mu)
 If (var occurs anywhere in x) then return Failure
 Add x/var to mu and return
```

A *unifier* of two formulas P and Q is a substitution σ that makes

$$P\sigma = Q\sigma$$

A given pair of expressions may have several unifiers or none. The substitution σ is *more general* than ϕ if

$$\phi = \sigma \, o \, \theta$$

for some substitution θ and o is the composition operator. A substitution σ is the *most general unifier (MGU)* of terms $t_1, \ldots, t_k$ if:

- σ unifies $t_1, \ldots, t_k$ and
- σ is more general than every other unifier of $t_1, \ldots, t_k$

The practical implication of the notion of *MGU* is the following: by using the composition operation, the *MGU* can generate all unifiers of the terms. In general, unification algorithms focus on finding the *MGU* for a given set of formulas. MGU for a given set of formulas is unique, up to renaming.

4.7.2 Sequent rules for classical first-order logic

Sequent calculus is used in the implementation of the prototype first-order theorem prover used as the baseline system in the case-study discussed in chapter 7. §4.4.2 and §4.4.4 covered sequent calculus and backward proof. Table 4.3 provides an enumeration of the sequent rules for classical first-order logic without equality.

4.7.3 Meta variables

Scenarios involving quantifiers pose a challenge in terms of the appropriate substitution for the variables. In particular, consider the sequent rules $\forall : left$ and $\exists : right$ from Table 4.3. A successful instantiation of the term will be one that will ultimately generate subgoals and a successful proof. An application of the rules thus amounts to predicting one such candidate. However, this *prediction* is clearly not possible and a feasible solution is to postpone the commitment and apply a systematic process of trying out various possibilities along with other heuristics etc to synthesise value(s) for the candidate. [9]. This calls for a suitable device to capture this pending value and yet be able to continue with the rest of the proof. One such device is the introduction of **meta-variables**, which act as placeholders for terms which require their instantiation to be postponed/kept pending. In this thesis, we use the following notation to denote meta-variables: precede the variable with a question mark symbol, e.g. $?a_1$.

As will be discussed later in the thesis in chapter 7, implementation of the *unification* procedure for first-order logic described earlier, benefits from the notion of meta-variables.

4.8 Some relevant background on parallel computing

Parallel computing is a rapidly evolving field, both in terms of the machine architectures and the software and tools to use them. In this section, we provide an overview of some related concepts, which are of relevance to the material discussed in this thesis.

[9] This is analogous to the way mathematical reasoning works: try out multiple candidates for a particular variable that will allow for the proof to be finished or while trying to give a value to a variable which will fit the rest of the proof.

Connective	**:left**	**:right**
$\wedge$	$\dfrac{\phi, \psi, \Gamma \vdash \Delta}{\phi \wedge \psi, \Gamma \vdash \Delta}$	$\dfrac{\Gamma \vdash \Delta, \phi \qquad \Gamma \vdash \Delta, \psi}{\Gamma \vdash \Delta, \phi \wedge \psi}$
$\vee$	$\dfrac{\phi, \Gamma \vdash \Delta \qquad \psi, \Gamma \vdash \Delta}{\phi \vee \psi, \Gamma \vdash \Delta}$	$\dfrac{\Gamma \vdash \Delta, \phi, \psi}{\Gamma \vdash \Delta, \phi \vee \psi}$
$\rightarrow$	$\dfrac{\Gamma \vdash \Delta, \phi \qquad \psi, \Gamma \vdash \Delta}{\phi \rightarrow \psi, \Gamma \vdash \Delta}$	$\dfrac{\phi, \Gamma \vdash \Delta, \psi}{\Gamma \vdash \Delta, \phi \rightarrow \psi}$
$\leftrightarrow$	$\dfrac{\phi, \psi, \Gamma \vdash \Delta \qquad \Gamma \vdash \Delta, \phi, \psi}{\phi \leftrightarrow \psi, \Gamma \vdash \Delta}$	$\dfrac{\phi, \Gamma \vdash \Delta, \psi \qquad \psi, \Gamma \vdash \Delta, \phi}{\Gamma \vdash \Delta, \phi \leftrightarrow \psi}$
$\neg$	$\dfrac{\Gamma \vdash \Delta, \phi}{\neg\phi, \Gamma \vdash \Delta}$	$\dfrac{\phi, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, \neg\phi}$
$\forall$	$\dfrac{\phi[t/x], \forall x\, \phi, \Gamma \vdash \Delta}{\forall x\, \phi, \Gamma \vdash \Delta}$	$\dfrac{\Gamma \vdash \Delta, \phi}{\Gamma \vdash \Delta, \forall x\, \phi}$ proviso: x must not occur free in the conclusion
$\exists$	$\dfrac{\phi, \Gamma \vdash \Delta}{\exists x.\phi, \Gamma \vdash \Delta}$ proviso: x must not occur free in the conclusion	$\dfrac{\Gamma \vdash \Delta, \exists x.\phi, \phi[t/x]}{\Gamma \vdash \Delta, \exists x.\phi}$

Table 4.3: Sequent calculus rules for classical first-order logic without equality

Definitions of the terms used are provided in Appendix §A 1. [Andrews, 2000; Karp and Ramachandran, 1990] are good sources for further details on related background.

4.8.1 Relevant architecture categories and some emerging architectures

In this section, we enumerate some of the major hardware architectures, with an overview of where various emerging computing platforms fit in this taxonomy. An overview of related software tools that support programming for these architectures is also provided.

4.8.1.1 Classification of architectures

The conventional taxonomy for classifying parallel architectures is based on notions of *instruction* and *data stream*. However, today's processors have built-in parallelism in the way they execute instructions. The architecture of interest in this work is the Multiple Instruction Multiple Data (MIMD) category. MIMD systems are further classified into the following two categories. A feature-wise comparison of the two is given in Table 4.4.

Multi processor All processors have direct access to all memory

Multi computer a.k.a Distributed systems Each processor has its own local memory and access to non-local memory; remote access to memory requires the use of some form of *message-passing* mechanism

4.8.1.2 Emerging computing architectures

Some relevant emerging architectures are described below.

Multicore architectures contain two or more independent units(cores) that can read and execute instructions, all housed in the same physical unit. Shared-memory is the most common memory model though inter-core communication models are also used. The term *many core* is used when the number of cores is very high, typically in the order of a million cores.

Attribute	Multiprocessor	Distributed systems
Cost	Generally expensive: Involves advanced hardware implementations	Relatively less expensive as the main costs of setup are negligible
Ease of building	Faces limitations of the number of processors that can be part of a single multiprocessor unit	Easier to build: Individual computers can be added to an existing network with relative ease, thus keeping the costs low
Ease of extension	Number of processors is fixed	Easier to extend: Number of machines in a given network can be changed easily
Inter processor communication	Interconnection between the processors is fast	Some form of message passing has to be used, which in turn, may introduce a time overhead
Idle resource utilisation	Hard: It is hard for application software to employ load balancing strategies	Good support: E.g., application software routinely employ strategies to optimally utilise idle machines, during their down time for instance
Suitability for particular parallel implementations	Ideally suited for parallel implementations that utilise shared memory and/or heavy duty inter process communication and number of processing units are not too large	Ideally suited for parallel implementations that typically do not rely on shared memory and can operate with little inter process communication
Examples	Multicore machines	Clusters of workstations, grid

Table 4.4: Comparison of multiprocessor architectures and distributed systems

GPGPUs stands for General Purpose computation on Graphics Processing Units. It is also referred to as GPU Computing. Originally designed for high-performance graphics, GPUs are increasingly used as many-core processors, capable of supporting implementations with a high degree of parallelism. This has been aided by the availability of accessible development tools and interfaces. These are very well suited for operations like *stream processing*, i.e. to do the same job on a large data set, where the jobs themselves do not need to communicate with each other.

Distributed computing architectures came into existence with the advent of networks and thus has been around for a very long time. In this time, it has assumed many identities, some of which are described below. All of them use a group of computing elements (CEs), often called workstations and rely on the message-passing computational model rather than the shared-memory model. In addition, each variant has its own specifics that need to be catered to, as explained below:

Clusters A group of CEs that are interconnected by general purpose communication networks such as fast ethernet or other advanced forms of high-speed connections like a local area network.

Grid A distributed network of often heterogeneous CEs that communicate using the communication infrastructure of the Internet. Grid- like environments pose the following specific challenges in comparison to e.g., a cluster of locally connected CEs:

- higher message latency due to the widely distributed nature of the network and the reliance on relatively low-speed bandwidths
- limitations posed on the access to a CE; e.g. in most cases, due to security reasons, access is via a gateway machine and the operations that can be performed on a CE are limited to submitting a job, querying the status and retrieving the results; for the same reasons, communication between the CEs is not possible always
- resource limitations imposed by scenarios where a given CE may be scheduling multiple jobs
- higher likelihood of interruptions and hence higher degree of fault-tolerance is required

- inter-operability issues borne out of the heterogeneous nature of the network
- a CE might fail, making robust fault-tolerance capabilities, an important consideration

Cloud computing Cloud computing [10] is focussed on the *virtualisation* of applications, thus allowing for software to be provided as services running on huge commodity clusters. Cloud computing works on a *pay per use* basis and is operated by dedicated, special purpose, large and homogeneous data centres with virtualized services. The individual application developers/users can buy more processing power as and when needed.

Both clusters and grids provide enormous potential for idle-resource-utilisation strategies. These architectures are ideal deployment vehicles for distributed programming applications with different parts of a given application running on various nodes with load balancing strategies to make the most of the idle time of nodes and/or employing collaborative problem solving approaches and they are very well suited for algorithms adopting the message-passing computational model. Multiprocessor machines are better suited for algorithms that adopt a shared-memory computational model. GPGPUs are specifically targeted at data parallelism.

4.8.2 Computational models

Exploring the scope and efficacy of employing these new computing paradigms and architectures to address the challenges of an application domain entails effective addressing of a variety of issues. Some of these are: effective task decomposition, coordination mechanisms, resource allocation strategies, choice of computational model(s) etc.. Increasingly, no single computation model fits all the requirements of an application and quite often a combination of models are used. In this section, we describe two main computational models relevant to this thesis.

A *concurrent program* contains simultaneously executing threads that are orchestrated

[10] For more on cloud computing, the reader may want to read this url http://en.wikipedia.org/wiki/Cloud_computing

in predefined ways to perform a task. The threads can use inter-process communication and/or synchronisation to accomplish a task. Thus, *concurrent programming* encompasses programming for both multiprocessor and distributed systems. *Shared memory programming* and *distributed programming/message-passing based programming* refer to two specific ways of writing a concurrent program and are most commonly used in the context of multiprocessor and distributed memory models respectively.

Shared memory programming assumes that all the processes have access to all parts of the memory. Thus, conceptually they are very similar to sequential programming, except for the asynchronous nature of the processes. There are obvious factors to deal with, in the form of race conditions and synchronisation of memory, owing to possible scenarios of concurrent access of the same memory location.

On the other hand, *distributed programming* relies on message passing for its communication and is faced with the challenges of: message latency, heterogeneity of the architectures and/or operating systems of the individual workstations, optimal load-balancing strategies (to keep all the processing units as busy as possible).

An additional distinction that is sometimes made in the literature, particularly in the context of high-performance applications, is *parallel programs*. It is used to refer to a subset of concurrent programs that are specifically targeted at reducing the execution time, compared to the sequential counterpart. Both *shared memory programming* and *distributed programming* can be used to write parallel programs and is usually dictated by the target architecture. However, in this thesis, we use the term *parallel programs* with no particular specialised usage.

In the next section, we discuss some relevant parallelisation techniques.

4.8.3 On implementing parallelisation

Typically, the starting point for a parallel algorithm is to take a sequential algorithm and parallelise it using an appropriate parallelisation strategy. The process of choosing an appropriate strategy entails making choices about the following:

1. Number of processors to use

2. Using some form of decomposition of work for distribution across processors
3. Load-balancing strategies
4. Choosing the computational model in conjunction with all these factors and the associated overheads of communication and synchronisation

To address the decomposition aspect of a given algorithm, the following two approaches are generally adopted:

Functional decomposition/Task parallelism Splits the algorithm into more or less independent procedures that can be executed in parallel, essentially giving rise to a new algorithm which may share some similarity with the original algorithm.

Domain decomposition/data parallelism This focusses on the data set used in the execution of the algorithm, enabling concurrent processing of independent sets of input, internal or output data. The typical case of data parallelism is when the same operation is performed on a huge data set.

The rationale for picking an implementation choice for parallelisation is a function of the application domain, the particular target system, algorithms, target architecture(s) and the techniques of parallelisation employed. e.g.

Parallelisation technique Data parallelism, co-routining, hybrid approaches

Computational models Shared memory, distributed memory, hybrid memory models

Target architectures of deployment Clusters, grids, multicore machines, GPUs

Choices for implementation

In Table 4.5, we summarise some of the commonly used implementation approaches to incorporate parallelisation in shared and distributed memory models.

Language-based parallel programming

Use of API based approaches like openMP and MPI give access to parameters closer to the machine architecture. However, they do not offer abstraction, from the programming point of view and are notoriously hard to program and debug. The developer has

Option	Shared memory model	Distributed memory model
Smart compilers	Use a parallelising compiler that automatically converts a sequential program to a parallel one. This is very language dependent and has to be tuned to work for specific memory models	Not used
Use OS based resources	Processes, threads, semaphores	Sockets
Parallel libraries: Used by sequential languages	OpenMP	PVM, MPI
Parallel languages: Most cater to multiple programming models	Ada, Cilk, HPF, NESL, Java, C#, PolyML	Java, C#, Ada, Linda, Alice ML

Table 4.5: Commonly used programming approaches to incorporate parallelisation in shared and distributed memory models

to take care of all the low level details like lock synchronisation and thread scheduling. Architectures are changing, with greater internal concurrency (multi-core), better fine-grained concurrency control (threading, affinity), and more levels of memory hierarchy. This topic is addressed in greater detail in chapter 5.

Performance measurement

Speedup: is defined as the ratio of the CPU time of the sequential version and the parallel version

Efficiency: is the dual of speedup and is defined as the ratio of speedup and the number of processors; this gives rise to notions of *linear, sub-linear and super-linear* speedups, depending on efficiency being equal to 1.0, less than 1.0, greater than 1.0 respectively

Scalability Speedup and efficiency are relative measures and the empirical behaviour can fluctuate depending on the number of processors. Also, a program may display different behaviours with respect to speedup and efficiency depending on problem sizes. The notion of *scalability* tries to address these anomalies. A program is said to *scale* if the efficiency behaves consistently over a large range of values of the number of processors and problem sizes

Overhead: is defined as the ratio of the extra CPU time and the sequential CPU time, where the extra CPU time is the difference between the total CPU time of all the machines in the parallel version. This metric serves as a performance indicator taking into account time spent on communication, workload balancing, data structure creation/re-creation etc..

Sources of overheads The main sources of overhead in a concurrent/parallel program are:

- Process creation and scheduling
- Inter-process communication
- Synchronisation

All the aforementioned notions are bound to machine-level parameters. An alternative language-based performance measurement technique has been used informally in teaching and prototyping. This uses the following two measures (more abstract than

running time of the processors etc): *work* and *depth*: work is defined as the total number of operations executed by a computation, and depth is defined as the longest chain of sequential dependencies in the computation [Karp and Ramachandran, 1990]. However, this cannot account for locality issues and other overheads.

4.9 Summary

In this chapter, we provided background material useful for understanding material discussed in this thesis along with notations and terminology used in the thesis. Propositional logic and first-order logic were introduced with descriptions of their syntax, semantics and associated definitions. A brief introduction to theorem proving was provided and details were given for specific proof systems used later in the thesis. Bacckground material related to SAT solvers was discussed, covering the DPLL and Stalmarck's algorithm in detail, with an overview of key techniques used in optimised DPLL-based SAT solvers. The discussion of first-order theorem proving methods focused on sequent calculus, unification and meta-variables. Finally, some relevant material related to distributed programming was provided. A description of Alice ML, the implementation language used in this thesis and and programming abstractions is given in chapter 5.

Chapter 5

Why parallelise a theorem prover and how to do it

5.1 The free lunch is over

A much cited recent paper titled *The free lunch is over* [Sutter, 2005], provides some significant insights on why parallelisation is not just a choice, but is an imperative to enable performance gains in the future. The paper talks of how, until now, applications have been seeing performance gains, without any significant redesign, simply by virtue of the advances in hardware technology and how the performance lunch is not going to be free any more. With the physical limitations of processor speeds reaching saturation levels, parallel architectures are becoming the default choice to provide more computational power and engineering better applications is set to be accomplished in fundamentally different ways compared to the past. These emerging architectures come in varied forms from multicore machines to different kinds of distributed computing architectures.

Theorem proving with its inherently vast search spaces is facing challenges in terms of both problem size and complexity, fueled by the increasing range of applications that theorem provers are being used to tackle. Engineering better theorem provers with improved speed and/or improved success rates for both more complex and bigger problems, is thus a pressing need. With the imperative trends of parallelisation and the increasing ease of accessibility of emerging architectures and availability of

a wide variety of related software tools, it becomes more interesting now than ever before to investigate novel ways of using parallel technologies to identify and harness latent parallelisation, distribution and co-routining/collaboration opportunities present in theorem proving tasks. This need is echoed in a recent book [Kaufmann and Moore, 2009], where parallel, distributed and collaborative theorem proving is cited as being one of the key research problems for the future of automated theorem proving.

5.2 Parallelisation of theorem provers: for the diverse opportunities that it can open up

While the imperatives dictated by the limitations of processors and the concurrent/-parallel nature of emerging architectures is a strong motivation for investigating application of concurrent approaches to theorem proving, the use of these techniques can enable novel approaches to reasoning that are not possible in a sequential mode of execution. In this section, we describe some of these possibilities.

5.2.1 Enabling novel approaches

The origin and development of logic from early on, was motivated by the desire to understand reasoning. In fact, not just understand, but to be able to reduce reasoning to calculation/computation. The advent of computers facilitated the automation of the reasoning, paving the way for the field of *automated reasoning*. However, sequential computers have been used predominantly for building automated reasoners.

Use of parallel computer infrastructure to perform the automation of (the inference involved in) reasoning holds promise for implementing novel ways of *automated reasoning* and potentially to introduce *new* automated patterns of reasoning. Here are some examples, some of which we have used in the case studies discussed in this thesis (signposted in the material below):

Fastest first A fastest-first approach for performing a reasoning step E.g. if there are multiple OR-choices for the next inference step to be applied (where any one successful step suffices), a concurrent setting allows one to do the following: spawn threads for each choice and choose the one that returns first. We have

used this approach in the context of an LCF prover to introduce novel tacticals. This is described in detail in §7.6.2.

Asynchronous implementation of least-commitment strategies *Meta-variables* (see §4.7) are a standard technique used in theorem provers. A meta-variable is a special variable that acts as a device for implementing a *least commitment* strategy as follows: it stands for a pending choice whose instantiation is made later in the proof.

Scenarios using least commitment strategies like meta-variables are a good place to employ asynchronous modes of execution. Because, with the parallel model, one can now spawn an asynchronous process to find a suitable choice for the candidate of the *least commitment* technique, while carrying on with the rest of the computation.

Asynchronous implementation of proof and refutation steps Another possibility that one would not normally consider in a sequential model, is to spawn proof and refutation steps asynchronously. There could be other variations like spawning refutation step(s) for one or more part(s) of the proof (e.g., for proving/disproving a lemma).

Variables shared across multiple goals A parallel model opens up new possibilities for devising proof procedures for tackling scenarios where variables are shared between multiple goals: e.g., using message-passing and asynchronous execution. We have used this approach in the context of an LCF prover to use exchange of partially evaluated information across multiple goals. This is described in detail in §7.8.3.

5.2.2 Modeling of mathematical reasoning: automating the dynamics of proof discovery

Mathematical assistants refer to theorem provers applied to the mechanisation of mathematics and/or discovery of proofs. Compared to sequential algorithmic search-based *mathematical assistants*, a system that has been augmented with support for concurrency and parallelism, can open up novel opportunities for a radically different treatment to mimic the *dynamics* of proof discovery. In the next section, we explain the term *dynamics*, as used in this thesis.

5.2.2.1 The dynamics of proof discovery

There are not many accounts of how mathematicians actually discover proofs, as the published work invariably is a polished result and the details of the process of discovery are seldom documented. The publication, *How the proof of Baudet's conjecture was found* [Waerden, 1971], is a rare account of a mathematician's attempt to explain the process by which a proof was constructed.

It is an illustrative exposition of a phenomenon that is quite often encountered during the discovery of most proofs, whether it is by an individual or by a group of mathematicians. This account illustrates the *dynamics* of proof discovery, i.e. the interaction and communication between the different processes that happens in the course of the discovery of a proof by human mathematician(s): trial and error, proposal of an induction hypothesis, modification and learning from failure.

5.2.2.2 How to automate the dynamics of proof discovery?

One possible approach to model the interactive nature of the process of finding a proof is by using agent-oriented mechanisms incorporating notions of *utility functions* for proof processes (agents) and associated notions of *rational* approaches that try to maximise the utility etc ([Woolridge, 2001] is a good reference for background on agent based systems). As discussed in §2.2.3, the OANTS system [Benzmüller et al., 2008], uses an agent-oriented approach to implement a command suggestion mechanism for a tactic based LCF prover, with the possibility of using heterogeneous external provers.

Another possibility is to use approaches that draw inspiration from other fields which display similar *dynamics*, as seen in the TEAMWORK approach [Denzinger and Kronenburg, 1996], discussed in §2.3.1. It uses the dynamics involved in a hierarchical *team* setting, as observed e.g. in a typical (competitive) workplace: a hierarchy of experts, who are workers with *specialised* expertise, managers and supervisors. The dynamics involved is as follows: an iterative setup is provided and workers work asynchronously on pre-allocated tasks; a clear system of evaluation is performed by the supervisors and managers; this in turn,, is used to perform resource allocation for the workers for the next iteration.

Here is another possibility: we can try to mimic the *dynamics* at the inference level of

a proof, in the context of a single individual or a team of human beings working on a proof (similar to the dynamics in the process of discovery of the mathematical proof described above) [1]. With the various possibilities of inter-process communication and asynchronous execution, application of concurrency and parallelisation techniques to engineer theorem provers can offer a whole new set of possibilities for enabling such an approach to mimic the dynamics of a proof.

It can make mechanised mathematics assistants more powerful, by providing better facilities to mimic a human mathematician's reasoning. Furthermore, it can also allow for forms of reasoning that are not within the scope of human mathematicians. E.g., consider the possibility of executing an inference step that involves a million inter-related sub-steps. Now, consider the scope for potentially executing all the million inter-related sub-steps asynchronously and allowing them to communicate and share information and/or return the fastest computation, a task certainly beyond human capabilities. If mechanised mathematical assistants are provided, capable of performing such concurrent computations, it can lead to new possibilities for enabling a mechanised mathematical assistant for use beyond the role of a computational assistant.

5.3 Some choices for introducing and implementing concurrency and parallelisation techniques for the theorem proving domain

Above, we saw the importance of the application of concurrent programming techniques and the adoption of emerging architectures to engineer better theorem provers. We then provided some thought experiments on how these techniques can enable novel approaches to theorem proving that hitherto were not possible in a sequential mode of execution. In this section, we address the topic of implementation.

[1] Here, the term *dynamics*, refers to the interaction that happens between multiple proof steps/processes spanned in the course of finding a proof.

5.3.1 Object-level and developmental factors

Parallelisation of a theorem prover entails significant challenges along the following two dimensions and an effective redesign of theorem provers to incorporate concurrency and parallelisation has to address both these issues in an efficient manner:

Object-level: how to apply concurrent techniques to a theorem prover It is a non-trivial task to identify opportunities to apply concurrent and parallel techniques to a theorem prover and to make the right choices for implementing them. Some key questions are:

1. Where are the points in an algorithm/system with latent opportunities for effective employment of these techniques?
2. What form of parallelisation should be used: functional decomposition/-data decomposition?
3. What form of communication/task co-ordination should be used?
4. What are the overheads and tradeoffs?

These are in turn, influenced by the particular theorem proving system under consideration: the underlying logic, the proof system and the intended mode of operation. We refer to this strand of investigation as *object-level*.

As discussed later in §5.3.2, the theorem proving domain poses some specific challenges for parallelisation in terms of irregular solution spaces and shortage of uniform hardness criteria, which makes load balancing very difficult. The effective application of parallel techniques to the domain of theorem proving is still at a fairly nascent stage and it can thus stand to gain by more exploratory research involving an iterative process of experimentation at the algorithmic level and empirical analysis.

Developmental level: how to implement the concurrent techniques The experimentation phase referred to above can often be stifled by the difficulties of concurrent programming, which is notoriously error prone and difficult to program. This can prove to be a huge barrier for application of concurrent and parallel techniques to a domain, where, exploratory research is particularly needed. To this end, implementation techniques should ideally support the following:

- Rapid prototyping

- Ease of experimentation
- Portability
- Programmability and scope for incremental development

In addition to the criteria mentioned above, an approach that incorporates the use of effective high-level programming constructs that abstract the low level implementation details allows for **separation** of **design** and **implementation**. Such an approach allows the theorem prover designer to focus effectively on the exploration and experimentation aspects, in working towards synthesising novel proof search procedures, using concurrent and parallel techniques.

A detailed discussion of the use of high-level programming constructs, as an implementation approach is given in §5.4.1. We have adopted this approach in the case studies described later in the thesis. We have developed parallel/concurrent/distributed programming a bstractions, which we will henceforth refer to as *programming abstractions* for the theorem proving domain. These have all the advantages mentioned above and potential to be employed in other theorem proving scenarios other than those implemented in the work described here.

5.3.2 Issues to consider for effective parallelisation

In this section, we discuss important considerations for effective parallelisation, specific to the theorem proving domain.

Domain related challenges for theorem proving Parallelisation of theorem proving poses challenges that are different from other scientific computing domains, e.g., numerical computation, a domain that has seen widespread adoption of parallelisation approaches. Numerical algorithms possess a fair amount of regularities that can be exploited for the purpose of parallelisation. But, this is not the case with most symbolic algorithms found in theorem proving systems. Parallelisation of theorem proving entails a different set of challenges and requires different solutions, in many instances. Some of the challenging issues are:

- Irregularity of search spaces makes it hard to estimate the time needed for a computational step. A uniform characterisation of the difficulty of a subproblem is not always possible. This calls for a dynamic form of task

decomposition and interaction.

- Effective work partitioning is hard for most theorem proving domains, e.g. SAT solvers (see §2.1.11 for a discussion on this)
- Another related issue is that of performance variation. A small variation in a problem can potentially have drastic effects on its hardness and hence the time taken to compute it. This makes it very difficult to perform evaluation of the efficacy of a particular parallelisation approach/technique.
- Theorem proving problems come from a variety of domains and in turn, differ in their structure, difficulty levels and distribution of solution spaces. Thus, the utility of concurrent techniques can vary a lot depending on the problem class as well, apart from the theorem proving flavour under consideration. Thus, it is important for the user to have the flexibility to customise the suite of concurrent techniques, to a particular problem class. We have implemented one such approach for an LCF prover and have provided a suite of concurrent tacticals that allows the user to build on them to implement their own novel proof search procedures. This is described in detail in §7.2.1.
- The predominant flavour of parallelisation of theorem provers, particularly for the case of automatic theorem provers has been the use of some form of decomposition of work (see §4.8.3), for distribution across processors. However, parallelisation of theorem proving need not stop at being decomposition of one form or another. As we will see later in the thesis, there are many more useful ways in which concurrency, parallelisation and asynchronicity can be put to use. E.g., some theorem proving scenarios can benefit from the use of co-routining techniques, collaborative approaches and sharing of (partially evaluated) information, spanning multiple proof attempts or proof attempts of sub-goals for the same proof.

Efficiency criteria for algorithms: sequential vs parallel It is well known that designing better parallel algorithms requires a different set of considerations. It is not always the case that the primary criteria that make a sequential algorithm efficient necessarily carry forward to making a parallel algorithm efficient. E.g., a key criterion for an efficient sequential algorithm is the effective reuse of previously computed data and avoiding repetitions in computation. But, the priorities

are different for a parallel algorithm. Redundant computations are often performed to reduce communication costs and to effectively harness a huge array of machines/processors and similar considerations hold for space requirements as well [Steele, 2009].

In the light of this hugely important aspect, parallelisation of theorem proving needs development of novel parallel algorithms as well as reusing existing sequential algorithms and adopting ways of decomposing the computation.

Implementation choices Some of the key considerations are:

- The concurrent programming techniques and computational models to employ
- Choice of granularity, to apply these techniques on
- Target machine architecture, e.g., multi-core(shared memory), clusters (distributed memory) etc.
- Choices for implementation: use APIs, a functional programming language, an imperative language etc.
- Programmability: is the user going to be able to extend and further develop the concurrent techniques implemented? Do soundness criteria have to be considered for extensions?
- Use of APIs vs language-integrated parallelism: §4.8.3 gives some of the commonly used options for implementation of parallelisation. A quick glance at these reveals that the options to implement concurrent algorithms are spread along the spectrum of decreasing proximity to the machine level and operating system level resources and increasing ease of programming. This draws a not totally surprising parallel with the world of programming languages from the machine level languages to intermediate languages to higher level languages which can equally be placed on a similar spectrum of speed and access to machine level resources to ease of programming, portability and implementation

5.4 Parallelisation and programming abstractions

As mentioned earlier, parallel programming is notoriously difficult to program. It is error-prone, hard-to-debug and performance analysis is extremely difficult. From the point of a theorem prover developer, this is not a desirable situation. The developer's effort is better invested in the investigation of *how to apply* the new computational paradigms of concurrency, parallelism and distribution to their given application or algorithm rather than trying to deal with *how to implement* it. Thus, the need to *separate design* and *implementation* is of critical significance for effective adoption of these new paradigms of programming. This in turn, can enable novel algorithmic solutions that hitherto were infeasible in a sequential model of computation. In this work, we have used concurrent programming abstractions as a device to achieve this separation. They are described in detail in this section.

5.4.1 Abstractions: what are they and how are they useful

In the world of sequential programming, *design patterns* [Gamma et al., 2002], are used to capture recurring patterns of computation. A similar notion extended to the world of parallel programming, is provided by the notion of *algorithmic skeletons*, introduced in the book [Cole, 1991]. It is based on the observation that applications from diverse domains employ parallelism in the form of a few recurring patterns of computation and communication. Algorithmic skeletons are higher-order programming constructs that encapsulate these patterns with appropriate parametrisations.

For the purpose of this work, we use the terms *programming abstractions* and *abstractions* synonymously to refer to the following: capturing recurring patterns of computation, independent of an individual algorithm or program, as a *higher-order programming construct* with appropriate *parametrisations*. [2]

A simple example is the *task farm* skeleton, parametrised by: task-supply function (say, *f1*), task-doer-function (say, *f2*), data-location(s). This captures the following recurring pattern of computation: input data is generated (independently) by *f1*; *f2* works (independently) on the generated data.

[2] *Algorithmic skeletons*, as used in the parallel programming literature, includes various compiler translations and optimisations for the abstraction. We exclude these aspects in our usage and treat the abstraction aspect alone.

The utility of skeletons is two fold:

- From a software engineering perspective, it offers: modularity, ease of prototyping and development, code reuse and the potential for incremental development, facilitated by the *compositional* nature of the algorithmic skeletons
- With a focus on resource utilisation: it allows for engineering efficient APIs, tailored for particular parallel programming languages. Furthermore, skeletal programming advocates the following: the abstractions should transcend the architectural variations and architecture tuning should be handled at the implementation level [Cole, 2004].

Use of domain specific *programming abstractions*, for application of concurrent techniques has been advocated by leading experts in the field of concurrent programming as well [Asanovic et al., 2006] and has been adopted by many application domains.

As mentioned before, the speed at which the parallel computing architectures and paradigms are emerging further accentuates the need for an abstraction based approach, especially from an application point of view. Tying oneself down to a particular architecture or a particular implementation, can potentially make the work obsolete and extracting the crux of the implementation and porting it to another system may not be possible always.

In the context of an LCF style theorem prover, introducing parallelism and co-routining using programming abstractions, is particularly attractive as it is very much in tune with the essence of LCF approach of a trusted kernel of rules as the primitives with everything else built around it. The LCF style of theorem proving captures this separation very well compared to other schools of theorem proving.

To conclude this discussion, we provide an enumeration of the advantages of using *programming abstractions* to apply concurrent programming techniques to theorem proving:

- Allows for the separation of design and implementation
- Is independent of the target machine architecture
- Allows for portability to a wide range of platforms and languages
- Being higher-order functions, they can be composed and nested, thereby allowing for incremental design and development of richer and more sophisticated

abstractions

- Facilitate code reuse: scope for one abstraction to tackle multiple scenarios, via appropriate parametrisations
- Improves the clarity of design
- The modularity and reasoning power given by the abstractions make it easier to address issues related to formal notions of correctness

Using abstractions and high-level constructs comes at some cost to the developer in terms of losing control over the low-level (machine-level, OS-level) choices that could potentially be made. This is due to the significant abstraction gap between the design (high-level abstractions) and the implementation (low-level details). But, the benefits could potentially outweigh the costs for a domain like theorem proving and particularly so at the experimental stage, where dealing with low level APIs etc requires specification of too many details and can prove to be highly detrimental to the enterprise of experimentation. It can often obscure the meaning of the algorithm/technique being used as well.

5.4.2 Some concurrent/parallel programming abstractions

In this section, we describe some well-known programming abstractions that we have used in our case studies discussed in this thesis.

5.4.2.1 Producer-consumer

Producer-consumer is a commonly used parallelisation pattern and is commonly implemented using *streams*. A thread (the *producer*), puts data onto a stream (say *data stream*). The *consumer* threads read the data off the *data stream* and can read the datums off as and when they are generated. The code fragment given in Listing 5.1 is an illustration of the *producer-consumer* pattern.

Some of the advantages are: (i) to address scenarios where the data generation step is time consuming and/or unpredictable (ii) allows for *data parallelism*, by virtue of multiple consumers working on the data. This model can be used typically to replace iterative computations by using multiway data decomposition and aggregation of data (as opposed to dealing with singular decomposition and accumulation).

We discuss how we have used this to address particular theorem proving scenarios for SAT and LCF style first-order theorem proving in §6.5 and §7.6 respectively.

5.4.2.2 Pipeline

Pipeline is an abstraction that captures the following scenario: multiple computations need to be performed in sequence, with the output of one, serving as the input for the next computation. A simple example is the computation of the composition of multiple functions. It is an extension of the producer-consumer abstraction, to include more than two computational threads, with intermediate streams between any two threads. Each computation is performed in its own thread and has an input stream and an output stream. A discussion on how we have used this in our work on LCF style first-order theorem proving is given in §7.6.

5.4.2.3 Barrier

Barrier is an abstraction used to capture the following computational pattern that commonly occurs in many iterative algorithms. Typically, the same computation is performed on all elements (of the input), allowing for a simple multi-way decomposition with multiple threads working on each element. The key factor is that each thread cannot start its next iteration until all the others have completed the current iteration. This is due to the mutual dependency on the data computed by the concurrent threads in the current iteration. The computation times may be different for each of thread, as each of them is working on different data. Barriers are commonly used to capture this pattern of *forced waiting*.

For the set of the concurrent threads participating in a computation, a *barrier point* is defined in the algorithm. Upon reaching the barrier point, each thread has to wait until all other threads have reached the point. Different languages and APIs implement this abstraction in a variety of ways. For the purpose of this work, we use the term to refer to the computational pattern captured by it.

The field of numerical algorithms is an application class with many cases of barrier-like patterns, e.g. in algorithms computing better approximations to an answer. In theorem proving, a similar behaviour can be found in algorithms which use the *saturation* technique of performing an inference step iteratively until no more new inferences can be

found and where the $(n+1)^{th}$ iteration has to wait for the results of the n^{th} iteration to be fully computed, before it can commence its own computation. We have investigated the possibility of utilising a barrier-like computational pattern in the implementation of a novel concurrent algorithm for SAT that we have developed. This algorithm is particularly amenable to large scale parallelism. This is discussed in detail in §6.5.6.

5.4.2.4 MapReduce

MapReduce is an abstraction that has gained a lot of attention in recent research, partly because of it being championed in a big way by Google, which has developed its own implementation of the abstraction to run on its huge commodity clusters. For more details of Google's implementation, the interested reader is referred to [Dean and Ghemawat, 2004]. The abstraction sets out a specific programming pattern with the claim that many algorithms for generation and processing of large data sets can be re-cast to fit into the pattern, with appropriate parametrisations.

A high-level description of the abstraction is as follows

- A user-specified *map* function processes a key-value pair to generate an intermediate set of key-value pairs (an iterative operation)
- A *reduce* function groups the intermediate values by the key and merges them (an aggregation operation)

It is targeted at optimal utilisation of distributed clusters. The specific implementation (e.g. Google's implementation) takes care of the details of load balancing, fault-tolerance etc. Various APIs implementing the abstraction are available, with variations in the resource utilisation strategies employed and their implementations, as well as the architectures targeted.

Among other things, the popularity of the abstraction is due to the simplicity of the control structure, widespread availability of implementation APIs for a variety of languages and platforms, availability of technical infrastructure to deploy them as well as effective dissemination of the APIs promoted by organisations like Google.

The MapReduce abstraction has not been applied in the prototypes discussed in this thesis. But, a possible opportunity for its application in the concurrent Stalmarck's algorithm for SAT is discussed in §8.6.

5.5 Using the functional programming paradigm for implementation of and experimentation with concurrent/parallel techniques in theorem provers

In §5.5.1, we outline some features of the functional programming paradigm which make it a good choice for implementing concurrency/parallelism. In §5.5.2, we discuss some techniques for introducing some key concurrent/parallel programming devices to a functional programming model, for communication and synchronisation, while retaining the pro-parallelism factors of the functional programming paradigm.

In §5.6, we provide an overview of Alice ML, the implementation language for the case studies discussed later in this thesis. Alice ML is a functional programming language, augmented with support for concurrency and distribution and provides robust support for type-inference, in the distributed context as well. This discussion provides an illustration of a concrete instance of a programming language that implements the features discussed in §5.5.2. Furthermore, it also serves as an illustration of the desirable features of a concrete concurrent/distributed programming language that meets the criteria outlined in §5.3.1.

It is useful to draw the attention of the reader to the following: this discussion aims at general theorem proving as the target candidate, but is especially geared towards LCF style programmable provers. The work described in this thesis addresses only two case studies of SAT solvers and a first-order LCF style (programmable) prover. Thus, the entire spectrum of features discussed here, have not been put to full use in our experiments with these prototypes. Nevertheless, we believe that the description here serves a purpose of its own and provides the context for some of the work that we have outlined in the *future work* section, (see §8.6).

5.5.1 Advantages of using functional programming to implement concurrency

The advantages of functional programming are well known: easier to reason about, easier composition etc. It turns out that functional programming languages are a good substrate for implementing concurrency. Some of the key advantages of functional

programming languages are [3]:

- Immutable state
- Lack of side effects
- Referential transparency
- Allows for composition
- Ease of synchronisation, one of the biggest challenges of concurrent programming. Many imperative languages use *explicit synchronisation*, i.e. the mechanisms of synchronisation have to be completely handled by the programmer and requires careful use of locks, semaphores etc. One of the established techniques that circumvents the need to use these devices is that of *implicit data flow synchronisation* (explained in detail in §5.5.2.1). This technique fits naturally into the declarative concurrency paradigm and hence a functional programming language is well placed to support this.
- A functional programming language equipped with concurrency support, provides the perfect setting for development of *concurrent programming abstractions* as higher-order programming constructs that can be composed and reused.

Some functional languages that have tried to provide concurrency support are:

Erlang has been used in real-time telecommunications applications at the Ericsson laboratories, Sweden [Armstrong, 1997, 2007]. Its computational model treats processes as black boxes with message-passing as the sole form of communication. The emphasis is on robustness and fault-tolerance, driven by the target domain of real-time applications. However, it does not have support for type inference.

Haskell is a pure functional programming language and various libraries have been developed to provide support for parallel programming [Jones and Singh, 2008].

Scala integrates features of object-oriented languages and functional programming languages and uses static typing [Odersky, 2004].

F# provides language-integrated support for asynchronous functional programming with a focus on reactive event-driven programming [Syme et al., 2007].

[3] Some of these apply only for *pure* functional programming languages

OCaml is an established functional programming language [Leroy, 1996]. *OCamlMPI* [Leroy, 2003], an implementation of bindings for OCaml is available, based on the message-passing interface standard (MPI). MPI bindings allow for restricted forms of programming models. In particular, the multithreaded model is not possible with MPI bindings.

PolyML provides support via libraries for a small selection of asynchronous programming features like *futures*. The focus is to use multicore machines using native threads [Matthews, 2010]. It does not provide support for distribution.

Alice ML is a standard ML based language with support for concurrency and distribution [Rossberg et al., 2006]. It provides static typing while allowing for dynamic type checking of higher-order modules loaded at runtime. This is the implementation language used in this work and is described in detail in §5.6.

5.5.2 Language-integrated concurrency in a declarative setting

The term *declarative concurrency* is used to refer to a model of deterministic concurrency that is compatible with declarative programming. For a detailed discussion on this topic, the reader is referred to [Roy and Haridi, 2004]. A formal definition of the term, *declarative concurrency*, as given in [Roy and Haridi, 2004] is as follows:

A concurrent program is declarative if the following holds for all possible inputs. All executions with a given set of inputs have one of two results: (1) they all do not terminate or (2) they all eventually reach partial termination [4] *and give results that are logically equivalent (i.e. though the order of computation may be different, the end result is same)*

Enabling declarative concurrency in a language, by using libraries, can make it very cumbersome to use [5]. Declarative concurrency needs low-level support on the level of individual assignments and conditional checks. Provision of support for declarative concurrency by using libraries will require library calls to achieve each of these steps and to manage their interdependencies. The more natural solution is for the support

[4] A thread of execution is said to have *partially terminated* if it has not terminated completely yet. Further binding of inputs would cause it to execute further, up to the next partial termination, and will execute no further if no binding happens.

[5] A discussion on this with contributions by one of the authors of [Roy and Haridi, 2004] can be found here: http://lambda-the-ultimate.org/node/458

to be incorporated into the language definition and system, i.e. *language-integrated declarative concurrency*.

In the rest of this section, we discuss some key concurrent/parallel programming techniques and constructs, that can be introduced within a functional programming model, while still ensuring that the declarative aspects are retained.

5.5.2.1 Dataflow synchronisation

Some of the main challenges related to writing concurrent programs are: maintaining consistency of data across threads/processes, race-conditions, locks, synchronisation and shared state in data structures. *Synchronisation* is a fundamental concept in concurrent programming. When a thread needs the result of a computation done by another thread, it waits until the result is available, i.e. it *synchronises* on the availability of the result. Many imperative languages use *explicit synchronisation*, wherein the mechanisms of synchronisation have to be completely handled by the programmer. This requires skilful handling of various concurrent programming techniques like *locks*. This is one of the many reasons that concurrent programming is very difficult.

An alternative approach to handle synchronisation is referred to as *implicit synchronisation*. Here, the synchronisation operations are part of the operational semantics of the language.

Use of *dataflow variables* is one of the established techniques to implement implicit synchronisation. The motivation for dataflow variables is as follows: what happens if an operation tries to use a variable that is not yet bound? It would be nice if the operation would simply *wait*. Perhaps some other thread will bind the variable, and then the operation can continue. This behavior is known as *dataflow* and the consequent implicit synchronisation that happens is referred to as *dataflow synchronisation*. The variable in question is referred to as a *dataflow variable*. An unbound dataflow variable is said to have a *partial value*.

The following consequences of the *dataflow* behaviour are particularly well-suited to concurrent programming:

Incremental evaluation, a.k.a Data-driven evaluation allows for incremental evaluation, i.e. if the input is given incrementally, the program will compute the output incrementally. See Listing 5.1 for an example.

The code given in Listing 5.1 is a concurrent program. But, in a situation without dataflow variables, *list1* will need to be computed completely before the function *consumeInt* can even start. Given the time delay in this contrived example, the computation of *list1* takes at least 10,000s, before the first result gets printed. On the other hand, with the dataflow variable situation, the *consumeInt* function starts as soon as the first element becomes available (after 1000s, in this example).

If in the example, *list1* is a *stream* of data, then, we get a scenario where the call to *consumeInt* will never terminate completely, leading to what is referred to as *partial termination*. It will kick in every time further binding (of *list1*) happens, i.e. further elements start appearing in the stream *list1*. This feature of *partial termination* is a unique consequence of employing *dataflow variables* and facilitates *incremental evaluation*. We use the term *data-driven* evaluation synonymously to refer to this phenomenon.

Incremental evaluation vs Lazy-evaluation I.e. data-driven vs demand-driven evaluation: It is useful to observe here that while the above example share some similarities with *lazy* evaluation, a closer examination will highlight the following differences:

- lazy evaluation does a form of lock-step execution alternating between the producer and consumer
- it is demand-driven, rather than data-driven
- a producer cannot keep generating data unless the previous data have been consumed, in contrast to our example, where the producer can keep generating data, even if say, there is a delay in the computation of the consumer function

Order of execution does not matter The result of a program remains the same whether the program is executed concurrently or otherwise. E.g., if a program contains the following as concurrent computations (and hence without a deterministic order of execution/evaluation): a = b + 2 and b = 3, then with the dataflow variables scenario, the end result will be always same, as the order of execution does not matter.

Listing 5.1: Simple example illustrating incremental evaluation using dataflow variables

```
% Enumerate integers from low to high, giving a
% pause of 1000s in each iteration
fun produceInt low high = let
  do sleep 1000s
in
  if low > high then [] else low :: (produceInt low+1 high)
end

%Print the square of each element of the given list
fun consumeInt source = List.map (fn x => print x * x) source

%Spawn a thread to compute list1
val list1 = spawn (fn _ =>  produceInt 1 10);

%Spawn a thread to apply the function consumeInt to list1
do spawn consumeInt list1
```

5.5.3 Summary of advantages of dataflow variables and overview of how we have used it in our work

Here is a summary of some key advantages of the use of dataflow variables, in relation to concurrent/parallel programming:

- It is a powerful tool for enabling *implicit synchronisation* for concurrent programs
- It allows for static dependencies between different parts of a program (as specified by the code) to be replaced by dynamic (data-driven) dependencies, allowing for incremental evaluation and parallelisation
- It allows for the output of one part of the program to be passed as input to the next part, independent of the order in which the two parts are executed, as the in-built synchronisation takes care of the dependencies
- The same behaviour makes it a good device for distributed programming, where communication is handled across machines and issues like latency need to be taken into account. By virtue of the dataflow behaviour, *implicit* communication of the result of a computation happens
- It is very useful for addressing scenarios, where all the information needed for a computation is not available, by considering the end result as a complete value

with gaps (*unbound variables*) that need to be filled

We have used the powerful feature of incremental evaluation and the resulting data-driven behaviour in our case studies described later in the thesis in the following ways:

To implement waiting for work, in a work-partitioning scenario for SAT In the SAT case study, we describe the implementation of a novel concurrent algorithm for SAT that is amenable to large scale parallelism (§6.5). In this implementation, the work allocation mechanism is organised as a *data-driven* execution, thus allowing for effective work stealing without the costly overheads of communication to achieve work stealing that is often observed in the literature in other systems.

To implement asynchronous composition of tactics, for an LCF prover In the LCF prover case study, we have used the data-driven behaviour to implement a novel control structure for applying two tactics one after another[6]. The shortcomings of a sequential implementation of composition and how the data-driven behaviour helps to address them is described in detail in §7.6.1.

5.5.3.1 Language constructs for concurrent/parallel programming in the declarative model

A *thread* is an independently executing instruction sequence. If the language support for threads adheres to the dataflow principle, then all the benefits of dataflow synchronisation are carried over, paving the way for declarative concurrency based thread programming.

5.5.3.2 Message-passing and distribution mechanisms

The free lunch may be over (see §5.1), in terms of the memory speeds and what the architecture can offer, but certainly, there is scope for improvements in the network speeds, which are still steadily increasing. This has, in fact, paved the way for emerging paradigms such as cloud computing. As the network speeds go up and become more reliable, implementation techniques like message passing, remote procedure call (RPC) and related distributed memory models can provide a wide range of possibilities, in terms of parallelisation techniques. This extends to trends in supercomputers as

[6]When applied (to a proof state), a *tactic* returns a list of next-possible proof states.

well, which are increasingly adopting the road of connecting a massive array of CPUs with an extremely fast interconnect. Thus, the use of message-passing techniques for parallelisation of theorem proving deserves serious investigation.

The declarative concurrent model can be augmented with a message passing mechanism using *streams*. In concurrent programming parlance, a *stream* refers to a list with an unbounded tail. When used for message passing, streams are used to hold the messages and posting a message to a stream corresponds to extending the list by one element. Treating the tail as an unbounded dataflow variable enables us to include streams within the declarative concurrent model. Furthermore, streams allow for implementing asynchronous communication models, making the send and receive (read) actions independent of each other. In this work, we use the *channels* feature of the Alice ML library, to implement *streams* and use the two terms synonymously in the exposition.

Serialisation, also referred to as *marshalling*, refers to the process of converting a data structure into a format, such that it can be stored in memory and/or can be transmitted over a network, to be reassembled into the original data structure in a similar or different environment. It is a very useful feature in the context of message-passing based distributed systems, particularly in the context of the declarative model.

5.6 Alice ML

Alice ML [Rossberg, 2007] is a standard ML(SML) [Milner et al., 1997] like functional programming language with support for two seemingly contrasting features: dynamic exchange of higher-order values with other processes and strong static typing, thus enabling *type-safe distributed programming*. This is achieved by the provision of the following features:

Higher-order modules and dynamic type checking achieved with the aid of *packages*

Higher-order serialisation accomplished with the aid of *pickling*

Concurrency related features realised with the aid of *threads* and *futures*

Distribution support using *tickets, pickles, proxies*

In addition to the above features, Alice ML includes optional *lazy* evaluation which can be enforced on an expression by prefixing it with the keyword *lazy*. *Exceptions* are also included as part of the language definition. The declarative nature of the language and language-integrated support for concurrency and distribution make it an ideal vehicle for rapid-prototyping and experimentation. The suite of concurrency/distribution primitives facilitates the expression of programming abstractions in a concise way, allowing for code reuse, portability and incremental development, all highly desirable features for applying concurrent techniques to theorem proving, as discussed in §5.3.1. These factors assume special significance for LCF style programmable provers, as discussed later in the thesis in chapter 7.

In the rest of this section, we describe the features mentioned above as well as the support provided by Alice ML for the features discussed in §5.5.2. A consolidated listing of the relevant language constructs is provided in Appendix §A 2. A thorough discussion of the technical details related to type checking etc can be found in [Rossberg, 2007]. Another source of comprehensive information on the language specific features is the web page for the Alice ML manual,
http://www.ps.uni-saarland.de/alice/manual/sitemap.html.

5.6.1 Support for thread-based programming

Operating system threads are computationally expensive as they involve allocation/deallocation of system resources and stacks. For this reason, the use of language-based lightweight threads is highly recommended [von Behren et al., 2003]. Alice ML provides support for lightweight threads. Furthermore, creation of threads in Alice ML is relatively straightforward, a positive aspect, from the development perspective. Prefixing an expression with the keyword *spawn* results in the creation of a concurrent computation (*thread*), evaluating the expression.

The result of the computation thus spawned is a *future*, a placeholder for the result of the asynchronous computation that has been spawned. As soon as the thread terminates, its result globally replaces the *future*. Thus, the result of a thread's computation (a *future*) can be referred to, before the computation is complete and the operational semantics of *future* will implicitly take care of the synchronisation. Futures are explained in detail in §5.6.2.

Threads are treated as *first class values* in Alice ML. This allows for pending computations to be communicated over a network, allowing for effective distribution. This feature, along with Alice ML's robust support for dynamic typing for distribution, can be of great use for using distributed computing resources for theorem proving. E.g., in the context of theorem proving, the notion of *futures* can be used for implementing constructs like *holes* in proofs, which can be used to stand for a pending computation. The distribution support can be used for using grids and clusters to execute parts of the proof or to outsource a heavy-duty computation to a remote (powerful) server.

Alice ML supports lightweight threads. This enables thread-based programming, even on machines with modest resources. From a prototyping and experimentation point of view, it allows for multiple threads to be run on a single processor machine. The simple constructs provided for thread-based programming, along with the support for dataflow synchronisation make Alice ML a developer-friendly language for doing thread-based programming. This is of crucial importance to us, as theorem provers are complex systems and the development efforts required for introduction of concurrent techniques should not be too high as it can stifle ease of prototyping of complex techniques and experiments.

5.6.2 Synchronisation in Alice ML

In §5.5.2.1, we described the importance of synchronisation for concurrent programming and the options for implicit synchronisation facilitated in a functional setting. In this section, we describe how Alice ML supports these features.

5.6.2.1 Implicit synchronisation

Effective devices to implement synchronisation between threads is a fundamental necessity for concurrency support in a programming language. In §5.5.2, we saw the ad-

vantages of language-integrated *implicit synchronisation* using the dataflow behaviour. Alice ML uses the concept of *futures* to provide *implicit synchronisation* and *communication* between threads [7]. It is defined as follows: *A future is a transparent place-holder for an (as yet) undetermined value that allows for implicit synchronisation based on data flow.* Alice ML provides an additional language construct called *promises*, which is explained below. Alice ML offers four kinds of *futures*:

Concurrent future Place-holder for the result of an expression computed in its own thread. In functional programming terminology, it is a place-holder for the result of a concurrently evaluated expression. For the purpose of this work, we use the term *future*, to refer to *concurrent future* unless specified otherwise

Lazy future It is very similar to concurrent future, in being a place holder for the result of a concurrently evaluated expression. However, the computation is delayed until another thread actually requires its result. Thus, it is useful to model a demand-driven computation. In Alice ML, an expression can be made lazy by prefixing it with the keyword *lazy*

Promised future It is created through an explicit handle called a *promise*. A promised future is eliminated by fulfilling the associated promise through an explicit operation. Promises are akin to single-assignment variables or logic variables and allow for the construction of data structures with holes. *Promises* are created uninitialised, but may be assigned only once.

Failed future Replaces a future that could not be eliminated because the associated computation terminated with an *exception*. Whenever a failed future is accessed, the respective exception will be re-raised in the thread accessing it.

A thread might want to create a future without making a commitment to the way the information is obtained. *Promises* are useful for addressing such scenarios, as they separate the operations of creation and elimination of futures. A promise is an explicit handle for a future. A suitable value determining the future will be made available at some later point in time and this is done explicitly using the operation *fulfill*. While it is still a form of dataflow synchronisation, the key difference between promises and concurrent futures is the use of the operation *fulfill*. A corresponding *fail* operation is also provided, yielding a *failed future* carrying the corresponding exception.

[7] The original idea of *futures* has its origins in the parallel language, MULTILISP [Halstead, 1985]

Futures can be passed around as values. Once an operation actually requests the value that the *future* stands for, then the corresponding thread will block until the future has been determined. This serves as a powerful mechanism for high-level concurrent programming. It also allows for *lag tolerance*: the rest of the computation can continue while the result is being computed. In the context of LCF-style theorem provers, *futures* and *promises* can be used to spawn proof attempts of sub-goals in a concurrent manner. The implicit synchronisation will allow for the rest of the proof process to continue without having to wait for these proofs to be completed.

5.6.2.2 Explicit synchronisation

Alice ML provides support for *explicit synchronisation* using the following two constructs:

await It triggers the computation of the argument, waits until the computation has been completed and then returns the result.

```
val await : 'a --> 'a
```

---

awaitEither Implements *non-deterministic choice*: triggers computation of two futures and blocks until at least one has been determined. This simple primitive can be used to encode complex synchronisation with multiple events.

```
(* alt refers to the standard SML datatype;
datatype ('a,'b) alt = FST of 'a | SND of 'b *)

 val awaitEither : 'a * 'b --> ('a,'b) alt
```

5.6.3 Support for Stream-based programming

Alice ML provides the construct *channels*, a fully concurrent imperative abstraction for streams [8]. Also provided are associated operations to insert elements into and to take elements off the channel. A consumer takes elements available at the beginning of the channel and a producer inserts elements in the channel, either at the beginning (LIFO) or at the end (FIFO).

[8] *Streams* are used in concurrent programming to refer to a list with an unbounded tail.

In Alice ML, channels are *thread-safe*: many consumers and producers can operate concurrently on the same channel. However, channels contain implicit locks. Thus, stopping a thread while it is manipulating a channel, may cause all further access to the same channel to block, until the thread is restarted.

The elements of a given channel can be obtained as a list using the operations *toList* and *toListNB* (explained in Appendix §A 2). Both functions return a *lazy* list with the elements of the channel.The latter returns an empty list, if there are no elements in the channel and the former *waits*, till the channel gets populated. If the list is evaluated, then the current elements of the channel are emptied and form the elements of the list returned (lazy semantics), while the tail of the list still refers to the tail of the channel. Thus, a subsequent operation of insertion of an element to the channel results in an insertion of the element to the list when the list is evaluated (lazy semantics). A list can also be *cloned*. The cloning operation returns a new channel initialized with the elements of the given channel.

5.6.4 Support for distributed programming and message-passing

As mentioned earlier, Alice ML provides support for dynamic exchange of higher-order values with other processes and strong static typing, thus enabling *type-safe distributed programming*. A language that allows for encapsulation of modules as first-class values and allows for them to be exchanged over a network, ensuring type safety, is a very desirable choice for implementing a distributed theorem prover. e.g., it can open up an entire spectrum of potential opportunities of using richer message-passing techniques, where the messages can have higher-order content. The type-safety guarantees make it an ideal choice to use, to extend an LCF style prover with *sound* extensions incorporating concurrency, parallelism and distribution. In the rest of this section, we briefly describe the mechanisms used by Alice ML to enable distribution.

Pickling A generic mechanism for import and export of language-level data structures, including code. A pickle is a self-contained, platform-independent, external representation of an Alice ML value.

Proxy Remote procedure calls (RPCs) (see Appendix §A 1 for definition) are the main means of inter-process communication in Alice ML. A thread in an Alice ML process can call a function that actually resides in another process. To perform

RPCs, Alice ML employs the notion of a *proxy* function. A proxy is a (mobile) wrapper for a stationary function. It can be pickled and transferred to other processes, independent of the wrapped function. When a proxy is invoked/applied, the proxy is evaluated in the process that it was created in, irrespective of which process the proxy was invoked from (see [Rossberg et al., 2006],[Rossberg, 2007] for more information). Proxies help to address two key scenarios encountered in distributed programming:

Establishing/retrieving connections, Tickets Connections can be provided by *offering* a module containing proxies on the network. *Tickets* are URLs which are globally-unique and dynamically generated at the time of *offering* a module. These are used to retrieve a module, achieved using an operation called *take*. The ticket identifies the machine/process where the module is located. The module itself is wrapped in an Alice ML language construct called a *package*.

Remote execution Spawning processes remotely is achieved using the notion of *components* and the functions provided in the Alice ML library, *Remote*. *Components* are the units of compilation and deployment in Alice ML. The export of a component is a module expression that will be evaluated when executing the component. The *remote* library provides a function *run*, which enables remote execution and performs most of the low-level steps needed. It takes as arguments: the target machine name and the component. It connects to the given remote machine, using a low-level service like *ssh*. It then starts a fresh Alice ML process on the remote machine, as a *worker*. The worker immediately connects to the master (the machine that invoked the *run* function) to receive the component argument, and evaluates the component, giving a *package*, which is sent back.

Dynamic type checking The notion of *packages* was mentioned briefly above. This is the device that is used to perform dynamic type checking, in relation to distribution. A *package* is a value encapsulating an arbitrary (higher-order) module and its signature. It has two associated operations: *pack* and *unpack*. *Unpack*ing a package performs a dynamic type check. Thus, along with tickets, packages enable the realisation of distributed dynamic exchange of higher-order values with other processes and strong static typing, enabling *type-safe distributed programming*

Listing 5.2: Sample code for some concurrent programming abstractions in Alice ML

```
(*Higher-order barrier*)
fun barrier fs = map await (map (fn f . spawn f ()) fs)

(*Time-out*)
exception TimeOut
fun timeOut time f =
case awaitEither (f, spawn sleep time) of
| FST f => x
| SND {_} => raise TimeOut

(*Fastest-first: Returns the computation that complets first terminates the other*)
fun fastestFirst f1 f2 =
let
 val (t1,r1) = Thread.spawnThread f1 (*t1: Thread, r1: result, a future*)
 val (t2,r2) = Thread.spawnThread f2
in
 case (Future.awaitEither(r1,r2) ) of
   FST({_}) =>  (if (Thread.state(t2) <> Thread.TERMINATED) then Thread.terminate(t2);r1
       )
 | SND({_}) =>  (if (Thread.state(t1) <> Thread.TERMINATED) then Thread.terminate(t1);r2
      )
end
```

Network transparency A process can obtain references to values in another process (remote values), which are handled in (almost) the same way as local values. Hence the same abstraction mechanisms and idioms can be applied for local and remote operations and communication.

5.6.5 Ease of prototyping and developing abstractions in Alice ML

The language-integrated support for key concurrent programming primitives are expressive enough, to engineer concurrent programming abstractions, as higher-order functions. We provide code samples of some abstractions in Listing 5.2. As highlighted earlier, Alice ML provides support for *network transparency* in the context of distribution. Hence the same abstraction mechanisms and idioms can be applied for local and remote operations and communication. The utility of developing effective concurrent/distributed programming abstractions for theorem proving is highlighted in both the case studies discussed in this thesis. See chapter 6,chapter 7 for more details.

5.6.6 Suitability of Alice ML for implementing programmable parallel extensions for LCF-style provers

The following features of the **LCF** paradigm make it very well-placed to take advantage of the Alice ML features of type-safe distributed programming, implicit synchronisation and ease of developing abstractions:

LCF feature	**Alice ML feature**
Theorem as an abstract data type with restricted constructors and a trusted kernel	Implicit synchronisation, type-safe distributed programming
Modularity	Components
Programmability	The power of ML with support for concurrency and distribution

Table 5.1: Match between features of Alice ML and the LCF paradigm

5.6.7 Limitations of Alice ML

Not suited for Multi-core Alice ML uses a virtual machine constructed on top of the SEAM infrastructure (Simple Extensible Abstract Machine), a portable infrastructure for building virtual machines which implements generic services like memory management, thread management, pickling etc. [Rossberg, 2007]. SEAM and the Alice virtual machine have been implemented in C++, while the rest of the system is almost entirely bootstrapped in Alice ML. SEAM implements threads purely in software, using its own scheduling mechanism. It does not yet enable employment of system threads. Consequently, an Alice ML program cannot yet take advantage of multi-processor machines and multi-core processors. As the langage is not being actively developed any more, it is unclear if support for these features will be included, in the near future.

Overheads of distributed programming mechanisms In §5.6.4, we described how Alice ML supports type-safe distributed programming. However, our development experience shows that using these facilities comes at a significant cost due to the cloning and proxy operations performed at the various nodes of the dis-

tributed architecture. The tradeoff of using these facilities in relation to their utility needs to be considered for effective use of these techniques.

Non-deterministic thread scheduling For the same reasons mentioned above, thread scheduling is non-deterministic. The runnable threads are scheduled in a round-robin fashion. Thus, execution of priority mechanisms needs to be implemented via explicit coding.

Possible space leaks This is related to the garbage collection mechanism in Alice ML. Proxies represent a form of inter-process reference in Alice ML. Currently, a function for which a proxy has been constructed can never be collected, thus potentially creating a space leak.

Termination of child threads When a thread (that has spawned many other child threads) is terminated, the child threads are not terminated. We have addressed this problem by implementing, what we have called *hierarchical threads*. This inherits the Alice ML thread structure, but with facilities to handle termination of the child threads when the parent thread is terminated. This has been done by implementing bookkeeping to ensure that the parent thread's identifier is visible to the child threads and vice versa. See Listing 1 and Appendix §A 3.

Interactive top-level support Alice ML is an extension of Standard ML, and the Alice interactive top-level works in a similar fashion to those of other SML based systems. However, certain under-specified/unspecified features of Standard ML, like *use*, are not implemented. Thus, a system that has been written in some dialect of SML which assumes such implementations, faces these limitations, when being ported to Alice ML. We ran into one such limitation, in our efforts to port the theorem prover *Isabelle* [Nipkow et al., 2002] to Alice ML. More details about this are explained in §7.4.

Other incompatibilities An enumeration of incompatibilities with SML is maintained in the Alice ML project webpages. It needs to be added that most of these have easy workarounds as we discovered both in our efforts to port Isabelle to Alice ML as well as in porting a prototype first-order theorem prover (see chapter 7).

5.7 Summary

In this chapter, we briefly touched upon the imperatives of the hardware world driving the paradigm shift in the programming techniques used for engineering better applications and how these hold for the theorem proving domain too. We then set out an agenda for application of concurrent techniques to theorem proving (see §5.3.1). This agenda makes a distinction between the object-level focus and the developmental focus. A set of criteria for desirable implementation methodologies was provided. These criteria are geared towards enabling easy prototyping of and meaningful, non-trivial experimentation with the application of concurrent techniques to theorem proving. This in turn, can lead to the synthesis of effective novel proof search procedures incorporating concurrency and parallelism and enable optimal utilisation of emerging computing paradigms and novel computing architectures.

Also presented was an overview of the advantages of functional programming and some related concurrency features, in a declarative setting. Alice ML was presented as a concrete example of a real language that supports these concurrency features. Some possible theorem proving applications of the Alice ML features were alluded to, with references to details discussed later in this thesis.

Another topic that was discussed was the importance of developing effective programming abstractions (higher order programming constructs that capture concurrency patterns) for specific theorem proving scenarios, that can particularly be applied to address more than one theorem proving task. The use of abstractions ticks many boxes of the desirable criteria for implementation: portability, code-reuse, ease of programming, separation of design and implementation.

5.7.1 Conclusions and choice of case studies

The question of how to parallelise a theorem prover is too broad in scope, to tackle in a PhD project, given the particular challenges posed by the theorem proving domain. As seen in chapter 2, the *introduction* aspect in terms of the spectrum of techniques employed, the *implementation* and the empirical studies have all been vastly different across the flavours of theorem proving. Thus, the question certainly needs to be considered in the context of a given flavour of theorem proving: logic used, proof system used, mode of usage.

However, what certainly holds in all cases, is the need for an implementation methodology that will facilitate rapid prototyping of and experimentation with the application of concurrent/parallel techniques, facilitating the development of novel proof procedures and re-engineering of some existing proof procedures.

Some important considerations for the effective employment of the parallel paradigm to engineer better theorem provers are:

- Provision of frameworks that will allow for rapid prototyping and experimentation with and incremental development of novel parallelisation approaches
- There is a lot of similarity in the problem scenarios encountered and the algorithms used in different theorem provers. Thus, an effective implementation of parallelisation to tackle one scenario can be reused to tackle another similar scenario. Likewise, parallelisation approaches employed to improve/redesign an existing algorithm can be extended to another similar, if not identical algorithm/implementation. Thus, extracting these generic patterns can be extremely useful to facilitate portability, reuse and incremental development and it is desirable for implementation efforts to address these issues.
- The use of language-integrated parallelism offers a completely different set of possibilities for applying concurrent and co-routining approaches to theorem proving. Particularly, in the case of LCF style theorem proving, language-integrated parallel programming, as opposed to API-based parallel programming, allows for introduction of programming abstractions at the kernel level.

In the rest of the thesis, we discuss two specific case studies of theorem proving, where we have applied the object-level/developmental agenda set out here: (i) *SAT*, the propositional satisfiability problem (discussed in chapter 6) (ii) *HAL*, a prototypical LCF-style classical first-order prover without equality (discussed in chapter 7). These case studies were chosen to give a balanced view of the object-level possibilities in two disparate and representative flavours of theorem proving: (i) automatic, axiom-oriented style and (ii) interactive, human-reasoning oriented style. The developmental aspects have been effectively addressed and an enumeration of the abstractions developed and how they can possibly be reused are discussed in the respective chapters.

The *SAT* case study has identified opportunities for:

- Using two asynchronous communicating SAT solvers, each with a different ap-

proach to spanning the search space, with one learning from another, allowing it to possibly prune its search space.

- Recasting an existing recursive breadth-first search algorithm for SAT (the Stalmarck algorithm), giving a new algorithm that is more amenable to large-scale parallelisation

The *HAL* case-study has

- Showcased a multilayered approach to introducing concurrency/parallelism for LCF-style provers, focussing on programmability
- Give end users and theorem proving developers the opportunity to experiment with and develop novel proof search procedures as well use the primitives and abstractions to re-engineer existing search procedures.

Chapter 6

Novel approaches to SAT solving: lateral thinking, co-operation, concurrency and large scale parallelism

Given a propositional formula, the problem of finding whether there exists a variable assignment such that the formula evaluates to true is called the propositional satisfiability problem, often abbreviated as SAT. In this thesis, we have investigated the use of concurrent/distributed programming techniques for theorem proving, by considering two independent case studies of SAT and first-order theorem proving. SAT is the topic of discussion of this chapter. Relevant background material on propositional logic and SAT solvers were provided in §4.2.3 and §4.5 respectively.

Despite its NP-complete status, recent years have seen great advances in the development of new techniques and effective implementations for SAT. These advances have pushed the tractability threshold of SAT solvers in terms of size, hardness and complexity. However, the increasing suite of application domains present bigger and more complex problems and create a need for better SAT solvers that can handle the challenges of size and complexity, a phenomenon shared with the wider theorem proving world. As discussed in chapter 5, utilising emerging concurrent architectures and developing new ways of using concurrent/ distributed techniques to address these challenges for the domain of theorem proving, merits serious investigation.

6.1 About this case study

§2.1 provides a detailed review of published research related to the field of parallel SAT solvers. §2.1.11 distills this review and identifies some of the unexplored opportunities that merit investigation, in the context of applying concurrent approaches to engineering efficient SAT solvers, some of which are addressed in this case study.

As explained in §3.1 and §5.3.1, in this thesis, for each case study, we have explored the following two strands of investigation:

Object-level aspects: previously unexplored or little-explored ways of using concurrent/distributed techniques for the particular theorem proving flavour considered in the case study

Developmental aspects: developmental effort required, ease of prototyping and experimentation, scope for incremental development and portability

In this case study, addressing the object level strand of investigation, we discuss two *novel* ways of using concurrent/distributed programming techniques for SAT, using the DPLL [Davis et al., 1962] algorithm and the Stalmarck algorithm [Sheeran and Stalmarck, 1998, 2000].

DPLL: As described in §4.5.2.1, DPLL is a depth-first search based complete algorithm for SAT, used in many successful state-of-the-art sequential SAT solvers and many parallel SAT solvers are also based on the DPLL algorithm.

Stalmarck algorithm: As described in §4.5.3, the Stalmarck algorithm is a tautology checking algorithm. For the purpose of the prototypes developed in this project, we use the algorithm to compute *learned clauses* (described in §6.3.3.1). In the rest of this exposition, use of the term *Stalmarck algorithm* in the context of the hybrid solver, refers to this clause learner, unless specified otherwise.

While the DPLL method is a depth-first search approach, the Stalmarck algorithm can be interpreted as a breadth-first search approach, spanning all possible trees in increasing depth, with several enhancements.

An additional strength of this algorithm is its ability to leverage on the structure of the given propositional formula. Some of the key strengths of the method that have contributed to its success in the hardware domain and other industrial applications [Borälv, 1997] were enumerated in §4.6.

These novel concurrent approaches for SAT have been implemented in proof-of-concept prototypes, developed in Alice ML [Rossberg et al., 2006]. On a developmental level, programming abstractions encapsulating the concurrent techniques employed in the implementation, have been developed as higher-order functions in Alice ML. These can be ported to and/or used along with other SAT solvers.

Coarse granularity, DPLL-Stalmarck, a hybrid solver: In §6.3, we discuss the development of a novel co-operative hybrid approach to SAT. This combines the depth-first approach based DPLL algorithm and the breadth-first approach based Stalmarck algorithm, in an asynchronous setting. This allows for dynamic interaction and exchange of information, enabling dynamic pruning of search spaces.

multithreaded and distributed versions of this hybrid solver have been implemented. Empirical results show performance gains for the hybrid solver, compared to the stand-alone DPLL solver for two of the three problem classes considered. The behaviour of the third class was more random and non-uniform, but largely the hybrid solver was slower than the DPLL. In fact, the DPLL solver fared better without the CDCL. These are discussed in §6.7.1, with an analysis of the empirical behaviour.

An abstraction *dodpllWithHelper*, has been developed. This can be used to implement a DPLL solver with one or more external solvers that work asynchronously, acting as information providing *helpers* for the DPLL process. We have used *doDPLLwithHelper* to engineer two more hybrid solvers, DPLL-CDCL-Stalmarck and DPLL-ConcurrentStalmarck.

Fine granularity, Concurrent Stalmarck: In §6.5, we describe a novel algorithm that we have developed, by applying concurrent techniques to the Stalmarck algorithm. This is amenable to large scale parallelism. It provides an alternative approach to tackling task partitioning, different from the ones used by DPLL-based methods in the literature.

An abstraction has been developed to implement the *saturation* technique used in the Stalmarck algorithm (see §4.5.3). This abstraction uses the computational pattern captured by the standard *barrier* abstraction found in concurrent programming literature (see §5.4.2.3).

A novel form of work allocation has also been implemented using the power of *data-driven* evaluation. A proof-of-concept prototype of this new algorithm, has

been implemented in a multithreaded setting and early empirical results for the multithreaded version are provided.

At this point, it is worth drawing the attention of the reader to the following: the objectives of the investigation and prototypes discussed in this chapter have not been geared towards building an industry-standard SAT solver, but, rather, focuses on conducting exploratory investigations: identifying latent opportunities of applying concurrent techniques in novel ways, with a focus on the developmental aspects of using programming abstractions in a way that promotes portability and incremental development.

6.2 Implementation details for sequential SAT solvers based on DPLL and Stalmarck algorithm

In this section, we provide details of two independent sequential systems based on the DPLL and Stalmarck algorithms, implemented in Alice ML.

The code for these sequential solvers has been adapted from the SML versions of the same, found in the code repository accompanying a recent textbook on automated reasoning, entitled, *Handbook of Practical Logic and Automated Reasoning* [Harrison, 2009]. The code can be found in the following web pages: SML code for sequential DPLL and SML code for sequential Stalmarck tautology checker. The full Alice ML code for the sequential SAT solvers, based on the DPLL and Stalmarck algorithms, are provided in full in Appendix §A 4 and Appendix §A 5 respectively and include the relevant copyright notices. Brief, high-level descriptions of the data structure and the DPLL implementation are given in Listing 6.1, Listing 6.2 and Listing 6.3.

We ported the SML code to Alice ML and used the two sequential solvers as baseline systems to develop our parallel prototypes and to compare performances of the sequential and parallel versions. In particular, the Stalmarck solver provided by the above source is a tautology checker. We ported the code to Alice ML and did further modifications (described below) to engineer a clause-learning tool based on the Stalmarck algorithm, for use in DPLL-Stalmarck, our hybrid SAT solver.

In the rest of this section, we describe, in brief, some of the key features of the sequential versions, relevant for understanding the rest of the discussion. [1].

[1]For a more detailed presentation on the sequential implementations, the reader is referred to the

Some non-trivial features of the sequential Stalmarck implementation

In this section, we describe some non-trivial features of our Stalmarck implementation, in relation to the modifications that we have done in the ported Alice ML implementation, for use in our hybrid solver.

Equivalences An efficient key-value based data structure (using finite maps) is used for representing equivalences between formulas and this enables fast lookup, addition and deletion of equations [2]. Associated operations of insertion, equality are provided along with Stalmarck specific operations: checking for contradictions in an equivalence and intersection of equivalence classes

Trigger rules Trigger rules or simple rules that are used by Stalmarck's algorithm to derive equivalences between (sub)formuals. These are generated for a given formula, as a one-time operation.

Implementing zero-saturation As explained in §4.6, zero-saturation, one of the key components of the Stalmarck's algorithm is the exhaustive application of the trigger rules to derive new equivalences from existing ones.. This is implemented by the function *zero_saturate* in the original code. It takes an equivalence and a variable assignment as input and returns a new equivalence, augmented with the deductions derived as a result of the application of the simple rules. We have retained this implementation in our clause learner based on the Stalmarck algorithm and the concurrent Stalmarck prototype.

Implementing detection of contradiction Implemented by the function *truefalse*, which checks for the presence of a contradiction, i.e. an equation of the form $\top \equiv \bot$

k-saturation Uses two mutually recursive functions: *saturate* takes new assignments, 0-saturates to derive new information from them and repeatedly calls *splits* which in turn, splits over each variable in turn,, performing (k-1) saturations and intersecting the results

---

textbook [Harrison, 2009] which includes a detailed description of the Stalmarck algorithm as well.

[2] http://www.ps.uni-saarland.de/alice/manual/library/map.html

Listing 6.1: Code fragment for data structures used by sequential DPLL and Stalmarck solvers

```
datatype ('a)formula = False | True | Atom of 'a | Not of ('a) formula
| And of ('a)formula*('a)formula | Or of ('a) formula*('a)formula
| Imp of ('a)formula*('a)formula
```

Listing 6.2: Code fragment for an iterative implementation of the DPLL algorithm, using an explicit trail

```
datatype trailmix = Guessed | Deduced;; (*Explicit trail*)
fun backtrack trail= case trail of
 (p,Deduced)::tt => backtrack tt | _ => trail;
fun dpli cls trail=let val (cls',trail')=unit_propagate(cls,trail) in
if mem [] cls' then case (backtrack trail) of
 (p,Guessed)::tt => dpli cls ((negate p,Deduced)::tt) | _=>false
else case (unassigned cls trail') of [] => true |ps =>let
 val p=maximize (posneg_count cls') ps in dpli cls ((p,Guessed)::trail') end
end
fun dplisat fm = dpli (defcnfs fm) []; fun dplitaut fm = not(dplisat(Not fm));
```

Listing 6.3: Code fragment for iterative implementation of the DPLL algorithm, with non-chronological backjumping and learning

```
fun backjump cls p trail=case (backtrack trail) of (q,Guessed)::tt=>let
val (cls',trail') = unit_propagate (cls,(p,Guessed)::tt) in
if mem [] cls' then backjump cls p tt else trail end | _ => trail;

fun dplb cls trail=let val (cls',trail')=unit_propagate (cls,trail) in
 if mem [] cls' then  case (backtrack trail) of
 (p,Guessed)::tt => let
 val trail'=backjump cls p tt;  val declits=List.filter (fn (_,d)=>d=Guessed)trail';
 val conflict=insert (negate p) (smap (negate o fst) declits ord_forms) ord_forms
  in  dplb (conflict::cls) ((negate p,Deduced)::trail')  end
| _ => false
else case (unassigned cls trail') of  [] => true
|ps=> let val p=maximize(posneg_count cls') ps in dplb cls ((p,Guessed)::trail') end
end;
fun dplbsat fm = dplb (defcnfs fm) [];fun dplbtaut fm = not(dplbsat(Not fm));
```

6.3 Hybrid SAT solver: DPLL-Stalmarck

As discussed earlier, there is a need for exploring non-DPLL algorithms, so as to:

- address the limitations posed by DPLL solvers;
- explore the use of other complementary algorithms, alongside DPLL solvers;
- enable knowledge sharing between complementary approaches.

We have explored these possibilities by engineering a hybrid solver, by combining the Stalmarck algorithm with the DPLL algorithm. The rest of this section describes this hybrid solver, DPLL-Stalmarck.

6.3.1 Why combine DPLL and Stalmarck ?

As explained in §4.6, the Stalmarck algorithm has many distinguishing features, which make it a good candidate to be used along with the DPLL algorithm. A hybrid SAT solver that combines the breadth-first approach of the Stalmarck algorithm with the depth-first approach of DPLL in a co-operative manner, can enable the solver to span the search space in two different ways and will endow the hybrid solver with multiple, complementary viewpoints of the same problem (lateral thinking!). Furthermore, as the Stalmarck algorithm leverages on the structure of a given formula (see §4.5.3), it can help to offset the loss of implicit structural information, suffered by DPLL-based solvers.

6.3.2 How to combine the two ?

In our prototype of the hybrid solver, we have combined the two solvers in an *asynchronous* computational model. This allows for dynamic sharing of information and is well placed to prune the search spaces of the DPLL solver in a dynamic manner. Furthermore, the two solvers can work concurrently and independently on the problem, as autonomous, asynchronous computational processes. They communicate only when there is information to be shared, thus avoiding bottlenecks as well as being able to make the most of distributed architectures. The whole setup works in a co-operative manner by sharing the information found (which is one-way, from Stalmarck to DPLL, in our current implementation).

6.3.3 Implementation

In this section, we describe the implementation of the hybrid solver, DPLL-Stalmarck. §6.3.3.1 describes the Stalmarck-algorithm-based clause-learning tool that we have developed.

6.3.3.1 Using the Stalmarck algorithm, as a clause-learning tool

As described in §4.5.3, in the original Stalmarck (tautology checking) algorithm, after transforming the given formula to triplets, $v_i \equiv \bot$ is taken as an initial assumption, where v_i, a literal, stands for the entire formula. Using this as a starting point, the algorithm derives the consequences using the dilemma rule, zero-saturation and the saturation procedure; Obviously, if the given formula is a tautology, a contradiction will be derived as one of of consequences.

At this point, it is useful to observe that the key building blocks of the original Stalmarck algorithm of (i) equivalence relations between the formulas (ii) the dilemma rule (iii) zero-saturation and the k-saturation procedures are independent of the tautology checking in itself. In fact, for a given formula, the saturation Stalmarck algorithm can be used to derive the consequences, for a given list of assumptions.

To use the Stalmarck algorithm as a clause-learning tool for SAT, we use $v_i \equiv True$ as the initial assumption, where v_i, a literal, stands for the entire formula and derive the consequences, which are in the form of equivalences between (sub)formulas, of the form:

$$p \equiv q \text{ i.e. } p \leftrightarrow q,$$

where p, q can be any of the following: literal, sub-formula, $\top$, $\bot$.

We have implemented this modification to the original Stalmarck algorithm and use the modified version as an engine that generates the consequences, as mentioned above. The consequences are converted to clausal form and constitute the *learned clauses*, for our purpose. Given that formula structure plays a pivotal role in the derivation of these consequences, these learned clauses also stand to benefit from the same. In our implementation, $p \equiv q$ is converted to clausal form. This modified algorithm is used as a clause-learning mechanism and has been combined with the DPLL algorithm, in our hybrid solver, DPLL-Stalmarck.

If a contradiction is derived as a consequence by the Stalmarck algorithm, with the initial assumption of taking the original formula to be True, then it means that the original formula is UNSAT. This will get detected by the DPLL algorithm as the contradiction will be passed as an empty clause as part of the learned-clauses, to the DPLL algorithm, which will subsequently render the problem to be UNSAT. Furthermore, the Stalmarck algorithm can derive many consequences in one iteration. This further adds to the power of using this as a clause-learning mechanism.

6.3.3.2 Interaction between DPLL and Stalmarck

A high level description of the implementation of the hybrid solver, DPLL-Stalmarck, is given in Listing 6.4[3]. In our implementation of DPLL-Stalmarck, the main process is the sequential DPLL algorithm, that computes the final answer. The Stalmarck algorithm based solver is used in its clause learning form (as described above) and works as an independent process working on the same problem and supplies the learned clauses to the DPLL process. It thus acts as a *helper*, and supplies information to the DPLL process.

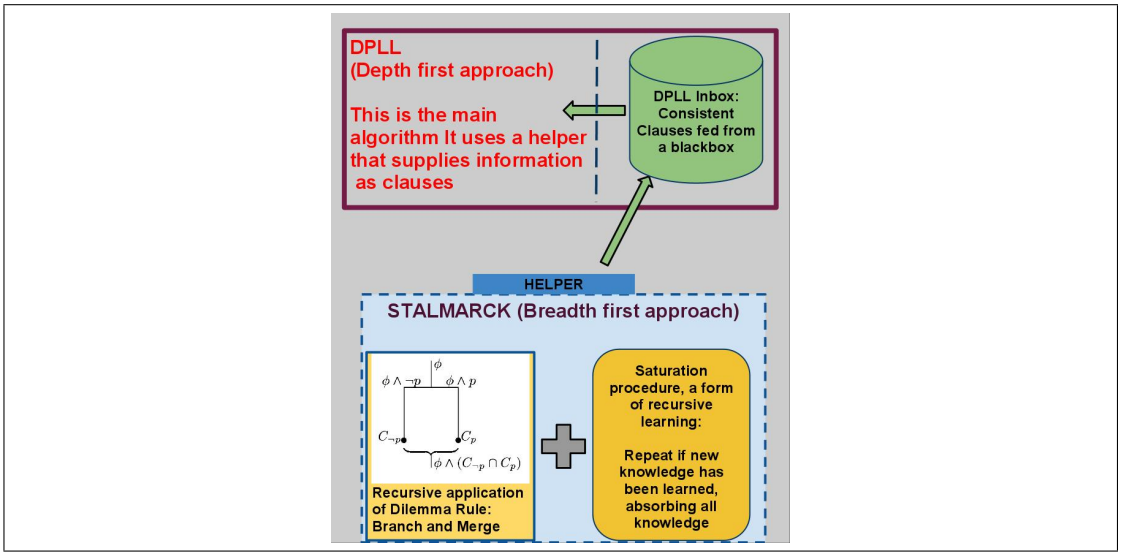


Figure 6.1: High level interaction diagram for DPLL-Stalmarck

[3]For DPLL-Stalmarck, *helper* should be interpreted as one or more Stalmarck processes.

Listing 6.4: High level design of implementation of DPLL solver with helper

```
type inboxElt  = prop formula list (*Clause: represented as a list of prop formulas*)

fun bootstrapHelper putTkt getTkt helperFn fm helperTime = tempHelperFun putTkt getTkt
     helperTime fm;

fun makeInboxAndGetAccessHandles () = let
 val (dpllInbox :  inboxElt  Channel.channel) = Channel.channel ();
(* Function to insert a list of clauses to dpllInbox *)
 fun dpllInboxPut1 eltList = List.app (fn y => Channel.put(dpllInbox ,y)) eltList;
(* Function to get elements from dpllInbox*)
 fun dpllInboxGet1()=let val tempCh=Channel.clone dpllInbox in Channel.toListNB tempCh
      end
(* Allow for remote invocation of the above functions *)
 val dpllInboxPutPack = pack (val dpllInboxPut = Remote.proxy dpllInboxPut1)
  : (val dpllInboxPut : inboxElt list  -> unit)
 val dpllPutTkt = Remote.offer dpllInboxPutPack;
 val dpllInboxGetPack = pack (val dpllInboxGet = Remote.proxy dpllInboxGet1)
  : (val dpllInboxGet : unit -> inboxElt list )
 val dpllGetTkt = Remote.offer dpllInboxGetPack;
in
 (dpllInbox, dpllInboxPut1, dpllInboxGet1, dpllPutTkt, dpllGetTkt)
end

fun doDPLLwithHelper helperFun helperTime fm = let
(* make the local dpllInbox channel; Any external agent (e.g., stalmarck agent) can
   post to this, as long as they know the appropriate ticket*)
 val (dpllInbox, dpllInboxPut1, dpllInboxGet1, dpllPutTkt, dpllGetTkt)=
      makeInboxAndGetAccessHandles()
 val thrHandle =  bootstrapHelper dpllPutTkt dpllGetTkt  helperFun fm helperTime

(* ------------------------------ *) (*DPLL*)
 val cls = defcnfs fm;  (*Converting to CNF*) val trail = [] (*Initial value*)
 fun dpli_stal_main cls trail = let
   val clsListFromInbox = dpllInboxGet1 (); (*Get clauses from Inbox*)
   val relClsFromInbox = dropDuplicatesFromClsList
                (List.filter (isClRelevant (varsInListOfClauses cls) ) clsListFromInbox)
   val cls = List.@(cls,relClsFromInbox) (*Add relevant Inbox clauses to problem*)
   val (cls',trail') = unit_propagate (cls,trail)
 in
  if mem [] cls' then case (backtrack trail) of
  (p,Guessed)::tt => dpli_stal_main cls ((negate p,Deduced)::tt) | _ => false
  else  case (unassigned cls trail') of  [] => true
 |ps=>let val p=maximize (posneg_count cls') ps in dpli_stal_main cls ((p,Guessed)::trail
      ') end
 end
(* ------------------------------ *) (*DPLL*)
 val res =  dpli_stal_main cls trail;do wrapUpHelper thrHandle;
in
 res
end
```

Some key features of the DPLL-Stalmarck solver are as follows :

dpllInbox A *dpllInbox* is created, using Alice ML's *Channels*, an abstraction for an unbounded list. Using the Alice ML's library functions of remote execution, the *tickets* to remotely access *dpllInbox* are provided (get, put operations: *dpllInboxGet* and *dpllInboxPut*). See §5.6 and §A 2 for explanation of Alice ML related terminology.

Bootstrapping, spawning of the helper process: Bootstrapping of the helper function is executed. The *helper* is spawned as an independent process. In the multithreaded version, this is executed in its own thread. In the distributed version, the helper is executed in a different machine. The *dpllInbox* access functions (or the respective *tickets*, where the helper is a remote process) are passed to the *helper* process . Different options to execute the helper are provided, as given below. These options are provided mainly for experimentation.

- Fully asynchronous: a helper is spawned and the execution of the rest of the solver is continued. In our implementation, this is achieved by giving -1 as the time for the helper.
- Compositional: a helper is spawned with the given time and posts its results to dpllInbox when the time is over. The main thread of execution of DPLL proceeds after this. By increasing the time parameter appropriately, this helps to address the scenarios where the helper is either too slow or a particular problem (class) is too difficult for the helper.

The helper posts the information as clauses. It can post any clause, because as described below, the DPLL process takes care of filtering out the relevant clauses from the contents of *dpllInbox*.

Dynamic pruning of the DPLL search space, using the helper info At every branch point, before descending in to the branch, the DPLL algorithm looks up the contents of *dpllInbox* and adds the *relevant* clauses from *dpllInbox* to its current problem. Here, we use the term *relevant* clause to refer to a clause which shares some variable with the current problem, i.e., the problem state at that branch point. We do this, because, adding other clauses does not help to reduce the problem.

The addition of the relevant learned clauses can potentially prune search spaces,

when unit propagation is carried out subsequently. Also, *dpllInbox* can be populated at any point during the execution of DPLL algorithm, thus allowing for the helper clauses to be posted as they are produced, enabling *dynamic* interaction between the DPLL process and the helper.

6.3.3.3 Other key features of the hybrid solver

Computational model: *dpllInbox*,which holds the information from the helper, does not share any memory with the DPLL process. Thus, it can potentially reside in a different OS process and possibly even in a remote machine and the DPLL process too can access it using the appropriate tickets, just as the helper process does. However, we have chosen to have it within the DPLL process (though it still does not share any memory with the DPLL process) for the following reason: Remote lookups are expensive in terms of computational time and the DPLL process performs the operation of lookup of the *dpllInbox* at every case split. Our implementation uses an *asynchronous message passing* model. The helper process posts the information to the *dpllInbox*, but neither process waits for the other's actions.

Performance and overheads: The hybrid implementation does incur some overheads in terms of the *dpllInbox* setup, lookups and associated processing. But, as mentioned above, the information from the helper can potentially prune the search spaces. Particularly so, since the information is being populated on-the-fly by the asynchronous helper(s), the DPLL process gets a chance to use possibly *new* information at each step. The speed at which the helper generates and posts the information to the *dpllInbox* is also crucial for the performance.

The utility of the information from a helper and the tradeoffs of the utility vs overheads is a topic that needs to be investigated more closely. A rigorous analysis of the same, possibly matching problems with a helper (as done in portfolio methods in SAT e.g. [Hamadi et al., 2009]) can greatly benefit the implementation. This is a possible option for future work.

Programming abstraction: Listing 6.5 gives the code fragment for *doDPLLwithHelper*, the programming abstraction that we have developed for the implementation of a DPLL solver with a helper. The full code is given in Appendix 5.

The abstraction is parametrised by the following:

- Dpll solver of choice
- Helper function of choice
- Time parameter for helper
- Type of learned clauses supplied by the helper
- Functions to bootstrap and wrap up the helper

Listing 6.5: Programming abstraction of DPLL solver with helper

```
fun doDPLLwithHelper dpllSolver inboxEltType bootstrapHelper wrapUpHelper helperFun
      helperTime fm = let
   (*make the local dpllInbox channel; Any external agent (e.g., stalmarck agent)
   can post to this, as long as they know the appropriate ticket*)
 val (dpllInbox, dpllInboxPut1, dpllInboxGet1, dpllPutTkt, dpllGetTkt)=
      makeInboxAndGetAccessHandles()
 val thrHandle =  bootstrapHelper dpllPutTkt dpllGetTkt  helperFun fm helperTime
 (* ++++++++++++ DPLL +++++++++++++++ *)
 val cls = defcnfs fm;  (*Converting to CNF*) val trail = [] (*Initial value*)
 val res =  dpllSolver dpllInboxGet1 cls trail;do wrapUpHelper thrHandle;
in  res  end
```

In tune with the motivation of the development of an abstraction, this allows for any helper to be used alongside the DPLL process.

- The only information that the helper needs is the problem and a handle to access *dpllInbox*.
- The helper is an independent process and is not dependent on the execution of the DPLL process. Depending on the user's preference, it can either be run asynchronously or run for a predefined time, before the execution of the DPLL process begins.
- The abstraction allows for a plug-and-play style of experimenting with different helper implementations. Even the user can become a helper agent by populating *dpllInbox* with information. This may or may not be useful in practical situations depending on the problem class considered, but, nevertheless, illustrates the potential of the abstraction and the ease of prototyping by adopting such an approach.

Multiple helpers As described earlier, the *doDPLLwithHelper* abstraction makes our

implementation generic enough to incorporate any helper, as any process can post information to *dpllInbox* as long as it knows the appropriate tickets. Thus, the implementation allows for multiple helpers, which can possibly be based on different algorithms as well.

6.4 Hybrid SAT solvers: DPLL-CDCL-Stalmarck, DPLL-ConcurrentStalmarck

As mentioned earlier, the *doDPLLwithHelper* abstraction allows for quick prototyping of a DPLL solver with a helper, with minimal developmental effort. To illustrate of the utility of this abstraction, we have used it to engineer two new hybrid solvers. These are explained in this section.

6.4.1 DPLL-CDCL-Stalmarck

The DPLL solver used in our hybrid solver, DPLL-Stalmarck, does not use the CDCL clause learning technique. CDCL has been widely adopted in most modern SAT solvers which are based on the DPLL algorithm. It can be useful to combine the power of CDCL and Stalmarck, by engineering a hybrid solver which uses the Stalmarck clause learner as a helper (as in the DPLL-Stalmarck architecture), and uses the DPLL algorithm, augmented with CDCL, as the main solver. To this end, we have engineered a new hybrid solver, *DPLL-CDCL-Stalmarck*. Empirical results for this are provided in §6.7. This solver has been developed by using the abstraction *doDPLLwithHelper* (described in §6.3.3.3), with the following parametrisation:

Dpll solver of choice: a sequential DPLL sat solver, augmented with CDCL, (see Listing 6.3 for a high-level design of this solver)

Helper function of choice: As in DPLL-Stalmarck

Time parameter for helper: As in DPLL-Stalmarck

Type of learned clauses supplied by the helper: As in DPLL-Stalmarck

Functions to bootstrap and wrap up the helper: As in DPLL-Stalmarck

6.4.2 DPLL-ConcurrentStalmarck

This solver has been developed by using the abstraction *doDPLLwithHelper* (described in §6.3.3.3), with multiple helpers, as explained below.

Concurrent Stalmarck is a piece of exploratory research approach that we have developed by applying concurrent techniques to the Stalmarck algorithm and is described in detail, later, in §6.5. The concurrent Stalmarck implementation uses multiple processes to tackle the problem. Each of these processes is independent of the others, works on the same problem and can generate learned clauses on its own. These learned clauses can be used by the DPLL solver, in the same way as the ones from the Stalmarck clause learner. Thus, each process can be used as a *helper* for the DPLL process.

In the case of using the concurrent Stalmarck algorithm as a helper, the bootstrapping stage involves: posting the problem to a pre-defined location; posting the units of work (combinations) to a predefined location and triggering the user specified number of agent services which are already running on remote hosts. A diagram describing the high-level design of DPLL-ConcurrentStalmarck is given in DPLL-ConcurrStal-interactionDiagram.

6.5 New concurrent algorithm for SAT, based on the Stalmarck algorithm

As mentioned in §6.4, as a piece of exploratory research, we have developed *Concurrent Stalmarck*, a novel algorithm applying concurrent techniques to the Stalmarck algorithm. The novel algorithm is amenable to large-scale parallelism and has allowed us to employ a producer-consumer approach and thus is well placed for optimal utilisation of bulk parallel processing resources. We have implemented an abstraction, which implements the *saturation* technique, a key component of the Stalmarck algorithm (see §4.5.3).

6.5.1 Gist of our approach

As described in §4.5.3, Stalmarck's algorithm uses the recursive *saturation* procedure (see Listing 4.3), which in turn, uses the *0-saturation* (see Figure 4.3) and the *branch-merge* rule (see Figure 4.1). As described in Listing 4.3, *saturate(P,k+1)*, performs a recursive application of the procedure, with 0-saturation serving as base-case for the recursion.

The key insight for the design of our new algorithm has been the fact that the recursive applications of the branch-merge rule can be flattened, as the operations are *associative* and thus independent of the order of execution. However, in a sequential setting, application of the *saturation* technique involves waiting for the completion of the computation of all candidates being considered in an iteration, before deciding to perform the next iteration.

This pattern of computation is similar to the *barrier* pattern found in the concurrent programming literature (see §5.4.2.3). We have implemented the application of the *saturation* technique as a programming abstraction, similar to the *barrier* abstraction.

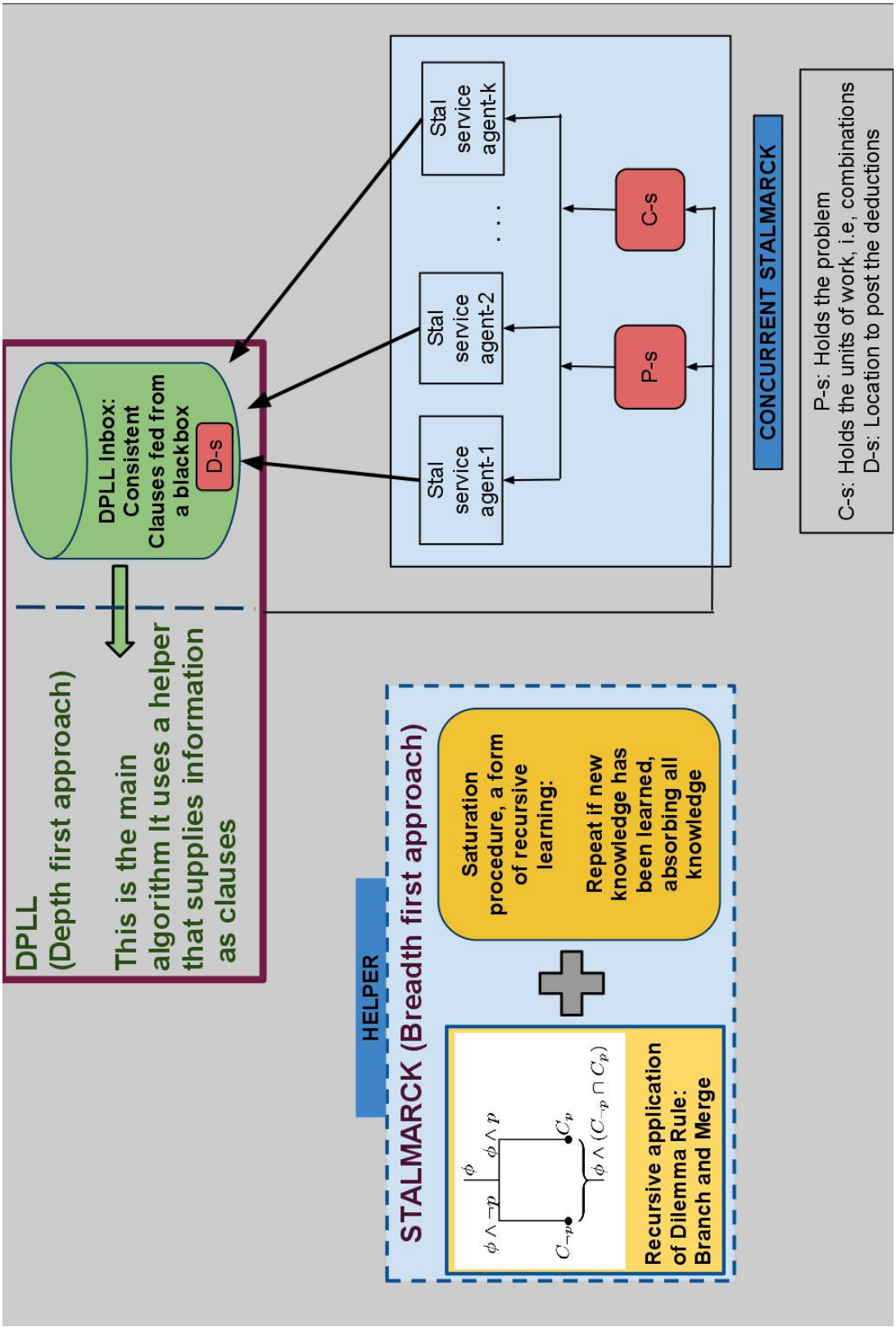


Figure 6.2: High level interaction diagram for DPLL-ConcurrentStalmarck

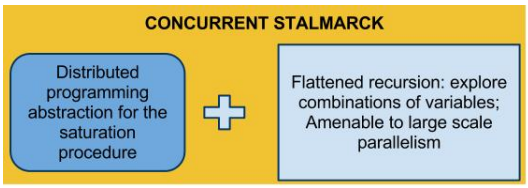


Figure 6.3: Gist of the concurrent Stalmarck implementation

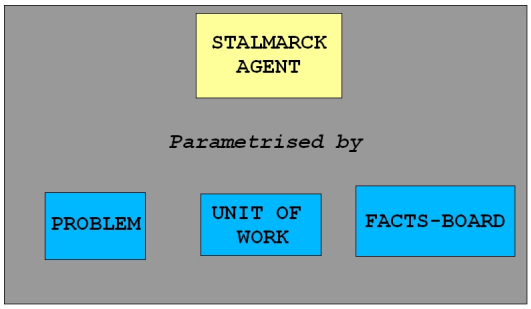


Figure 6.4: Stalmarck Agent

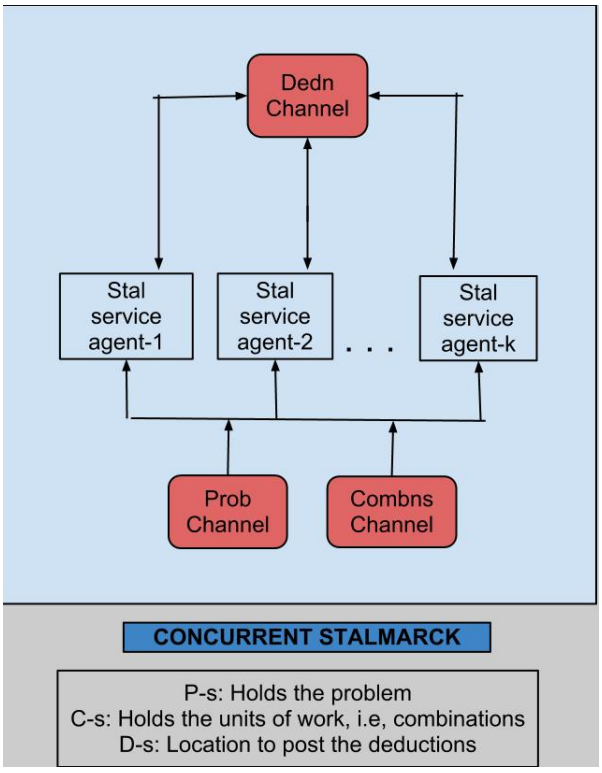


Figure 6.5: Interaction diagram for the concurrent Stalmarck implementation

6.5.2 High level description of the Concurrent Stalmarck algorithm

In this section, we provide a high-level description of the algorithm, *Concurrent Stalmarck*. The description is given in a top-down fashion, with key operations described individually.

concurrStal(P,n,k)

1. - Let P be the problem; number of variables: n; saturation level : r;
   - Let *DednChannel* be a *channel* that can hold equivalences. In Alice ML, *Channels* operate as a *stack* data structure and the *get* and *put* operations behave accordingly. Let the top element of *DednChannel* be $top(DednChannel)$.
   - Insert a dummy equivalence into *DednChannel*.
2. Convert the given problem into triplets and compute the associated simple rules (triggers). The triplicate conversion which introduces new variables will give the variable representing the whole problem, say, v_{prob}. Let the initial assignment be A_I. As explained earlier in §6.3.3.1, to use the solver as a tautology checker, we set the negation of the formula to false and aim to derive a contradiction; to use it as a clause-learning process, we set the formula to true and pass the derived consequences as the learned information and hence A_I is $v_{prob} \equiv \top$ or $\neg v_{prob} \equiv \bot$ as required.
3. The concurrent saturation procedure for a given problem , P, with n variables and with recursion depth, r : **concurrSaturationForGivenDepth(P,n,r)** is computed by iteratively exploring combinations for $i = 0, 1, \ldots, r$ using **exploreCombnsAndSaturateForLevel_k(prob,n,k)** that was described above with equivalences deduced at each level getting passed to the next level.

Concurrent saturation procedure for level k:
exploreCombnsAndSaturateForLevel_k(P,n,k)

- For a problem with n variables, for depth k, explore all possible $\binom{n}{k}$ combinations, $C_1, C_2, \ldots, C_{\binom{n}{k}}$, using the function, *exploreASingleCombn*, defined above.
- Let E_{C_j} denote the deductions that have to necessarily hold for the combination of variables, C_j.
- Let E_k = (equivalence) union of E_{C_j}, $j = 1, 2, \ldots, \binom{n}{k}$. Post E_k, to *DednChannel*.
- Check if new information has been found by comparing the original equivalence, E_0 and the equivalence in *DednChannel*. If yes, then, repeat the processing of combinations. I.e. go to the processing of combination step, i.e. **exploreASingleCombn(P,n,C_j)**
- In the step where equivalences are posted, the equations between (sub)formulas held in each E_{C_j} is valid for the entire problem. Thus, the processing required for a given combination can be carried out independently.

Function to derive the consequences of the given combination of variables, C_j:
exploreASingleCombn(P, n, C_j):
For the combination, C_j, consider all possible truth-value assignments (i.e. 2^k assignments): $A_1, A_2, \ldots, A_{2^k}$.

- For each assignment, say, A_q, Apply 0-saturation to the problem using the following input: the equivalence given by $top(DednChannel)$ and the assignment A_q . Get the deductions from the application of 0-saturation, in the form of the augmented equivalence, say E_{A_q}.

  As explained in §6.2, 0-saturation takes an equivalence and a variable assignment, applies the simple rules, augments the given equivalence with the deductions obtained and returns the new equivalence.

- The assignments, $A_p - s$, $p = 1, \ldots, 2^k$ are arranged as a truth-table, with two consecutive members differing in one column. Take the intersection of the equivalences E_{A_q}, for $q = 1, 2, \ldots, 2^k$. Call this E_{C_j}. The intersection is performed as follows to account for saturation at multiple levels:

  – Let *eqvAssList* be the list (E_{A_q}, A_q), for $q = 1, 2, \ldots, 2^k$

  – For every two consecutive members of *eqvAssList*, say (E_{A_i}, A_i) and (E_{A_j}, A_j), do the following:

    * Perform zero saturation for the pair and obtain the intersection of the resulting equivalences

    * If the new equivalence is different from the original one, repeat the above step, else return the new equivalence along with $baseAss_{ij}$, the assignment with the last column dropped. A_i and A_j differ in their last column. Thus, dropping the last column takes us one level down the saturation tree

  – Repeat the above step of pairwise reduction to progressively reduce *eqvAssList* to a single equivalence. This is the intersection of the truth-table assignments, with saturation performed for every branch-merge

As mentioned in the description above, the processing of an individual combination can be performed by an independent process (*agent*). Thus, a shared-memory computational model is not required. Furthermore, the individual processes are not tightly coupled and do not need to communicate very often. The only information that an agent needs is $top(DednChannel)$ and access to post to *DednChannel*. This gives the freedom of allowing these agents to run on many different processes and possibly different workstations, without any dependencies on the state of the other processes and without creating any bottlenecks for other processes that use the agent's results. Our implementation can be considered to be an implicit form of a *message-passing* computational model, because, though the computational agents do not communicate directly with each other, they do so via *DednChannel*. We have used Alice ML's *channel* feature (see §5.6) to implement *DednChannel*. We refer to this agent as the *Stalmarck agent service* and it is described in detail in the next section.

6.5.3 Stalmarck agents as services

A single *Stalmarck agent* can be described as a service that is running as an independent process that computes **exploreASingleCombn(P,n,**C_j**)**, as described above. In a multithreaded setup, the processes run on the same machine. In the distributed setup, they can run on different machines and their functions can be invoked remotely. The computation carried out by the service is described below. The Stalmarck agent services are bound to the following three channels (as given in Figure 6.4) at the time of its creation:

- A *work stream* (implemented using Alice ML channel feature), say *CombnsChannel*. This is the placeholder for the units of work (combinations) to be processed.
- A *problem stream*, where the problem gets posted, say *ProbChannel*.
- A third stream is created where the agents post their deductions, say *DednChannel*.

The *data-driven* consumption model enabled by the *incremental evaluation* behaviour implemented in Alice ML (described in §5.5.2.1, §5.5.2.1, §5.6.2) have been used to engineer the facilities of *waiting* for work. The computation performed by the *Stalmarck agent* proceeds as follows:

- Waits on *ProbChannel*, where the problem will appear.
- Once a problem appears, the service will proceed to the next step, to fetch a combination from *CombnsChannel*; if there is no combination it will *wait*.
- After fetching a combination successfully, it will apply the same to the problem and post the deductions, if any, to *DednChannel*.
- The DPLL agent or another Stalmarck agent can access the location, *DednChannel*, where the deductions are provided.
- Furthermore, at each stage, the *relevant* results from *DednChannel* are applied to the problem. In some cases, this can dramatically reduce the problem. By *relevant* results, we mean only literals that have a presence in the current problem, i.e. a unit clause with literal l is relevant only if either l or $\neg l$ is present in the problem.

6.5.4 Workflow of the Concurrent Stalmarck implementation

The workflow of the Concurrent Stalmarck implementation is as follows:

1. The problem and initial assignment are posted in the *ProbChannel* and the individual combinations which constitute independent units of work are posted in a *CombnsChannel*.
2. A user-specified number of *Stalmarck agents* are spawned. These workers are parametrised by: *ProbChannel, CombnsChannel, DednChannel*.
3. A worker picks a unit of work from the work stream and processes it and reports its results to the data-repository location.
4. Upon finishing its work, the worker picks up the next piece of work from the work stream location, if available, and waits otherwise.
5. When all the units of work are exhausted, a *referee* makes a check to identify if any *new* results have been deduced compared to the original state. If it is so, then the work stream is re-populated with the original content, the original assignment is augmented with the new information and the entire cycle is repeated. If no *new* results have been posted, the original process is terminated.

6.5.5 Producer-consumer pattern, Resource-management

The Stalmarck agent service implementation can be viewed as an instance of the *producer-consumer* abstraction as described in §5.4.2.1. As described above, once triggered, the computation of the agent acts as a *consumer* and picks a combination from *CombnsChannel*, the *producer*. It works on it and upon finishing the work, posts the results to *DednChannel* and *waits* for the next combination from the stream *CombnsChannel*. As described above, the *data-driven* consumption model enabled by Alice ML's *incremental evaluation* facilities have been used to implement the *waiting* feature. Any agent that has completed its computation picks the next unit of work from *CombnsChannel* without needing any explicit communication. No explicit communication to individual workers is involved either as all units of work are posted on to the same work channel. If *CombnsChannel* is empty, the agent waits for it to be populated with an element (the semantics of Alice ML's Channel library means that this waiting continues till *CombnsChannel* is closed explicitly). Thus, the need for expensive communication to facilitate work stealing and load balancing is avoided, achieving an implicit form of *resource management*. Furthermore, the flexibility on the number of agents working on the problem allows for enforcing resource-management techniques, as the user can specify the number of Stalmarck agents depending on the computational resources available. E.g., if the solver is deployed in a network of workstations, then the number of Stalmarck agents can be adjusted to optimally utilise the available number of idle workstations.

6.5.6 Abstractions developed

We have built on standard abstractions found in the parallel programming literature to address the particular scenarios in our implementation of the Stalmarck-based concurrent algorithm, as explained below.

Abstraction for the saturation technique, adaptation of *barrier* We have developed *saturation_abstraction*, an abstraction for implementing the saturation technique of Stalmarck's algorithm in a parallel setting. This involves Step 1, Step 2 and Step 5 of the above work flow given in §6.5.4. The code fragment for this abstraction is given in Listing 6.6. The saturation procedure is a technique that is used in other theorem proving scenarios as well and thus this abstraction

can potentially be reused to tackle those as well. Furthermore, the saturation technique is a form of *recursive learning*: one of deriving information and repeating the small steps in the light of the newly derived information, until no *new* information can be derived. The saturation abstraction can be reused/extended to address other scenarios of recursive learning as well. *saturation_abstraction* shares similarities with *barrier*, a standard abstraction found in concurrent computing literature (explained in §5.4.2.3). It involves waiting for all the combinations to finish their computation (i.e. compute consequences as equations between (sub)formula) before making the decision to perform the next iteration (if new information has been found by one or more combination) or not (no new information was derived).

Computational pattern used in deduction performed by each worker As explained in the earlier sections, in our implementation of the Concurrent Stalmarck algorithm, each agent works on a unit of work, i.e. a combination of variables and computes the deductions that have to compulsorily hold for that combination of variables. The deduction process of each worker in turn, involves:

- Application of the *simple rules* for all the possible truth assignments for the given combination of variables, giving the corresponding deductions.
- Aggregation of the deductions from all the truth assignments and computation of their intersection.
- The intersection thus computed is the required output for the given combination, problem and assignment.

Our implementation of this deduction process can be considered as an instance of the standard *Map Reduce* abstraction found in concurrent programming literature (see §5.4.2.4) as follows:

- The *data* is the list of truth assignments being considered.
- The *map* operation is the application of the simple rules to a given truth assignment.
- The *reduce* operation is the intersection operation over the results of the map operation carried out over the list.

Listing 6.6: Alice ML code for saturation abstraction

```
fun saturation_abstraction compareFn getState getAll takeStockFn agFnList=let
 fun saturation_abstraction2 compareFn getState getAll takeStockFn agFnList =let
  val oldState = getState();
  do returnWhenAllDone agFnList;
  val resList = getAll();  do takeStockFn  resList oldState;
  val newState = getState();
 in
  if not(compareFn oldState newState) then
   saturation_abstraction2 compareFn getState getAll takeStockFn  agFnList
  else  ( )
 end
in
  Exn.catch (Exn.reraise)
  (fn () => saturation_abstraction2 compareFn getState getAll takeStockFn agFnList)
end
```

6.6 Concurrent DPLL

This solver uses the standard DPLL algorithm, in an asynchronous setting. At each choice-point, two threads are spawned asynchronously to explore the respective sub-trees. The thread that comes back first with a satisfiable assignment terminates the other thread. Furthermore, termination of a thread terminates all the sub-threads spawned under it. Alice ML's implementation of threads does not support automatic termination of child threads. We have implemented a modified version which does terminate the child threads; the code for the same is given in Appendix §A 3.

This implementation can show performance gains in cases where the satisfiable assignment is at a shallow level on one of the branches and exploration of the other branch takes a very long time. The gains made by this feature can be analysed by comparing the performance of: DPLL, DPLL with orders-flipped at choice-points and concurrent-DPLL. The code outline for this solver is given below.

Listing 6.7: Concurrent-DPLL

```
fun takeFastestAndKillOther(t1,r1) (t2,r2) =
 case (Future.awaitEither(r1,r2) ) of
   FST(Sat(_)) => (Thread.terminate(t2);r1)|FST(Unsat) => r2
 | SND(Sat(_)) => (Thread.terminate(t1);r2)|SND(Unsat) => r1

fun doConcurrent_DPLL prob : result  = let
 fun solveAssign(prob,lit): result =  let
    val rProb =  doAllUnitCl (doAllPureLit(remTauts (prob,lit)))
 in
  case testProb(rProb) of
    Sat sat_assign  =>  Sat sat_assign|Unsat =>  Unsat
    |UNKNOWN => branch(rProb, pickBranchingLit (rProb))
 and
 branch ( prob, lit) : result   =   let
   (*Spawn two searches with orders of traversal flipped *)
    val (t1,r1)=  spawnThread (solveAssign (prob, lit) );(*r1 _future *)
    val (t2,r2)=  spawnThread (solveAssign (prob, ~lit));(*r2 _future *)
   in     do takeFastestAndKillOther(t1,r1) (t2,r2)   end
in  solveAssign prob end
```

6.7 Evaluation

As described in 1, our *object-level* hypothesis for SAT is as follows:

> Use of an asynchronous mode of execution enables development of two novel algorithms:
>
> 1. A hybrid solver, based on the DPLL and Stalmarck algorithms, which shows gains in some test problem classes considered and does not show significant slowdown in some other problem cases examined in this work.
> 2. A novel concurrent algorithm based on applying concurrent techniques to the Stalmarck algorithm, such that it is amenable to large scale parallelism.

In this chapter, we have described these new approaches to engineer SAT solvers, made feasible by an *asynchronous* mode of execution. Proof-of-concept prototype implementations of these approaches were also described. In this section, we report on experiments conducted on these prototype solvers, using different problem classes.

We claim the following:

1. In comparison to the DPLL-CDCL solver, the hybrid SAT solver *DPLL-Stalmarck*, by virtue of using the Stalmarck solver as an asynchronous clause learning process, uses the learned clauses *dynamically* to:
   - Prune its search space;
   - Reduce the time taken to find an answer (SAT or UNSAT).
2. In comparison to the sequential Stalmarck, *Concurrent Stalmarck* enables a previously unexplored, novel way of applying concurrency and distribution, to engineer a new algorithm, based on the original Stalmarck algorithm. Used as a tautology checker, the concurrent version reduces the time taken in comparison to the sequential Stalmarck.

In §6.7, we describe the limitations to the empirical evaluation conducted. In §6.7.1 and §6.7.2, we explain our process of evaluation of the prototypical implementations of DPLL-Stalmarck and Concurrent Stalmarck respectively. For each of these, we give the following:

- Rationale for design;
- Why we expected it to work;

- Choice of empirical tests to test the performance;
- Empirical results and an analysis of the same.

We end the section with an assessment of how the prototypes fare on the other aspect of our hypothesis, i.e. the methodological criteria of : use of abstractions that aid portability, ease of prototyping and incremental development.

Platform imposed limitations to empirical evaluation

Both the prototypes described in this chapter use a message-passing style of communication and do not use shared-memory. Thus, they are ideally placed for multithreaded implementations and distributed computing architectures. However, Alice ML's distribution and remote invocation facilities incur a significant overhead in terms of computational time as they involve cloning of data structures and proxy function calls. This drawback of Alice ML as a platform proved a limiting factor in our empirical evaluation of distributed versions of the prototypes described in this chapter. So, we restricted ourselves to multithreaded versions of the prototypes for the purpose of empirical evaluation.

Thus, though the use of a functional approach via Alice ML serves as an excellent implementation platform choice in terms of high-level language support for developing abstractions and ease of prototyping, it has limited the scope of our experiments.

6.7.1 DPLL-Stalmarck

In this section, we explain our evaluation process for the prototypical implementation of the hybrid approach for SAT, explained in §6.3.

Rationale for design

- Combination of complementary approaches.
- It can derive many clauses simultaneously.
- Using the concurrent variant of Stalmarck's algorithm that we have implemented, we can organise the learning process as a collection of distributed processes, thus enabling optimal utilisation of distributed architectures.

- This new form of distribution gives an alternative to the current work-partitioning methods found in the literature on parallel SAT solvers almost all of which are DPLL-based and use a variant of the *guiding path* technique for search space partitioning.

Why we expected it to work?

- The manner of spanning the search tree is different from that of DPLL and the process of learning is not conflict-driven unlike the CDCL techniques embedded in DPLL.
- Stalmarck algorithm's clause-learning mechanism is different from that of CDCL (conflict-driven clause learning) based DPLL solvers [Marques-Silva et al., 1996].
- It does not rely on the DPLL arriving at a conflict in its search tree and *learning* a clause from the *conflict*. It explores the search tree in a breadth first manner and uses the formula relations and hence the structure in the given formula, to derive the learned clauses, with the aid of the *dilemma rule* and the *saturation* technique.
- The hybrid architecture is generic enough for any information providing agent to be plugged in and relies only on message-passing. Thus, the state-of-the-art in DPLL can still be used and the hybrid design can be ported to other solvers, achieving the separation in design and implementation mentioned in our developmental hypothesis.

6.7.1.1 Details of experiments: problem classes, solvers, metrics

Problem classes: We describe below the classes of problems that we have used to compare the performance of our hybrid solver, DPLL-Stalmarck, with that of DPLL-CDCL:

Pigeon hole problems: For a given n, the well known pigeon hole problem, *PHole(n)*, states that *(n+1)* pigeons cannot fit n holes. Our encoding gives $n*(n+1)$ propositional variables and $(n+1)+n*(n*(n+1)/2)$ clauses. This is *UNSAT* for all n. We have conducted experiments for $n = 2, 3, \ldots, 13$

Urquhart problems: Originally described in [Urquhart, 1987] as a hard class

of problems for resolution, *Urquhart(k)*, for a given k, is a chain of equivalences of the form

$$l_1 \leftrightarrow l_2 \leftrightarrow \ldots \leftrightarrow l_k \leftrightarrow l_1 \leftrightarrow l_2 \leftrightarrow \ldots \leftrightarrow l_k.$$

Urquhart(k), when converted to CNF (using a naive conversion procedure), has k variables and 2^{k-1} clauses. This is a tautology for all k, with the trivial assignment of setting all variables to True. We have conducted experiments for $k = 2, 3, \ldots, 13, 15, 20, \ldots, 50$

Uniform Random-3-SAT: Uniform Random-3-SAT is a family of SAT problems distributions obtained by randomly generating 3-CNF formulae in the following way: For an instance with n variables and m clauses, each of the m clauses is constructed from 3 literals which are randomly drawn from the 2n possible literals (the n variables and their negations) such that each possible literal is selected with the same probability of 1/2n. Clauses are not accepted for the construction of the problem instance if they contain multiple copies of the same literal or if they are tautological (i.e., they contain a variable and its negation as a literal). Each choice of n and m thus induces a distribution of Random-3-SAT instances. Uniform Random-3-SAT is the union of these distributions over all n and m. One particularly interesting property of uniform Random-3-SAT is the characterisation of hardness of a problem of this class, using the clause-variable ratio, i.e. m/n [Gent and Walsh, 1994b].

Solvers used

DPLL-CDCL: Sequential SAT solver based on DPLL algorithm, augmented with CDCL.

DPLL-Stalmarck: Our novel hybrid SAT solver, combining the DPLL and Stalmarck algorithms

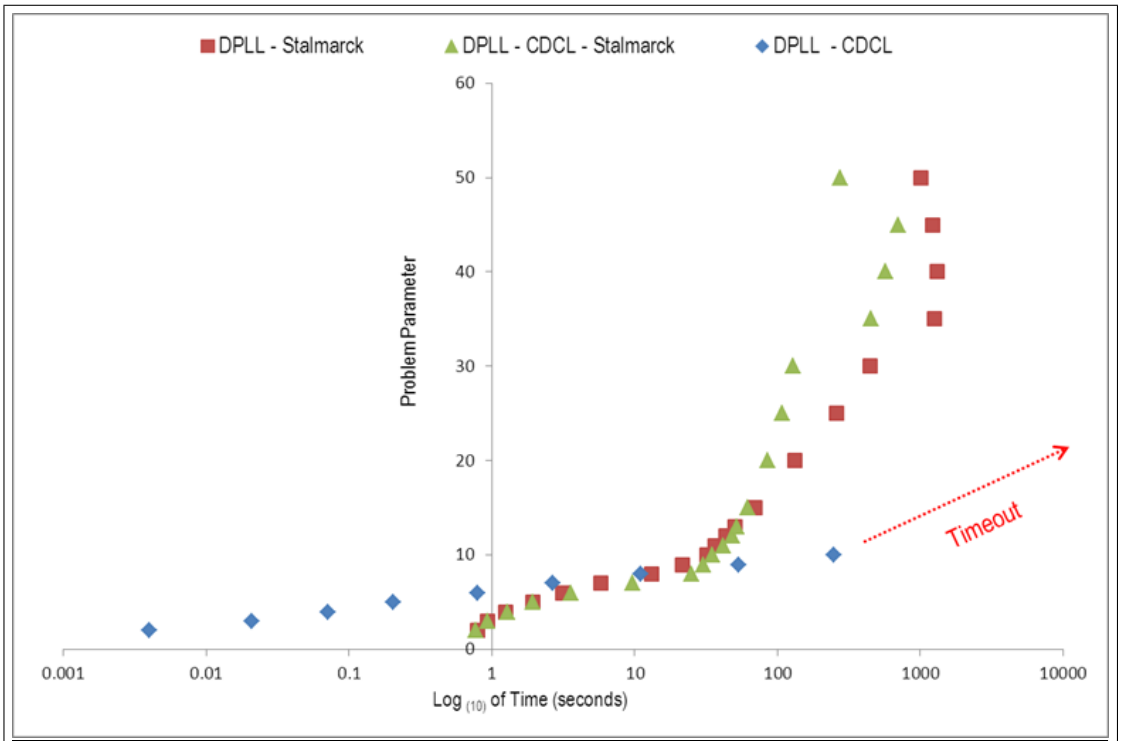
1. Fully asynchronous mode
2. With a pre-set time for the helper to work on, before the DPLL process starts

DPLL-CDCL-Stalmarck: Same as DPLL-Stalmarck, but with DPLL-CDCL, instead of DPLL.

Metrics used: Time taken by the solver to compute the answer and size of search-space spanned by the solver. By size of the search space, we refer to the number of case-splits performed by the solver.

Concurrent implementation considered for empirical results multithreaded version

Platform specifications Intel(R) Xeon(TM) CPU 3.60 GHz, 3.86 GB RAM, running Scientific Linux release 6.3 (Carbon); Alice ML version: 1.4



Prob. Param -->	2	3	4	5	6	7	8	9	10	11	12	13	15	20	25	30	35	40	45	50
DPLL - CDCL	0.004	0.021	0.071	0.204	0.794	2.666	10.967	53.143	249.147	>400	>400	>400	>600	>600	>800	>1300	>2200	>2400	>3400	>4400
DPLL - Stalmarck	0.799	0.933	1.252	1.928	3.158	5.79	13.209	21.753	31.865	36.659	43.368	50.271	69.989	131.649	258.195	443.472	1249.12	1304.08	1214.96	1008.24
% Performance	-19875%	-4343%	-1663%	-845%	-298%	-116%	-20%	59%	87%	DPLL - Stalmarck outperforms DPLL - CDCL										

Timeout Better Performance

Prob. Param -->	2	3	4	5	6	7	8	9	10	11	12	13	15	20	25	30	35	40	45	50
DPLL - Stalmarck	0.799	0.933	1.252	1.928	3.158	5.79	13.209	21.753	31.865	36.659	43.368	50.271	69.989	131.649	258.195	443.472	1249.12	1304.08	1214.96	1008.24
DPLL - CDCL - Stalmarck	0.777	0.932	1.284	1.927	3.582	9.646	25.139	30.146	34.844	41.585	48.553	51.738	62.339	85.687	108.459	129.192	450.563	570.672	697.741	275.13
% Performance	3%	0%	-3%	0%	-13%	-67%	-90%	-39%	-9%	-13%	-12%	-3%	11%	35%	58%	71%	64%	56%	43%	73%

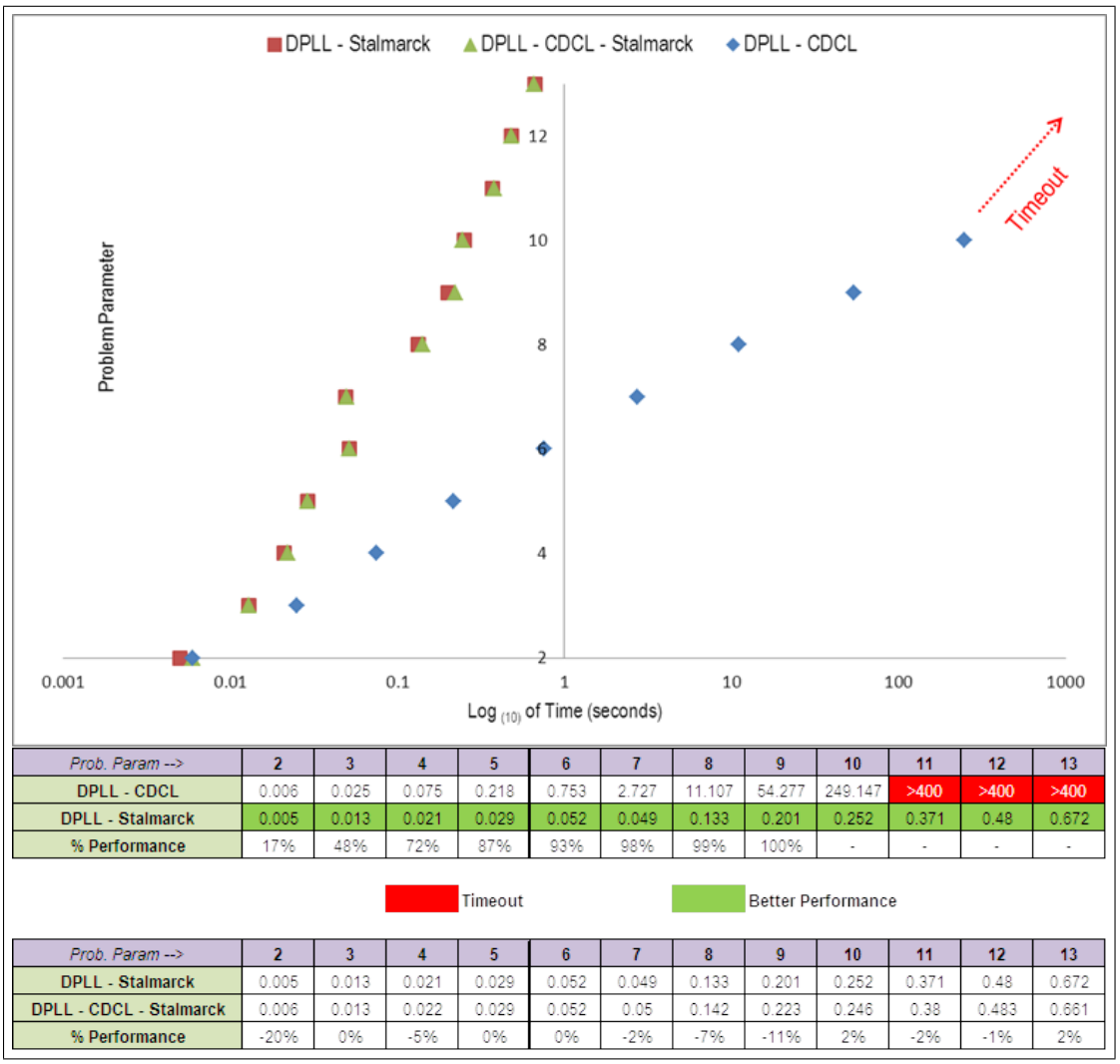
Figure 6.6: Test data (time taken) for Urquhart(n), with an asynchronous Stalmarck-helper

Prob. Param -->	2	3	4	5	6	7	8	9	10	11	12	13	15	20	25	30	35	40	45	50
DPLL - CDCL	6	14	30	62	126	254	510	1022	2046	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
DPLL - Stalmarck	6	14	30	62	126	254	510	1022	818	882	790	860	832	770	728	718	980	1054	1074	1048
% Pruning	0%	0%	0%	0%	0%	0%	0%	0%	60%	DPLL - Stalmarck outperforms DPLL - CDCL										

Prob. Param -->	2	3	4	5	6	7	8	9	10	11	12	13	15	20	25	30	35	40	45	50
DPLL - Stalmarck	6	14	30	62	126	254	510	1022	818	882	790	860	832	770	728	718	980	1054	1074	1048
DPLL - CDCL - Stalmarck	6	14	30	62	126	254	416	390	379	385	379	374	373	343	324	303	417	430	416	392
% Pruning	0%	0%	0%	0%	0%	0%	18%	62%	54%	56%	52%	57%	55%	55%	55%	58%	57%	59%	61%	63%

Figure 6.7: Test data (size of search space) for Urquhart(n), with an asynchronous Stalmarck-helper

[1]Dotted line shows that the solver timed-out and N.A refers to the corresponding search space

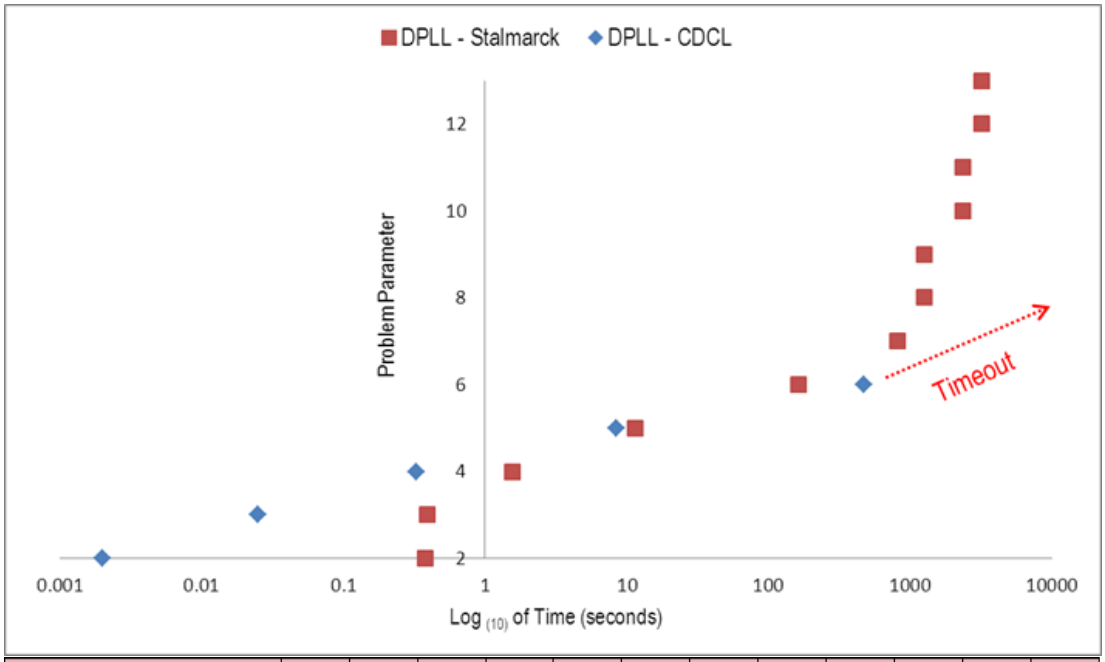


Prob. Param -->	2	3	4	5	6	7	8	9	10	11	12	13
DPLL - CDCL	0.006	0.025	0.075	0.218	0.753	2.727	11.107	54.277	249.147	>400	>400	>400
DPLL - Stalmarck	0.005	0.013	0.021	0.029	0.052	0.049	0.133	0.201	0.252	0.371	0.48	0.672
% Performance	17%	48%	72%	87%	93%	98%	99%	100%	-	-	-	-

Timeout Better Performance

Prob. Param -->	2	3	4	5	6	7	8	9	10	11	12	13
DPLL - Stalmarck	0.005	0.013	0.021	0.029	0.052	0.049	0.133	0.201	0.252	0.371	0.48	0.672
DPLL - CDCL - Stalmarck	0.006	0.013	0.022	0.029	0.052	0.05	0.142	0.223	0.246	0.38	0.483	0.661
% Performance	-20%	0%	-5%	0%	0%	-2%	-7%	-11%	2%	-2%	-1%	2%

Figure 6.8: Test data (time taken) for Urquhart(n), compositional approach: with an initial time of 60s for the Stalmarck-helper



Prob. Param -->		2	3	4	5	6	7	8	9	10	11	12	13
Time Taken	DPLL - CDCL	0.002	0.025	0.326	8.452	469.946	>1200	>1400	>1800	>3000	>3000	>4000	>4000
	DPLL - Stalmarck	0.379	0.39	1.57	11.53	164.58	814.21	1270.81	1270.81	2359.48	2359.48	3203.24	3203.24
	% Performance	-18850%	-1460%	-382%	-36%	65%	DPLL - Stalmarck outperforms DPLL - CDCL						
Search Space Size	DPLL - CDCL	2	16	70	428	3358	NA	NA	NA	NA	NA	NA	NA
	DPLL - Stalmarck	2	16	74	540	4152	12084	13248	13248	14170	14170	13582	13582
	% Pruning	0%	0%	-6%	-26%	-24%	DPLL - Stalmarck outperforms DPLL - CDCL						

Figure 6.9: Test data (time taken, search space) for PHole(n), with an asynchronous Stalmarck-helper

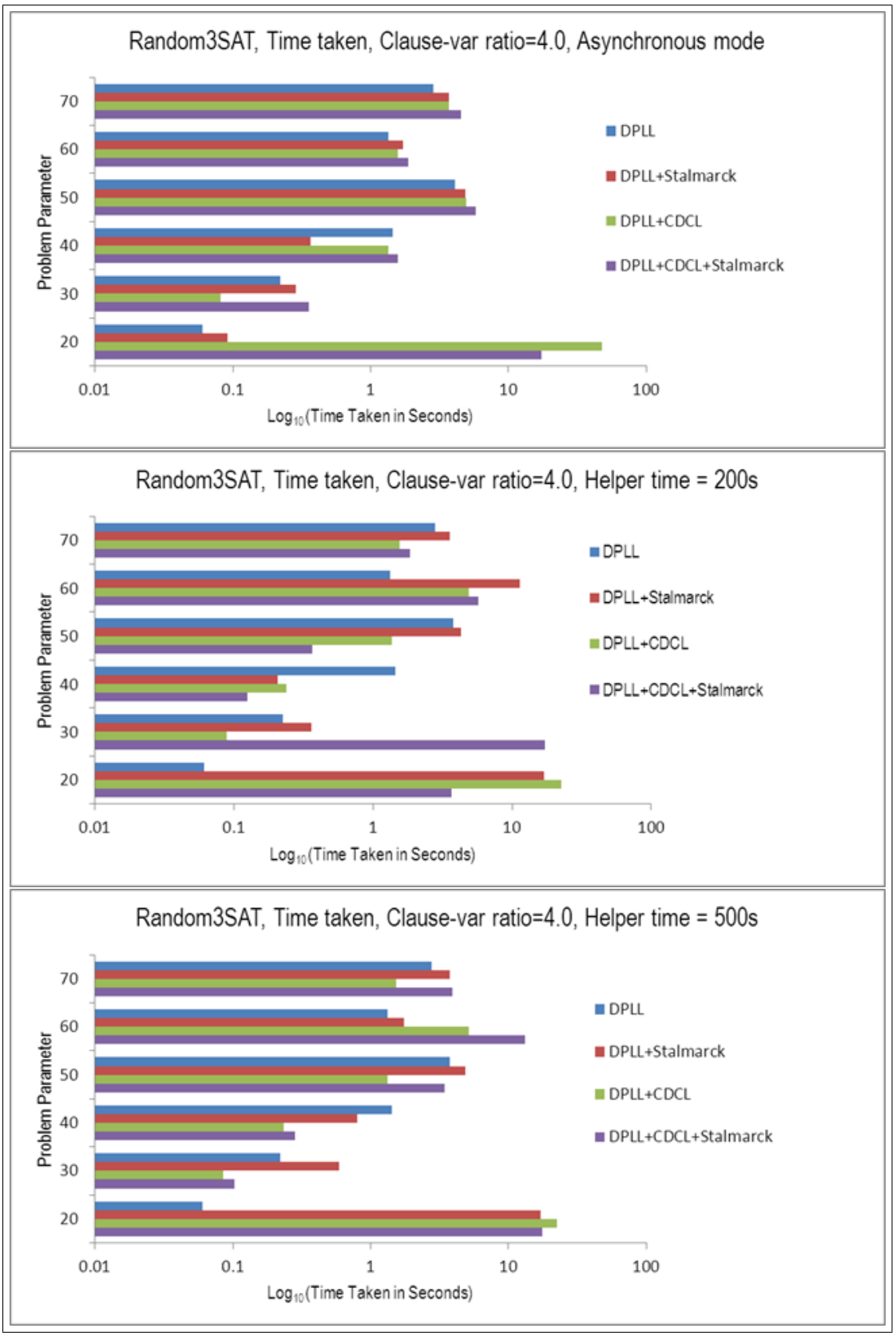


Figure 6.10: Test data (time taken) for Random3SAT; Clause/Var=5.0; n=20,30,...,80; From top: using an asynchronous Stalmarck-helper, compositional approach, with an initial time of 200s, 500s for the Stalmarck-helper

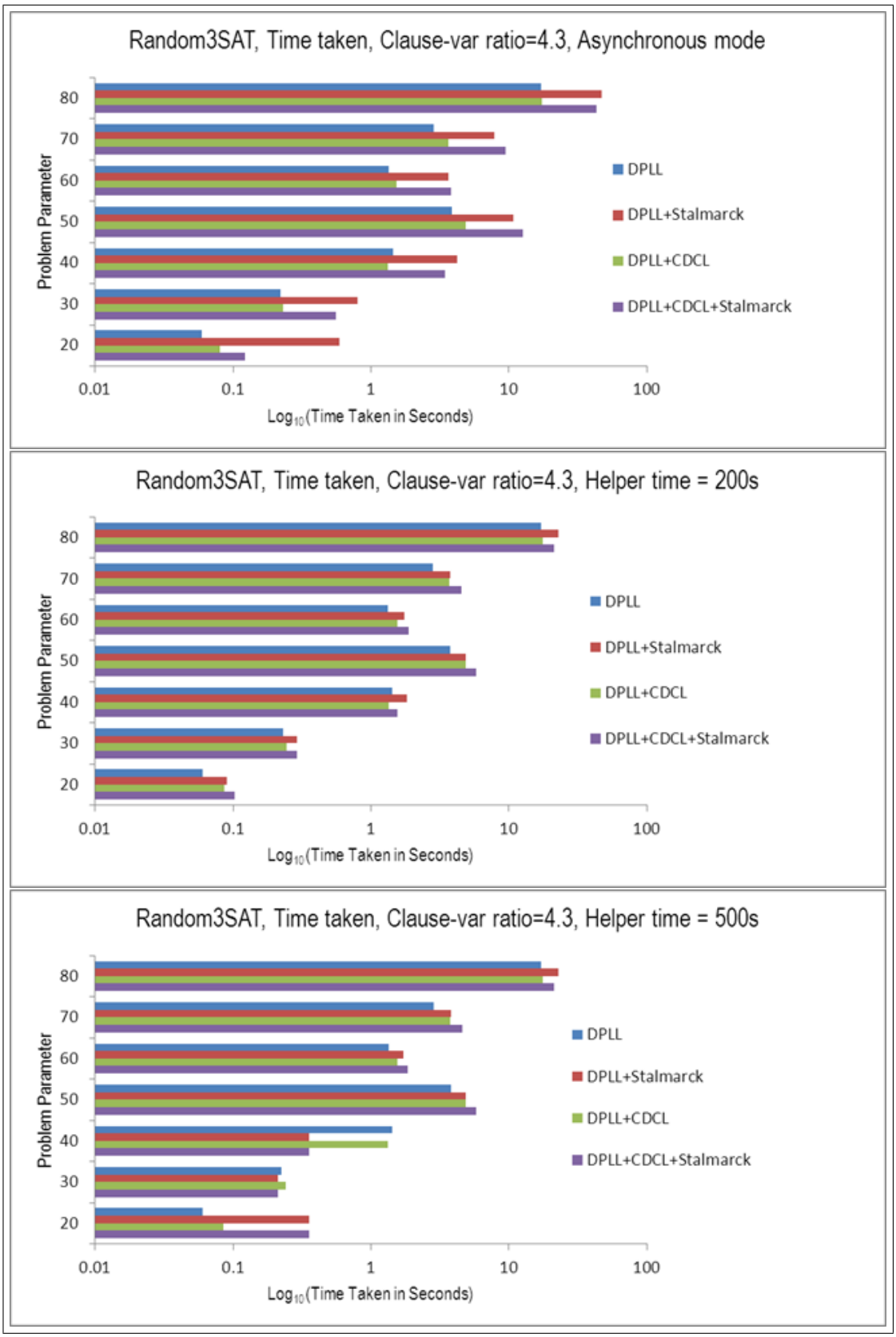


Figure 6.11: Test data (time taken) for Random3SAT; Clause/Var=5.0; n=20,30,...,80; From top: using an asynchronous Stalmarck-helper, compositional approach, with an initial time of 200s, 500s for the Stalmarck-helper

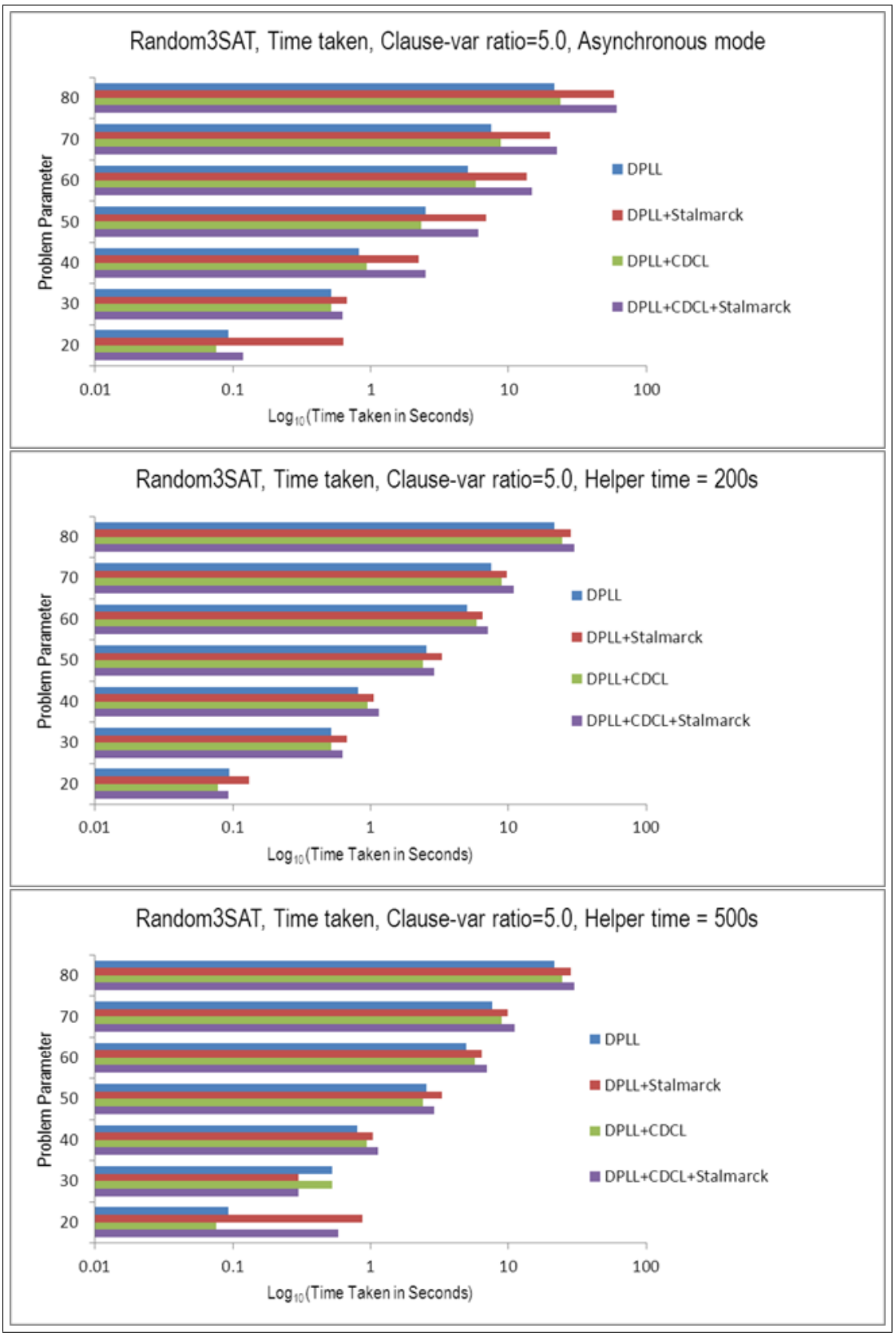


Figure 6.12: Test data (time taken) for Random3SAT; Clause/Var=5.0; n=20,30,...,80; From top: using an asynchronous Stalmarck-helper, compositional approach, with an initial time of 200s, 500s for the Stalmarck-helper

6.7.1.2 Analysis of empirical results

Urquhart Figure 6.6 and Figure 6.7 give the relevant data for the comparison, with the helper working in the fully asynchronous mode and Figure 6.8 gives the data, with the helper given an initial time of 60s. We tried a few different values, lesser and more than 60s. The lesser values did not help for the bigger problems and increasing the time did not make a difference for the problem parameters considered. So, we chose 60s as the value for the helper time, for all the problem parameters considered. The data can be summarised as follows:

DPLL-Stalmarck vs DPLL-CDCL This comparison gives an evaluation of the efficacy of the CDCL learning technique and the Stalmarck clause learner.

DPLL-Stalmarck outperforms the DPLL-CDCL solver, in terms of both search space and time. For $n > 9$, in the fully asynchronous mode and for $n > 3$, in the compositional approach, with a helper time of 60s, the DPLL-Stalmarck solver uniformly outperforms the DPLL-CDCL solver, in a significant manner.

Thus, the empirical data above confirms that for this problem class, when used with the DPLL algorithm, the clause learner based on the Stalmarck algorithm, used in our hybrid solver, DPLL-Stalmarck, fares better than the CDCL technique.

DPLL-Stalmarck vs DPLL-CDCL-Stalmarck This comparison informs us about the efficacy of the interplay between the CDCL and Stalmarck clause-learning mechanisms. The test data shows that the DPLL-Stalmarck solver is faster than the DPLL-CDCL-Stalmarck solver, for $n < 13$ and is slower for $n > 13$. The search space size shows a similar behaviour.

Thus, the empirical data above leads us to conclude that for large n, for this problem class, the combined power of CDCL and Stalmarck fares better than the stand-alone Stalmarck clause learner, when used within a DPLL-CDCL solver.

The Urquhart problem class is known to be difficult for the DPLL algorithm as it has to search through almost all possible cases [Urquhart, 1987]. It is also an example of a problem class, whose implicit structure is lost in the CNF conversion process and thus the CDCL learning technique will also fail to capitalise

on the implicit structure. This problem class has been proved to be of hardness class 2 for the Stalmarck algorithm [Stalmarck, 1994], partly due to the ability of the Stalmarck algorithm to capitalise on the formula structure. Though hardness class 2 is not very easy, it is tractable, for relatively large problems.

A point worth observing here is that CDCL is embedded within DPLL and Stalmarck is an external clause learner. Thus, there is no way to decouple the learner from the DPLL algorithm and execute it as an independent process as in the case of the Stalmarck learner.

Pigeon hole Figure 6.9 gives the relevant data for this problem class, comparing the DPLL-Stalmarck solver, with the DPLL-CDCL solver. The DPLL-Stalmarck solver outperforms the DPLL-CDCL solver, for n>5.

It is well known that this problem class is hard for DPLL. Resolution proofs for pigeon hole problems are exponential in n [Haken, 1985]. It is also a good example for the phenomenon of loss of implicit structural information as a result of CNF conversion.

Random 3 SAT We have tested for clause-variable ratio = 4.3, 4.0, 5.0. Unlike the above two problem classes, there is no uniform behaviour, in the asynchronous case. However, when the helper is given an initial time of 500s, the behaviour shows a more uniform pattern. Relevant data for the same is provided in Figure 6.10, Figure 6.11 and Figure 6.13, for clause-variable ratio = 4.3, 4.0, 5.0, respectively.

6.7.2 Concurrent Stalmarck

In this section, we report early results conducted using the proof-of-concept prototype of the novel concurrent-distributed algorithm for SAT, explained in §6.5. This has been developed by applying concurrent techniques to the original Stalmarck algorithm and enabling a producer-consumer style of processing.

Rationale for design As explained in §6.5, in our design of the concurrent-Stalmarck algorithm, we flattened the recursion involved in the saturation component of the original Stalmarck's algorithm. The individual processes are not tightly coupled and do not need to communicate very often. So, we exploited this latent opportunity for parallelisation in designing a new concurrent solver based on the

Stalmarck algorithm. Our design allows for employing producer-consumer style parallelisation and relies only on implicit message-passing without any requirements for shared memory. Thus, it allows for optimal utilisation of distributed computing environments like clusters and grids. However, in our current work, we have tested it only on multithreaded versions and a local cluster. Another orthogonal point is the following: It gives a new way of task decomposition compared to others seen in the DPLL-based systems in the literature (e.g., guiding path as in PSATO [Zhang et al., 1996]).

Why we expected it to work?

- For a given saturation level, r, the number of candidates for computation are all the possible combinations, i.e. $\sum nCj, j = 1,2,\ldots,r$. The *saturation* aspect of the procedure means that the combinations need to be processed repeatedly if new knowledge has been found. Thus, the number of times an agent performs the computation can be significant, particularly for problems where the number of variables is high. However, the communication needs are less. Thus, a distributed implementation using indirect message-passing is a promising candidate to show gains in speed.

- We have used task decomposition and have organised it as a *data-driven* execution, thus allowing for effective work stealing without the costly overheads of communication to achieve work stealing that is often observed in the literature in other systems.

Empirical results

We have used a multithreaded implementation for the purpose of these experiments on the Urquhart problem class [4]. These early results show significant performance gains for the concurrent implementation, in comparison to the sequential implementation.

[4] Our prototypes are designed to support a large scale parallel computing environment and we have tested these prototypes on a local cluster of workstations. However, as discussed earlier in §6.7, the limitations imposed by the Alice ML platform has meant that we include empirical results only for a multithreaded implementation.

Prob-param	Stalmarck	Concurrent-Stalmarck
3	1.395s	0.040s
4	2.155s	0.055s
5	3.367s	0.070s
6	3.263s	0.087s
7	4.157s	0.106s
8	5.097s	0.125s

Table 6.1: Comparison of time taken by Stalmarck and our novel algorithm, ConcurrentStalmarck, for Urquhart problems

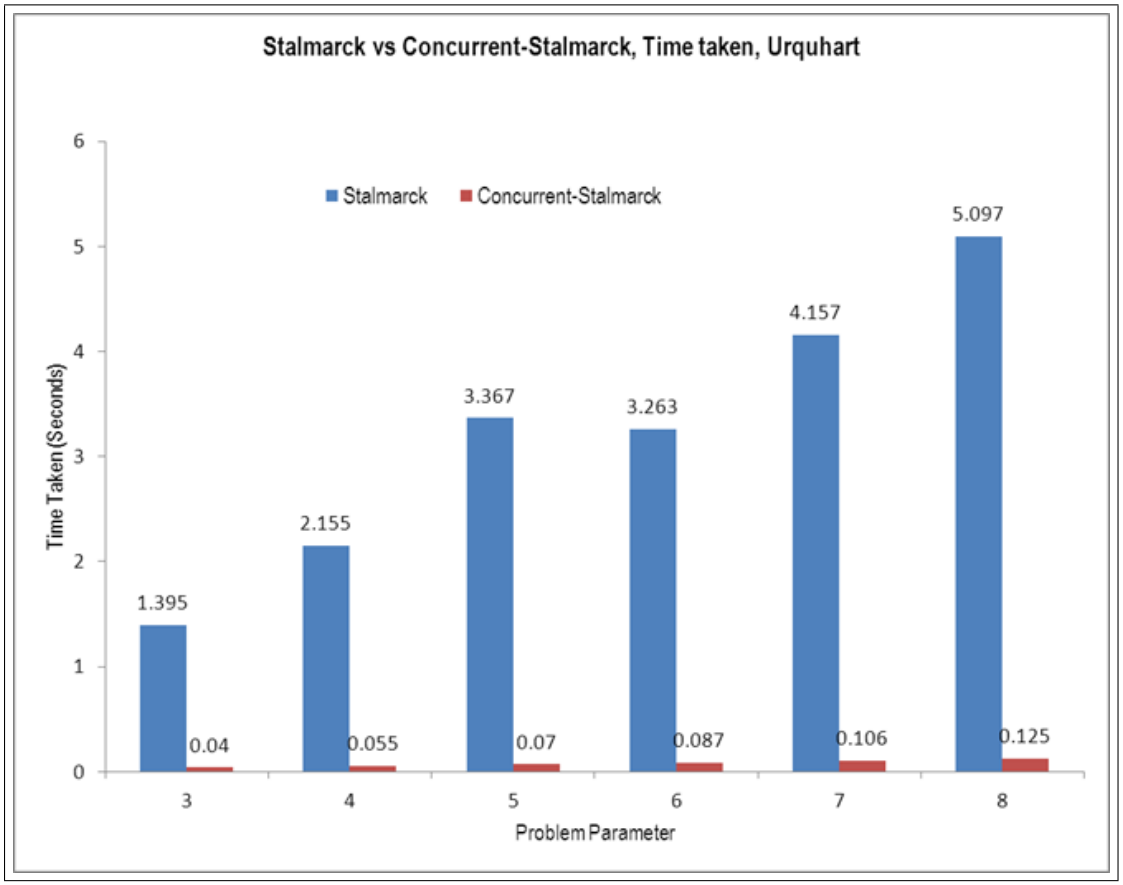


Figure 6.13: Test data for Urquhart problems, comparing sequential Stalmarck solver and the novel concurrent Stalmarck implementation

6.7.3 Methodological criteria

As explained in §6.3.3.3, the *doDPLLwithHelper* abstraction allows for easy prototyping of a hybrid solver allowing for flexible integration of one or more external information providing agents, with the flexibility of even using heterogeneous solvers as helpers. This has been realised using our approach of using programming abstractions and the additional advantage of using a functional programming language has enabled us to implement the abstraction as a higher-order function.

Our approach has allowed us to make an effective isolation of *design* and *implementation* as illustrated by our analysis of the criteria for the helper to be effective in the context of the DPLL-Stalmarck implementation. Our approach enabled easy performance analysis and easy prototyping of alternate experiments.

The use of the *doDPLLwithHelper* abstraction enables clarity of design with respect to the interaction between the solvers. It encapsulates the mechanism used by the DPLL process to use the clauses provided in *dpllInbox*. The mode of provision of the information is thus separated from how it is used. This allows for easy porting of the design to other platforms.

We have demonstrated the utility of using the abstraction to promote incremental development via our prototypes, *DPLL-CDCL-Stalmarck* and *DPLL-ConcurrentStalmarck*, as explained earlier in this chapter. We developed the abstraction *doDPLLwithHelper* to implement the hybrid solver *DPLL-Stalmarck*. This abstraction was used to engineer the solvers,

- *DPLL-CDCL-Stalmarck*, by using DPLL-CDCL as the main solver.
- *DPLL-ConcurrentStalmarck* by replacing the Stalmarck solver with the implementation of our novel algorithm *Concurrent Stalmarck*.

In the *Concurrent Stalmarck* implementation, we have developed a programming abstraction that is similar to the *barrier* abstraction found in concurrent computing literature (see §5.4.2.3). This implementation also employs a novel form of work allocation using the *data-driven* behaviour enabled by the use of incremental evaluation facilitated by the use of Alice ML. This prototype illustrates the scope of applying concurrent techniques via programming abstractions to existing algorithms to develop novel algorithms that are better placed to utilise large scale parallel processing resources.

Our use of a high-level language approach to implement concurrent techniques (as opposed to an API based approach), using abstractions thus greatly enables portability and aids incremental development. It also promotes an iterative development lifecycle as follows :

- Use a high level programming language and programming abstractions to engineer an experimental workbench to prototype and experiment with applying concurrent techniques to engineer a better SAT solver;
- Perform empirical evaluation and analyse the same;
- Use the analysis to improve the prototypes;
- When a prototype has been finalised, port it to a state-of-the-art solver.

Many of the industry-standard SAT solvers involve fine-tuning of various parameters, cache performance and the hardware that the solver is being run on. These SAT solvers are mostly written in C and applying concurrency/distributed programming techniques at a fine level of granularity to these systems is a complicated exercise and can often compromise the fine-tuning that makes them so efficient in the first place. A lightweight thread mechanism and rapid-prototyping facilities allows for easier and richer experimentation. Comparisons can be made with sequential solvers developed in the same framework and the results thus gleaned can be used to port the distributed programming abstractions to other industry standard solvers as well, using their own infrastructure for distribution.

6.8 Related work

In this section, we compare our work with other works on parallelisation of SAT and try to draw out the similarities and differences.

As we saw earlier, information sharing is increasingly being investigated as a technique to boost the efficiency of the current SAT solvers. But the current systems employ CDCL based approaches within a DPLL framework. CDCL techniques generate a huge number of learned clauses as a clause is generated at each conflict. Due to this, management of these clauses becomes a serious concern as adding all of them to the

problem will quickly exhaust the memory. So, most solvers employ some heuristics to choose clauses from the database of clauses generated by the conflicts. Furthermore, as discussed in [Hamadi and Sais, 2009], for many hard instances, conflict analysis leads to a learnt clause which has at least one literal from the level before the conflict level, causing the algorithm to backjump one level.

CDCL is embedded within the DPLL algorithm and so,

- the clause learning happens by spanning the search space in the same way, as the DPLL process
- it suffers from loss of implicit structural information (inherited from the DPLL algorithm)
- it cannot be decoupled from the DPLL process and so, cannot work independently, as a clause learner.

Our hybrid solver, DPLL-Stalmarck uses the breadth-first approach based, Stalmarck clause-learner. This has the advantage of using a complementary search procedure to provide the clauses as opposed to CDCL. An additional advantage is the ability to be able to spawn the Stalmarck solver or more generally one or more helpers in different machines, thus allowing for effective utilisation of a distributed architecture. A strength of the Stalmarck procedure is to leverage on the structure present in the problem as it works using relations between (sub) formulas. When used on the right problem classes, this aspect can greatly aid the power of the information provided by Stalmarck to the DPLL solver.

The combination of DPLL and Stalmarck has been previously explored in [Andersson et al., 2002]. They use a static compositional approach within a proof engine framework. It requires apriori judgements to be made on the hardness of the problem to determine the parameter for each solver in the framework. It does not allow for dynamic interaction between the solvers as enabled by our prototype DPLL-Stalmarck.

Effective work decomposition and managing the cost of load-balancing are critically important for effective parallelisation of applications. Most works on parallel SAT use *guiding path* for work decomposition and API based communication for load-balancing. In our concurrent Stalmarck implementation, we have demonstrated an alternate form of work decomposition for SAT that is not found in the literature to the best of our knowledge. Furthermore, using Alice ML's support for *data-driven*

execution and *incremental evaluation*, we have demonstrated a way of work allotment without the overhead of communication.

6.9 Conclusions

A concise summary of the content of this chapter is as follows:

- We demonstrated the scope and efficacy of applying concurrent programming techniques, to address some previously unexplored possibilities for engineering SAT solvers, by describing the following two novel approaches:

  DPLL-Stalmarck Our hybrid solver, DPLL-Stalmarck, demonstrates a co-operative architecture using two different but complementary solvers based on the depth-first approach based DPLL and breadth-first approach based Stalmarck algorithms, in an asynchronous setting.

  A clause-learner based on the Stalmarck algorithm works independently on the same problem as the DPLL solver. This clause learner acts as a helper to the DPLL solver and provides learned clauses dynamically, to the DPLL solver. It illustrates the power of using the solvers in an asynchronous setup so as to facilitate *dynamic* information sharing and is thus well placed to prune the search spaces of DPLL in a dynamic manner. This also eliminates the requirements of employing resource allocation strategies for the helpers, based on the hardness of the problem for a given helper, which is the approach that has been used in [Andersson et al., 2002].

  Furthermore, the two solvers can work concurrently and independently on the problem, communicating only when there is information to be shared, thus avoiding bottlenecks as well as being able to make the most of distributed architectures. In our current implementation, we have used asynchronous message passing whereby the helper posts the information to *dpllInbox*. This is accessed by the DPLL process which *reacts* to the information. The DPLL process does not directly communicate with the helper. This reactive approach can be looked at as an instance of what is sometimes referred to as the Hollywood principle: *Don't call us, we'll call you* and saves valuable resources in terms of communication costs.

Concurrent Stalmarck A novel algorithm, *Concurrent Stalmarck*, has been developed by applying concurrent techniques to the Stalmarck algorithm, such that the new algorithm is well-placed to use large scale parallel processing resources, such as a cluster of workstations. It can be used as a tautology checker or as a clause-learning process. The use of concurrency techniques has been done to enable harnessing large scale parallel processing resources or to deploy on massively parallel machines or a huge cluster. We have implemented work stealing using the *incremental evaluation* feature facilitated by Alice ML. This eliminates the need for costly communication to achieve load balancing. This prototype is a piece of exploratory research and illustrates the scope of applying parallelisation to an existing algorithm to synthesise a new algorithm that is well placed to utilise emerging architectures and novel computing patterns.

- Proof-of-concept prototypes implementing these approaches were described.
- Empirical results conducted using these prototypes was provided, with an analysis of the data. Performance gains were observed in two of the three problem classes considered for the hybrid solver implementation. Empirical results of the concurrent Stalmarck implementation also exhibited significant performance gains over the sequential implementation.
- Programming abstractions (developed as higher-order functions in Alice ML), encapsulating the concurrent approaches employed in the prototypes that have been developed, were described. The implementations discussed in this chapter serve as an illustration of the developmental claim of this thesis as outlined in §3.1, of the utility of a functional programming language for implementation and the advantages of using programming abstractions for implementation. A discussion of how these criteria have been met is given in §6.5.6 and §6.7.3.
- The utility of the abstractions was concretely demonstrated by using the abstractions to develop new hybrid solvers.

The DPLL algorithm is a tightly coupled algorithm and as such traditional work partitioning approaches have limited applicability. Optimal work partitioning and load-balancing are crucial for effective utilisation of distributed architectures. However, hardness characterisations of SAT problems are difficult and search spaces are highly irregular. These make optimal work partitioning and load-balancing very difficult to

achieve. The Concurrent Stalmarck algorithm that we have developed effectively address these issues. Furthermore, they can be used in conjunction with the hybrid solver architecture, as demonstrated in the DPLL-ConcurrentStalmarck prototype.

DPLL-based SAT solvers have evolved over a considerable time period and highly optimised data structures and implementations for these solvers have been developed, making them very successful in handling big and complex problems. Thus, it makes sense to use the advanced technology available for these solvers. The hybrid solver architecture that we have demonstrated, via the *doDPLLwithHelper* abstraction, opens up opportunities for using the state-of-the-art in DPLL-based SAT solvers, along with helpers, who act as providers of learned clauses. The helper(s) can be chosen, such that:

- it addresses the limitations of DPLL, such as loss of implicit structural information;
- its learning mechanism uses an alternate, complementary viewpoint of the problem i.e. an alternate, complementary search approach, compared to that of DPLL;
- it can run autonomously, independent of the DPLL process, thus preventing bottlenecks and optimally utilise multiple workstations;
- it is known to perform well on the given problem class, while focusing on a particular problem class.

Most of the parallel SAT solvers are also DPLL-based and use guiding-path for work partitioning and distributed clause learning (sharing the clauses learnt by CDCL). However, by design, the CDCL clause-learning technique, is embedded within the DPLL algorithm and so,

- the clause learning happens by spanning the search space in the same way, as the DPLL process;
- it suffers from loss of implicit structural information (inherited from the DPLL algorithm);
- it cannot be decoupled from the DPLL process and so, cannot work independently, as a clause learner.

The Stalmarck clause learner that we have used, addresses all these points, as

- the clause learning happens by spanning the search space in a breadth-first fashion, complementary to that of DPLL;
- it leverages on the structure of a formula;
- it is independent of the DPLL algorithm and can thus work independently, as a clause learner.

The hybrid solver architecture that we have demonstrated here can enable use of the Stalmarck clause learner along with an optimised DPLL (DPLL+CDCL) solver.

In the next section, we outline some future research possibilities, following on from the work described in this chapter.

,

6.10 Future research

We have explored a wide topic in this thesis, by addressing developmental methodology and object-level opportunities for two diverse case studies of theorem proving. Though we have achieved our initial goals of investigation, in course of the work reported here, we have come across more opportunities that merit further investigation. We could not follow up on these, due to insufficient time and/or not falling within the scope of this work. We enumerate some such opportunities, which are future research possibilities, building on the work discussed here:

1. Port the cooperative framework of DPLL-Stalmarck to other platforms, e.g. C-like platforms.
2. DPLL's search space is pruned by Stalmarck's results as shown in our work. The key to better performance is to get Stalmarck to produce information relevant to the DPLL at a faster rate. There are a few ways to do it. The obvious way is to use massively parallel machines. A complementary approach can be in the direction of using efficient data structures like a BitArray and bit-operations to perform the unit task.
3. Run on a wider variety of benchmarks, with particular emphasis on domain-specificity, in order to study the ones that will gain most from the DPLL-Stalmarck architecture.

4. Study the sensitivity of branching-heuristics in relation to the use of information supplied by the Stalmarck process.

5. Come up with new heuristics that will adjust according to the information supplied by the Stalmarck process.

6. Run experiments to study the relation between the number of Stalmarck agents and the performance.

7. Study phase-transition behaviour for the hybrid solver.

8. Stalmarck's algorithm's notions of proof hardness can inform the choice of recursion depth and allocation of resources to the solver for a given problem. Automate exploitation of this aspect.

9. Combination of complete solvers and incomplete solvers, using the DPLL-with-a-helper abstraction e.g.. In the work described in this thesis, we have used the Stalmarck solver primarily as a learning mechanism: as an information provider rather than as a solver by itself. If the strength of probabilistic and other incomplete solvers can be used in a similar way, they can be used as the *helper* in our abstraction. However, we have not explored this possibility in detail and so are not aware of the limitations or potential opportunities that such an architecture would entail.

10. In practice, tractability for sequential Stalmarck solvers is restricted by the hardnesss criteria and a value of two is captured by the maxim *if it is 2-hard, it is too hard* [Harrison, 1996]. A distributed approach as implemented in our concurrent Stalmarck solver can significantly help here as more work can now be handled thus giving scope for improving the tractability threshold. However, we have not tested these possibilities empirically. This is an option for future work.

11. The utility of the information from a helper and the tradeoffs of the utility vs overheads is a topic that needs to be investigated more closely. A rigorous analysis of the same, possibly matching problems with a helper (as done in portfolio methods in SAT e.g. [Hamadi et al., 2009]) can greatly benefit the implementation.

12. Exploration of the utility of a shared memory model to enable the DPLL process to immediately absorb the information from the helper.

13. Similarities of *saturation_abstraction* and the *barrier* abstraction were explained in §6.5.6. The *barrier* abstraction is quite popular and is provided as an optimised library implementation in many concurrent programming languages. Thus, this similarity can be utilised to open potential opportunities for porting this implementation to another platform to allow for utilisation of such optimised implementations.

14. As explained in §5.4.2.4, the *Map Reduce* abstraction is becoming increasingly popular and optimised implementations of the same are widely available for using on a variety of distributed architectures. In §6.5.6, we explained how the deduction process performed by an agent can be viewed as a Map-Reduce operation by giving the *map* and *reduce* operations for our implementation. This similarity can be exploited further to open potential opportunities for porting this implementation to other platforms using optimised implementations of the Map Reduce abstraction.

Chapter 7

Developing concurrent, programmable, sound extensions, for an LCF style theorem prover

7.1 Introduction

As explained in §3.1 and §5.3.1, in this thesis, for each case study, we have explored the following two strands of investigation:

Object-level aspects: previously unexplored or little-explored ways of using concurrent/distributed techniques for the particular theorem proving flavour considered in the case study

Developmental aspects: developmental effort required, ease of prototyping and experimentation, scope for exploratory investigations, incremental development and portability

In chapter 6, we saw the scope and efficacy of employing concurrency, distribution and synergetic use of complementary reasoning systems within a propositional satisfiability (SAT) solver. This is representative of a decision procedure for arguably the simplest logic with the employed techniques and algorithms falling into the category of fully automatic theorem provers. In this chapter, we consider a paradigmatic case of an LCF style first-order theorem prover, to explore the two strands of investigation mentioned above.

As discussed in §4.4.6, LCF style theorem provers have the following key distinguishing aspects:

- Small trusted kernel
- Proof as an abstract type
- Programmability
- Interaction via a functional programming language

There are arguably many possible approaches to applying concurrent techniques to an LCF style theorem prover. A survey of related published research was provided in chapter 2. The vast majority of these works focus on approaches using a coarse-grained, heterogeneous combination of provers and other systems, e.g., use of first-order theorem provers to tackle higher-order problems. While these approaches have their own purpose and merits, we have explored an orthogonal approach, to investigate the opportunities of applying concurrent approaches to engineer novel proof search procedures, in the first-order LCF context. We have focused on the following:

- providing the user with the tools to *program* their own concurrent proof search procedure(s), with minimal developmental effort;
- enable easy set up of experiments, comparing different concurrent approaches;
- encapsulate the concurrent techniques employed, thus separating design and implementation, and facilitating porting of the design to other LCF provers.

Theorem proving problem classes originate from a variety of domains and can vary greatly in problem structure, proof hardness, solution distribution etc.. Even within the same formalism, and for the same prover, the computational challenges presented by a problem class/instance can be vastly different. For example, in the first-order LCF context, a problem instance can pose a challenge,

1. by generating a significantly large number of independent sub-goals, each of which needs to be proved;
2. in the form of instantiation of shared meta-variables in sub-goals and management of the consequent interdependencies.
3. which is related to quantifiers: possibility a non-deterministic choice of inference-rule application, where particular choice(s) lead to non-terminating search(s).

The choice of appropriate concurrent programming technique(s), to tackle these scenarios can be possibly different and a concurrent approach that benefits one scenario need not necessarily benefit another. For example, 1 mentioned above, calls for an approach using multiple processes working on each of the independent sub-goals, an instance of AND-parallelism. Interprocess communication is not required in this scenario, whereas, it is necessary to effectively handle 2.

This diversity calls for a *programmable* approach to apply concurrent techniques, ensuring that the soundness and interactive aspects that distinguish LCF provers, are retained as well. By *programmable*, we mean the following: the user can use the concurrent primitives and control structures provided in the system and adapt and extend (i.e. *program*) them to code their own concurrent proof search procedure.

Such an approach can empower the user with the ability to choose the appropriate concurrent techniques for the scenario/problem at hand. Enabling the users to program their extensions, incorporating concurrent techniques and leveraging on the in-built soundness guarantees of the LCF approach, is very much in alignment with the fundamental motivation of the LCF style theorem provers, particularly, the programmability and interactive aspects.

To address these objectives, we have developed a multilayered approach to implement sound and programmable extensions to an LCF prover, such that they incorporate concurrent programming techniques. As an aid to our investigation, we have used a sequent-calculus based prototype LCF style prover for classical first-order logic without equality, *HAL* [Paulson, 1996][1], (described in detail in §7.3.1 and §7.5).

HAL is designed for a sequential mode of execution. We have used it as our baseline system, to develop a prototype that implements our multilayered approach and thus allows for programmable, sound extensions, incorporating concurrent and distributed programming techniques.

[1] HAL is not an acronym and as such carries no special meaning!

In the rest of this chapter, we describe the following:

1. The multilayered approach that we have developed, in order to facilitate sound, programmable extensions to an LCF style theorem prover, incorporating concurrent and parallel approaches.
2. How the HAL system serves as a representative prototype for our investigation and the design details of the HAL system. Also a summary of the problems highlighted by our efforts to port Isabelle 2006 to Alice ML, primarily related to the non-functional aspects present in Isabelle's design.
3. The new concurrent tacticals developed, aimed at addressing the limitations of their sequential counterparts in HAL.
4. A novel approach to implement unification, using multiple asynchronous processes and exchange of (partially) evaluated information.
5. A discussion comparing our approach to other related work.
6. An analysis of the related Isabelle-PolyML project, drawing out similarities and differences with our work.
7. Examples illustrating the utility of the concurrent tacticals provided in our prototype by demonstrating the ease of programming novel concurrent proof search procedures, tailored to individual problem scenarios.
8. The utility of a multilayered approach for ease of development, experimentation, incremental development and programmability. An outline of how our prototype serves as a proof-of-concept of the multilayered approach and how the approach can be applied to other LCF provers as well.

7.2 Multilayered approach to apply concurrency and distribution techniques, to an LCF style theorem prover

In this section, we describe the multilayered approach that we have developed to employ concurrent techniques in an LCF style theorem prover and a proof-of-concept prototype implementing this approach, implemented in Alice ML. We end the section, with a summary of the advantages of our approach.

7.2.1 Developing programmable, concurrent, sound extensions, for LCF provers: A multilayered approach

An LCF style theorem prover has a trusted kernel of tactics (primitive inference rules). Tacticals, control structures to apply the tactics, are provided. These tacticals and tactics are used to synthesise proof search procedures. This guarantees the soundness of proofs derived by the system, among other advantages. The kernel can serve as a good place to introduce a layer of concurrent and co-routining control structures for applying tactics. If the concurrency techniques used retain and adhere to the type-inference properties, this approach will ensure that

- the soundness properties are carried forward, by virtue of the LCF approach's treatment of treating theorem as an abstract type, whose only constructors are the sound inference rules from the trusted kernel;
- the concurrent control structures for applying tactics are available for use at the top-level and for use in proof search procedures (in the same way as the sequential ones).

We have adopted this treatment in the multilayered approach that we have developed: by working from the programming language to the kernel of the theorem prover. The framework thus engineered allows for programmable, sound extensions to an LCF prover, incorporating concurrent techniques.

The gist of our approach is as follows:

Concurrent techniques implemented as programming abstractions: We implement the relevant concurrent programming techniques as programming abstractions with appropriate parametrisations. For example, encapsulate the computational pattern used to return the fastest returning function from a list of functions, as a programming abstraction, (see §7.6.2). Another example is an abstraction for computing a consensus (see §7.8.3.1).

Concurrent tacticals, engineered via the abstractions: Using the tactics present in the LCF system and the concurrent programming abstractions developed (as above), we develop a suite of *concurrent tacticals*: control structures incorporating concurrent techniques, for applying the tactics. These allow the user to apply tactics using a variety of concurrent techniques. They can also be used for incremental development of new tacticals and proof search procedures, employing concurrent techniques. These are described in §7.6. A point worth observing here, is that, the concurrent tacticals, have the same functional behaviour (i.e. the same type) as sequential tacticals and can hence be used in the same way as sequential tacticals, e.g. composing one sequential tactical with a concurrent tactical.

Concurrent proof search procedures, engineered via the concurrent tacticals: We use the concurrent tacticals and abstractions to implement established proof search procedures and design new ones employing concurrent techniques in a gainful manner to accomplish proof search. These are described in §7.9.2 and §7.9.1.

At the start of this research project in 2006, within the realm of application of parallelisation to the LCF style of theorem proving, this direction of research had not been reported in any published research, to the best of our knowledge.

An early paper outlining the ideas of this approach and discussing the advantages of a multilayered approach as opposed to an adhoc MPI style approach, was presented in an Isabelle workshop in 2007 [Sripriya et al., 2007]. Subsequently, the utility of the approach has been taken on board by the Isabelle developers. They have since invested a lot of efforts to provide concurrency and parallelisation support in Isabelle and have recently published some early results on the same [Matthews and Wenzel, 2010; Wenzel, 2009]. This work is discussed in detail, later in the chapter, in §7.4. It

is useful to note that this work required considerable reworking of Isabelle's bootstrapping process and the ML interaction mode, addressing the non-functional aspects. It also required considerable, fundamental modifications to the PolyML platform, to accommodate concurrency support. Furthermore, the project required a concerted effort of two years, by one of the key Isabelle developers and an ML expert (PolyML).

7.2.2 Proof of concept

We have developed a proof-of-concept prototype of this multilayered approach, applied to an LCF style, sequent-calculus-based, first-order prover (described in detail in §7.3.1 and §7.5). HAL has been used as the baseline sequential system and has been ported to Alice ML. In §7.3.1, we describe the rationale behind the choice of HAL as the baseline system for our prototype.

Abstractions The programming abstractions layer has been developed in Alice ML, as a collection of higher-order functions, with appropriate parametrisations. In developing the abstractions, our focus has been to enable the use of asynchronous processing and co-routining techniques. These enable the realisation of collaborative problem solving approaches, even at the term level, e.g. as implemented in our collaborative unification tactic.

Concurrent tacticals The existing tactic base of HAL has been used along with the concurrent programming abstractions, to implement novel control structures for applying tactics. We have implemented a novel collaborative unification tactic, using the power of collaborative exchange of partially evaluated information.

Programming concurrent proof search procedures We have implemented a novel proof search procedure, based on the depth-first approach, using asynchronous operations and the collaborative unification tactic. Examples are provided, illustrating the potential of using concurrent techniques to engineer a proof search procedure to address specific scenarios. These demonstrate the utility of our approach, as it illustrates the ease of *programming* a search procedure, incorporating concurrent techniques, using the concurrent tacticals and other primitives provided.

Using programming abstractions to implement concurrent programming is advocated strongly within the parallel programming community [Asanovic et al., 2006]. The use

of abstractions is considered advantageous from a software engineering perspective, as it promotes modularity, code reuse, portability etc..

The requirements for implementation of our multilayered approach are as follows:

- The LCF system should be ported to a functional language that supports concurrency and distribution, preferably in a language-integrated manner, as described in §5.5.2.

- The design of the LCF system should be free of side-effects-based, imperative-style programming or should abstract them.

7.2.3 Advantages of our proposed multilayered approach

Below, we summarise the advantages of the multilayered approach that we have developed and implemented:

- The approach accomplishes an effective separation of design and implementation. So, once a suite of concurrent tacticals has been found to be useful, it can be *ported* to (i.e. replicated in) other eligible LCF systems. We use the term *portability* in this sense.

- The approach also allows for the concurrent-distributed features developed to be made available for interactive use, as they adhere to the definition of tactics and tacticals.

- As mentioned before, due to the diversity in the problem classes and their related computational challenges, a one-solution-fits-all approach may not always work, for the effective application of concurrent programming techniques to engineer better theorem provers. Our multilayered approach enables the provision of an experimental workbench based on a given LCF style theorem prover. The workbench can in turn, be used to quickly prototype experimental techniques incorporating concurrency in order to develop novel proof search procedures.

7.3 HAL as a representative prototype

In this section, we provide an overview of the HAL system and the rationale behind our choice of HAL, as an exemplar system, for developing our prototype.

7.3.1 About HAL

HAL is a sequent-calculus based first-order theorem prover. It is meant as an illustrative prototype and is described in the functional programming language textbook, entitled, *ML for the working programmer* [Paulson, 1996]. The code for HAL, written in ML, is provided along with the book and is available from the web[2]. We ported the code to Alice ML and used it as the sequential baseline system to implement our multilayered approach, introducing sound, programmable extensions to HAL. Design details of HAL are described later, in §7.5.

HAL constructs proofs by refinement steps, working backwards from a goal. At each step, an inference rule is matched to a goal, reducing it to subgoals. HAL implements *sequent calculus*, as a set of transformations, on an abstract type of proof states. Each inference rule is provided as a *tactic*. A basic user interface allows the tactics to be executed. In general, tactic-based theorem provers (see §4.4.6 for definitions) allow a mixture of automatic and interactive working. To provide more automation, HAL provides a collection of *tacticals*, i.e. control structures to apply tactics. These can be used to code (semi-)automatic proof search procedures for first-order logic.

7.3.2 Why HAL ?

The objective of development of the prototype described in this work has been:

- To provide a proof-of-concept prototype, implementing our multilayered approach.
- Allow sound, programmable extensions.
- Act as a workbench to prototype and experiment with synthesising concurrent proof search procedures, in the LCF, first-order context.

[2] http://www.cl.cam.ac.uk/ lp15/MLbook/programs/

Our multilayered approach fits well with an ML-based interaction mode of an LCF prover. So, GUI and additional layers that obscure the ML-based interaction with the LCF kernel, were not needed and particularly when it came at the cost of a complicated system. Thus, an LCF style system written in an ML dialect and allowing for interaction with the kernel via ML, was an essential requirement.

Like development of any prototype system, the choice of HAL, has been motivated by implementation considerations as well. We started off our implementation efforts, by trying to use Isabelle to implement our prototype. In §7.4, we describe the challenges encountered in porting Isabelle to Alice ML. Most of these were due to Isabelle's bootstrapping procedures, the non-functional aspects present in the system and reliance on non-standard features of SML like the availability of the function *use* to build the necessary bindings. These were not compatible with Alice ML's fully modular approach using *components* (see §5.6). Furthermore, Isabelle has evolved over two decades and has become a very complicated system to base a prototype on and as we discovered during our efforts, many non-functional aspects have crept into the system. Also, with the recent releases of Isabelle relying almost entirely on the proof script *Isar* mode, ML-based interaction, in the original LCF sense, has become almost obsolete and is not adequately supported.

To summarise, HAL serves as an ideal vehicle to base our prototype on, for the following reasons:

- It has a reasonably small and self-contained code kernel (written in SML [Milner et al., 1997]), a desirable feature, from a prototype developmental perspective.
- Is still powerful enough to realise our objective (as given in 1) for the LCF style of theorem proving in a first-order setting.
- We successfully ported HAL to Alice ML, our implementation language, unlike our efforts to port Isabelle to Alice ML (described in §7.4).

7.4 Porting Isabelle to Alice ML

In this section, we report our efforts to port Isabelle to Alice ML which were not completely successful and briefly mention the reasons for the same. Also described are the issues in Isabelle's bootstrapping process that were highlighted in the course of

our efforts.

At the time of the start of the development of the prototypes described in this thesis, we considered Isabelle as a possible vehicle to base our prototype on. However, at that time, Isabelle did not have any provisions for parallel support. As discussed earlier in the thesis, our rationale behind the choice of Alice ML as the implementation language has been in alignment with the hypothesis that we are investigating: light weight threads; message passing, concurrent and distribution possibilities offered; an SML based language; ease of developing programming abstractions as higher-order functions.

We spent a significant amount of time to port Isabelle to Alice ML which turned out to be not completely successful [3]. Our efforts highlighted certain aspects of the Isabelle system that were not purely functional, particularly in the way the logics are built. Our work helped to initiate many changes that have since been made (2006-09) to the Isabelle architecture and the bootstrapping process. We enumerate some of the key issues below:

- We successfully dealt with a fair amount of incompatibilities in the form of libraries and some quirks related to Alice ML. But, there were non-functional aspects related to the Isabelle architecture, in particular the bootstrapping process, which made the porting task unsuccessful. The bootstrapping phase relies heavily on two non-standard artefacts: the notion of a *heap*, a non-standard feature used in many ML dialects, and a top-level function called *use*. It is useful to point out here that *use* is not part of the SML definition and is not implemented in Alice ML. A *heap* refers to a dumped image of the ML top-level environment and holds all the bindings, evaluations and declarations created till that point. All this is within the same ML process. In Isabelle, to build a logic, the corresponding heap is in turn, built using nested applications of the *use* function for loading bindings and declarations. This severely compromises the modularity of the system. We experimented with some ways around to address this by using the Alice ML *component* system (Alice ML views programs as modular units, called *components*), which were not completely successful. Alice ML adopts a lazy approach to loading the components. This conflicts with the expected be-

[3] We used Isabelle 2006 for our porting efforts.

haviour of *use* in its nested application in Isabelle's bootstrapping stage. But, the dependencies and the bootstrapping process were too arbitrary to manage. However, using Alice ML's component system is still a promising option to follow through.

- The Isar proofscript environment with all its advantages of user-friendliness for interactive proof has also made the ML-level usage with the tactics less accessible, thus reducing the programmability of the system. Furthermore, the introduction of the *Isar* layer has introduced many modifications in the system architecture. The documentation for these was sparse at the time of our experiments.

- Isabelle has evolved over 20 years, and with the additional Isar layer, it has become too complex a system to tackle for the purpose of re-organising the architecture. Furthermore, with the Isar layer, the ML level usage with the tactics is less accessible. As we learnt in course of our efforts of porting Isabelle to Alice ML, such an enterprise requires non-trivial amounts of work requiring fairly in-depth knowledge of the internals of the implementation language and the theorem prover. Furthermore, the ML modules may depend on previous definitions and proofs produced at runtime. This bootstrapping process never stops, although some end-users may have the illusion that the environment distributed as "Isabelle/HOL" is something like a finished program.

Initially the use of OS level POSIX-threads and forking Isabelle processes was suggested by one of the Isabelle architects as a route to do parallel theorem proving [4]. The OS level approach would also limit the granularity at which the concurrent/parallel features can be used. E.g., using these features at a term level will be impossible using this approach. The importance, potential and advantages of using the lightweight threads approach as implemented in Alice ML was presented in an Isabelle workshop by the author in 2007 [Sripriya et al., 2007] and subsequently the efficacy of such an approach has been taken on board fully by the Isabelle developers. However, the considerations of a more robust language of production quality, of which PolyML is supposed to be one, along with the fact that Isabelle works best on the PolyML platform, among all dialects of ML, led the Isabelle developers to venture on their own project providing parallelisation support in Isabelle via PolyML [5]. This is an ongoing project since 2007

[4] Email communication: Larry Paulson, June 2006

[5] Personal email communication with Makarius Wenzel, Tech Univ of Munich, one of the key Is-

and addresses the development of features like futures (that are already supported by Alice ML) in PolyML [Matthews and Wenzel, 2010]. This is discussed in more detail in §7.11.2.

Our presentation of our approach at the Isabelle workshop also led to a useful collaboration with the Isabelle developers to facilitate changes in the Isabelle architecture to address the issues that we had identified during our efforts to port Isabelle to Alice ML. Some of these were fixed by the Isabelle developers and Isabelle's *Pure* kernel can now be ported to Alice ML. However, the implementation of the changes was too late for our project. Hence, we switched to work on the HAL system which is described in the next section.In §7.3.2, we described our rationale for this choice. Furthermore, porting Isabelle to Alice ML (despite the modifications done to Isabelle's bootstrapping issues highlighted by us) may require considerable effort, given the way that the logics are built in Isabelle, as an ongoing buildup of bindings on the top-level environment (with respect to Isabelle 2009).

7.5 Design overview of the HAL system

In this section, we describe the implementation details of HAL, required to understand the material discussed in this chapter. More details can be found in [Paulson, 1996]. The system is coded in Standard ML (SML) and should be portable to any SML dialect. For definitions and the background of sequent calculus, LCF style theorem proving and the notions of goal, proof state, tactics and tacticals, the reader is referred to §4.4.2 and §4.4.6. Table 4.3 gives a list of all the sequent calculus rules relevant for our purpose.

7.5.1 Data structures, treatment of bound variables and meta-variables, enforcement of quantifier-rule-provisos

Term: is defined as a datatype, as follows:

```
datatype term = Unknown
  | Var of string | Param of string * string list
  | Bound of int
  | Fun of string * term list
```

abelle/Isar architects

Formula: is defined as a datatype, as follows:

```
datatype form = Pred of string * term list
  | Conn of string * form list
  | Quant of string * string * form
```

Name-free representation of bound variables

HAL adopts a *name-free representation* of bound variables, with origins in a $\lambda-$calculus-based representation. Operations such as abstraction and substitution are easily performed in the name-free representation. We give a brief account of this treatment here, more details can be found in [Paulson, 1996, pg.376].

The name x of a bound variable serves only to match each occurrence of x, with its binding, so that reductions can be performed correctly. If these matches can be made by other means, then the names can be abolished. This can be achieved using the nesting depth of abstractions. Each occurrence of a bound variable is represented by an index (*de Bruijn* indices), giving the number of abstractions lying between it and its binding abstraction. E.g., the term,

$$\lambda x.x(\lambda y.xy(\lambda z.xyz))$$

can be represented, using the name-free notation as follows

$$\lambda.0(\lambda.10(\lambda.210))$$

Meta-variables:

As seen in §4.7.1, a *meta-variable* serves as a convenient device to handle the pending (yet-to-be-instantiated) status of a variable during the course of a proof. In HAL, a meta-variable is denoted with a leading *?* symbol, e.g. *?a*.

Enforcing provisos in quantifier rules:

The sequent rules $\forall : left$ and $\exists : right$ (see Table 4.3) impose the proviso that *x must not occur free in the conclusion*. When the conclusion contains meta-variables, additional machinery is required to enforce this proviso. In HAL, this is taken care of by labeling each *free variable* with a list of *forbidden meta-variables*. Thus, to express the condition that the free variable, b, must not be contained in a term substituted for the meta-variables $?a_1, \ldots, ?a_k$, the following notation is used:

$$b_{?a_1, \ldots, ?a_k}$$

. In other words, $b_{?a_1,\ldots,?a_k}$ says that any instantiation of the meta-variables in $?a_1, \ldots, ?a_k$ should not contain a free occurrence of b.

On an implementation level, this is enforced by maintaining a list of the forbidden meta-variables (which is empty to start with) for every free variable [6].

The unification algorithm implementation used in HAL, builds and uses this information while computing the possible compatible substitutions that will make the given pair of terms identical.

Unsafe quantifier rules: $\forall : left$, $\exists : right$**:** Table 4.3 gives a list of all the sequent calculus rules relevant for our purpose. $\forall : left$, *and* $\exists : right$ have one feature that is not present in any of the other rules. In backward proof, they do not remove any formulae from the goal. They expand a quantified formula, substituting a term into its body; and retain the formula to allow repeated expansion. It is impossible to determine in advance how many expansions of a quantified formula are required for a proof, a consequence of the undecidability of provability in FOL [7]. In this work, we refer to these two rules as, *unsafe* quantifier rules.

Sequent, Goal A *sequent* $\phi_1, \ldots, \phi_n \vdash \gamma_1, \ldots, \gamma_n$ is represented as a pair of formula lists. A *goal*, which in turn, is a sequent, is represented as a pair of formula lists, as follows:

```
type goal = form list * form list
```

Proof state: is represented by the tuple,

$$(sgList, g, i),$$

where, *sgList*: list of sub-goals, g: main goal, i: a number, used to generate fresh variables. The sub-goals are in turn, represented as sequents.

```
datatype state = State of Fol.goal list * Fol.form * int;
```

Data structure used to implement a sequence of proof states:

[6] An alternative to capture variables appearing in quantifiers and the proviso related issues described above is using the notion of *skolem functions*. Roughly speaking, $b_{?a_1,\ldots,?a_k}$ is treated as a (*skolem*) function and it is treated as a term for the rest of the proof. Use of *skolem functions* is very popular particularly in the engineering of automatic theorem provers. But, it is has the disadvantage of destroying the readability of the formula. Any standard text should provide details on this topic.

[7] This can cause a proof procedure to fail to terminate; first-order logic is undecidable.

HAL uses a data structure called *Seq* to implement an *unbounded list*, with the intended behaviour, as a *lazy* list [8].

The original code of HAL, uses an imperative implementation of the same, using references. We have modified this to a purely functional implementation and have used the *lazy* option of Alice ML to take care of the lazy aspect of the evaluation. The ML type signature is given in Listing 7.1. Note that in a sequential execution mode, lazy evaluation is the only option to achieve the desired behaviour of an unbounded list, while adhering to a purely functional behaviour. For the rest of this chapter, we use the term *sequence*, to refer to this data structure.

Listing 7.1: Code for the Sequence structure in HAL

```
signature SEQUENCE =
sig
  type 'a t; exception Empty;
  val empty: 'a t ; val cons: 'a * 'a t ->'a t
  val toList: 'a t -> 'a list ; val fromList: 'a list -> 'a t
end
structure Seq :> SEQUENCE =
struct
  datatype 'a t = Nil | Cons of 'a * 'a t; exception Empty;
  val empty = Nil; fun cons(x,xf) = Cons(x, xf) (*constructors*)
  fun lazy toList Nil = [] | toList (Cons(x,xp)) = x :: toList  xp
  fun lazy fromList [] = Nil | fromList (x::xs) = cons(x, fromList xs)
end
```

Demand-driven behaviour of lazy evaluation As described in detail in §5.5.2, the use of lazy lists results in a *demand-driven* consumer-producer computation model, where the producer produces data only if the consumer requests for it. As we will see later in §7.6.1, this model is in contrast to a *data-driven* producer-consumer pattern, that can be implemented using asynchronous computation.

Tactics in HAL: An LCF tactic represents a partial proof and is more commonly represented, by a function of type **thm** list $\rightarrow$ **thm**. HAL differs from this practice and implements the inference rules as functions on proof states, instead of functions on theorems.

A *tactic* takes a proof state(s) and a number(i) as input and returns a *sequence* (i.e. a lazy list) of states, the result of applying the tactic on the i^{th} subgoal of the

[8] We explain this in detail here, as this is used heavily, in the subsequent sections, where we describe the new concurrent tacticals that we have developed.

given proof state *s*.

```
type tactic = state -> state Seq.t
```

Application of a tactic: If it can be successfully applied, a tactic returns a *sequence* of next possible states, else it returns an empty sequence, which denotes a *failure*.

Basic tactics provided in HAL:

Primitive inference rules All primitive inference rules of sequent calculus for first-order logic without equality, are implemented as *tactics*, with self-explanatory names as follows: *conjL, conjR, disjL, disjR, impL, impR, negL, negR, iffL, iffR, allL, allR, exL, exR*. Note that we use these names to refer to these tactics and their respective inference rules (rather than the symbol-based names in Table 4.3), for the rest of this thesis.

basic Applying *basic* checks if a subgoal is a basic sequent (a sequent is called *basic* if both sides share a common formula; such sequents are clearly valid.); if it is, then, it is removed from the sub-goal list, else, the tactic-application is considered to have failed (i.e. an empty sequence is returned).

unify Unification is implemented as a tactic (described in §7.5.3). Calling *unify* attempts to solve a subgoal by converting it into a basic sequent

Safe tactics, unsafe tactics Corresponding to the notion of unsafe rules, we refer to a tactic which involves variable instantiation or unification as an *unsafe* tactic and a *safe* tactic otherwise. The order of application of the *safe* tactics do not matter. Thus, for the basic tactics provided in HAL, the

Safe tactics are: basic, conjL, conjR, disjL, disjR, impL, impR, negL, negR, iffL, iffR, allR, exL; and

Unsafe tactics are: allL, exR, unify

Other functions:

by The function *by* takes a tactic as argument, applies the tactic on the current proof state and updates the current proof state, with the first element of the sequence of states returned by the tactic. If the tactic fails, it prints an appropriate message.

initial For a goal p, calling *initial* p, creates a state containing the sequent $\vdash p$ as its only subgoal, p as main goal and 0 as variable counter.

final Given a state, the predicate *final*, tests for an empty subgoal list.

Tacticals: HAL has a suite of basic tacticals (implemented as higher-order functions) to apply tactics, which are described in the next section. The implementation of tacticals has been realised, via generic operators designed for the *Seq* structure (described earlier in Seq structure).

7.5.2 Basic sequential tacticals in HAL

Tacticals are control structures, which apply tactics in different ways, e.g. choice operator, composition operator. As explained before, a HAL tactic uses the Seq structure to return the sequence of next-proof-states. The control structures for applying the tactics, in turn, are defined via corresponding operations of the the Seq structure. In this section, we describe some of the basic sequential tacticals in HAL, with code-fragments related to their implementation.

-- , composition operator: Applies tac_1, followed by tac_2. For a given proof state, x and tactics, tac_1, tac_2, (tac_1 **--** tac_2) x gives the sequence of sequences:

$$tac_2(y_1),\ tac_2(y_2),\ \ldots,\ \text{where,}$$
$$tac_1 x\ =\ y_1, y_2,\ \ldots,$$

The operator returns the concatenation of all the individual sequences. It is worth observing here that each individual sequence, $tac_2(\ y_i)$ incorporates lazy evaluation. Thus, though the sequences are concatenated, because of the lazy aspect of the evaluation, the actual evaluation is triggered only when the result is demanded by another operation.

```
fun tac1 -- tac2 x = Seq.concat (Seq.map tac2 (tac1 x))
```

***all*, identity operator for --:** Accepts all states unchanged, returns a singleton sequence containing the given state

```
fun all x = Seq.fromList [x]
```

||, choice operator: Commits to the first successful tactic with no backtracking. For a given proof state, x and tactics, tac_1, tac_2, $(tac_1 \mid\mid tac_2)\ x$ does the following: if $(tac_1\ x)$ is non-empty, then returns $(tac_1\ x)$, else returns $(tac_2\ x)$

```
fun (tac1 || tac2) x = let
  val y = tac1 x
in if Seq.null y then tac2 x else y end;
```

|@|, less-committal form of choice operator: Combines the results of two tactics with backtracking. For a given proof state, x and tactics, tac_1, (tac1 |@| tac2) x returns the concatenation of the sequences (tac1 x) and (tac2 x). However, due to the lazy evaluation aspect, evaluation of tac2 is triggered only when the computation of tac1 is completed.

```
fun (tac1 |@| tac2) x = Seq.concat(Seq.cons(tac1 x,
 (*delay application of tac2!*)
Seq.cons(tac2 x, Seq.empty)))
```

***no*, identity operator for || and |@| :** Returns the empty Sequence, for all cases.

```
fun no x = Seq.empty
```

try: For a given tactic tac_1, *try* attempts to apply tac_1

```
fun try tac = tac || all;
```

***repeat*: repetition operator:** For a given proof state, x and tactic tac_1, the result of (repeat tac1 x) is the sequence of values obtained by repeated applying tac1 until a further application of tac1 would fail.

```
  (*Performs no backtracking: quits when stuck*)
  fun repeat tac x = (tac --- repeat tac || all) x;
```

***repeatDeterm*: deterministic repetition operator:** Considers only the first outcome returned at each step

```
(*Repetition, considering only the 1st outcome*)
fun repeatDeterm tac x = let  fun drep x = drep (Seq.hd (tac x))
            handle Seq.Empty => x
in  Seq.fromList [drep x]  end;
```

***sleepTactic*, a tactic for suspending the thread for a given duration:** For a given proof state, x and tactic, tac_1, and time (in seconds), n, *sleepTactic* n tac_1 does the following: it suspends the executing thread for n seconds; resumes execution, returning the result of the application of tac_1 on x. We added this tactic, purely for illustrative purposes, to mimic a time-consuming tactic. Some of the examples provided in subsequent sections, have been formulated using this tactic.

```
fun sleepForNSeconds n = Thread.sleep(Time.fromSeconds (IntInf.fromInt n));
fun sleepTactic n tac st = (sleepForNSeconds n; (tac st))
```

7.5.3 Unification as a tactic in HAL

HAL provides a tactic called *unify* for applying unification on a goal (a sequent). A general overview of the unification procedure was given in §4.7 and definitions of terms used in this section can be found in §4.4 and §4.7. In this section, we provide a description of the implementation of the *unify* tactic in the HAL system. In §7.8.3, we discuss the limitations of this implementation and how the use of concurrent programming techniques can address the same.

Unification algorithm used The unification algorithm used takes terms containing no bound variables. For a given pair of such terms, the algorithm computes a set of possible (variable,term) substitutions to make the given pair of terms identical and reports that the terms cannot be unified when a suitable substitution cannot be found. The pseudocode for the algorithm is given in Listing 7.2. The implementation of unification in HAL is purely functional. It uses a recursive version of the algorithm and incorporates the *occurs check*: If *?a* occurs in t, then the equation has no solution, for no term can properly contain itself. HAL provides the *unify* tactic to apply unification on a given goal. See Listing 7.3, for the code for the same.

Behaviour of the *unify* tactic:

- The *unify* tactic takes the following as arguments: (i, st), an integer and a proof state respectively.
- It attempts to solve the i^{th} subgoal, G_i (a sequent), of the state by applying *unification* and converting it into a *basic* sequent.

- If it can unify a formula on the left with a formula on the right then it deletes the i^{th} subgoal and applies the *unifier* thus found, to the rest of the proof state.
- There may be several different pairs of unifiable formulae, thus giving *several possible unifiers* and hence several possible next proof states, which in turn, is returned as a *sequence* of possible next proof states. E.g., applying *unify* to the subgoal

  $$\text{P(?a),P(?b)} \vdash \text{P(f(c)), P(c),}$$

  generates a sequence of four possible next proof states. However, only the first of these is computed, with the others available upon demand, since sequences are lazy ([Paulson, 1996, pg.423]).

Use of the notion of environments, for implementing unification

A code-fragment describing the implementation of the *unification* algorithm is given in Listing 10 (in turn, used in the implementation of the *unify* tactic). This returns the list of possible unifiers, for the given goal, as a list of *environments*. The notion of an *environment* is defined as follows: it is a place-holder for the mappings of variables and terms, $[(?a_1,t_1), \ldots, (?a_k,t_k)]$, where $?a_i \neq t_i$, for all i. A *dictionary* data structure holding values of type *string* is used for implementing the notion of an *environment*.

An *environment* acts as an accumulator and is used to build the unifier as the algorithm executes. It is not necessarily the final unifier, it may be subject to addition of a variable-term mapping which can subsequently prove to be incompatible with the rest of the environment and will have to be removed. It is used to apply the substitution to a term and in turn, to a goal.

Sequence of unifying environments, lazy evaluation The function *unifiable* in Listing 7.3, generates a *sequence* (lazy list) of unifying environments for a single goal (via the *Seq* structure in Seq structure).

The tactic, *unify*, returns a sequence of next states: Each unifier can result in a possible next-proof state. The corresponding next-state for an unifier, is obtained by applying the unifier (*environment*) to the entire proof, i.e. all the subgoals (in Listing 7.3, the function *inst* applies a given unifier to a given goal and *next* returns the corresponding proof state.).

Listing 7.2: Unification algorithm for first-order logic. The algorithm works by comparing the structures of the inputs, element by element. The substitution *mu* is built up along the way and is used to make sure that later comparisons are consistent with bindings that were established earlier.

```
Unify(x, y) = Unify_internal(x, y, [])

Unify_internal (x, y, mu)
 If (mu = Failure) then return Failure
 If (x=y) then return mu
 If (Is_a_variable(x)) then return Unify_variable(x, y, mu)
 If (Is_a_variable(y)) then return Unify_variable(y, x, mu)
 If (Is_a_compound(x)) and (is_a_compound(y)) then
 return Unify_internal( args(x), args(y), Unify_internal(op(x),op(y),mu )
 If (Is_a_list(x)) and (Is_a_list(y)) then
 return Unify_internal( tail(x), tail(y), Unify_internal(head(x),head(y),mu)
 return Failure

Unify_variable(var, x, mu)
 If (a substitution value/var is in mu) then
 return Unify_internal(value, x, mu)
 If (a substitution value/x is in mu) then
 return Unify_internal(var, value, mu)
 If (var occurs anywhere in x) then return Failure
 Add x/var to mu and return
```

Thus, when the *unify* tactic is applied to the i^{th} goal, say g_i, of the given proof state, say s, it returns a *sequence* of possible next-states. The sequence can potentially be empty if there is no unifier, in which case, by definition, the tactic is considered to have failed.

[9] See §A 9 for code for the Unify structure

Listing 7.3: Code fragment for the unification tactic in HAL [9]

```
(*The functions use the Unify structure which implements the standard unification
    operations and
uses the notion of environment, a variable-term mapping, to hold a unifier. Environment
    is in turn,
implemented as a dictionary data structure*)

(*Generates a sequence of unifying environments for a single goal (sequent: a pair of
    formula lists)*)
fun unifiable ([], _) = Seq.empty
|unifiable (p::ps, qs)=let fun find [] = unifiable (ps,qs)|find (q::qs)=Seq.cons(Unify.
    atoms(p,q), find qs)
handle Unify.Failed => find qs
in  find qs  end;

(*This function applies a given unifier to a given goal*)
fun inst env (gs,p,n) = State (map (Unify.instGoal env) gs,  Unify.instForm env p, n);

(*for solvable goals with unifiable formulae on opposite sides*)
fun unify i (State(gs,p,n)) = let
   val (ps,qs) = List.nth(gs,i-1); fun next env = inst env (spliceGoals gs [] i, p, n)
in    Seq.map next (unifiable(ps,qs)) end
handle Subscript => Seq.empty;
```

7.5.4 Sequential automatic proof search procedures in HAL

Using the basic tactics and control structures described above, the following automatic proof search procedures are defined :

firstF:

The *firstF* tactic uses the choice operator —— (commits to the first successful tactic, with no backtracking), to provide a convenient means of combining primitive inference rules in different ways. It is used in the implementation of the *safe* and *safesteps* tactic, described below.

```
fun orelseF (tac1, tac2) u = tac1 u || tac2 u
fun firstF ts = foldr orelseF (fn _ => no) ts
```

safe, safeSteps:

The *safe* tactic applies *one* safe tactic to the goal (as defined in safe tactics). It does not perform unification or variable instantiation and cannot render a provable goal into unprovable subgoals.

```
val safe =  firstF
 [basic, conjL, disjR, impR, negL, negR, exL, allR, (*1 subgoal*)
 conjR, disjL, impL, iffL, iffR (*2 subgoals*)]
```

The *safeSteps* tactic applies a nonempty *series* of safe tactics to a given subgoal (as defined in safe tactics). Tactics that create one subgoal precede those that create two subgoals. Apart from this, their order is arbitrary.

```
fun safeSteps i = safe i -- repeatDeterm (safe i)
```

quant:

The *quant* tactic applies both exR and allL, if possible. Expands at least one quantifier in the given subgoal, maybe two: if allL succeeds, then it still attempts exR too. Note that the composition operator -- is used in this implementation.

```
fun quant i = (allL i -- try (exR i)) || exR i;
```

step:

The *step* tactic refines the given subgoal, by using the safe tactics if possible, otherwise it tries unification and quantifier expansion. —@— is used to combine *unify* with the quantifier tactics. Thus, even if unification is successful, the search may investigate quantifier expansions too.

```
fun step i = safeSteps i || (unify i |@| allL i |@| exR i)
```

Depth-first (automatic) proof search procedure, *depth*:

Given a subgoal, *depth* attempts to solve it by breaking down some formula or by unification or by expanding quantifiers, using a depth-first-search based approach (using the function *depthFirst*).

The function *depthFirst* explores the search tree generated by the given tactic, by repeatedly applying the tactic until the given predicate, *pred*, is satisfied. Note that the composition operator -- is used in this implementation.

```
fun depthFirst pred tac x = if pred x then all else (tac -- depthFirst pred tac)
val depth = depthFirst final (safeSteps 1 || unify 1 || quant 1);
```

Quantifiers can be expanded repeatedly without limit; thus, the tactic may run forever. However, the individual components of *depth*: (i.e. safeSteps, quant, unify), are useful for interactive proof, especially when *depth* fails.

Iterative deepening: *depthIt* Applies the iterative deepening technique for search, using the *step* tactic defined above

```
fun depthIt d = depthIter (final, d) (step 1)
```

7.6 New concurrent tacticals

In this section, we describe two new concurrent control structures to apply tactics (i.e. concurrent tacticals), which we have developed and implemented in our prototype. Each control structure encapsulates the application of a concurrent programming technique, as relevant to the application of tactics. These two new tacticals demonstrate our multilayered approach to introduce concurrent techniques, of using abstractions to implement the concurrent tacticals:

distComp, an asynchronous composition operator to apply tactics: This illustrates the scope of harnessing asynchronous execution to (re-)implement established tactic-operations (in this case, `--`, the composition operator for tactics), such that it can handle scenarios that are beyond the scope of the corresponding sequential implementations. We have used an asynchronous, data-driven execution model to engineer *distComp*, addressing the limitations of the sequential composition operator of HAL, imposed by the demand-driven aspects of lazy evaluation.

Fastest-first, a novel control structure, to return the fastest-returning tactic: This illustrates the scope to utilise new computational patterns, made feasible by asynchronous execution (in this case, fastest-first), to synthesise novel concurrent tacticals.

In §7.8, we describe *crossTalk*, a new implementation of *unify*, the unification tactic of HAL. It uses a collaborative approach to the computation of the unifiers shared by various sub-goals.

Application of these new tacticals to implement new automatic proof search procedures in HAL is described in §7.9.2. Definitions and explanations of related parallel programming and Alice ML terminology can be found in §4.8 and §5.6.

7.6.1 Distributed composition

In §7.6.1.1, we describe the limitations of the sequential composition operator, -- . Examples highlighting the same are provided in §7.6.1.1. §7.6.1 describes the design of the new asynchronous composition operator, *distComp* and how it addresses the limitations of --. §7.6.1.3 summarises the efficacy of *distComp*, in comparison to -- and provides an example proof illustrating the same. The data-driven evaluation

model used in the implementation of *distComp*, has been realised via programming abstractions and these are described in §7.6.1.2.

7.6.1.1 Limitations of the sequential composition operator

As explained before, for a given proof state, x and tactics, tac_1, tac_2, tac_1 `--` tac_2 gives the sequence of sequences:

$$tac_2(y_1),\ tac_2(y_2),\ \ldots,\ \text{where,}$$
$$tac_1 x\ =\ y_1, y_2,\ \ldots,\ \text{and}$$
$$\text{each } y_i \text{ is a sequence, } i\ =\ 1,\ 2,\ \ldots,\ \text{say, } y_i\ =\ y_{i1},\ y_{i2},\ \ldots\ .$$

The `--` operator returns the concatenation of all the individual sequences. Each individual sequence, $tac_2(\ y_i)$ incorporates lazy evaluation. Thus, though the sequences are concatenated, because of the *demand-driven* behaviour of lazy evaluation, the actual evaluation is triggered only when the result is demanded by another operation. The execution is as follows :

- tac_1 tries to get its first result, i.e. computation of y_1 is triggered;
- $tac_2(y_1)$ is computed and after application of tac2 on all members of y_1 is completed,
- computation of y_2 is triggered (because of lazy evaluation) and then,
- $tac_2(y_2)$ is computed and after the computation is completed,
- computation of y_3 is triggered and so on.

However, in an asynchronous execution model, we can implement the composition operator, using a *data-driven*, *producer-consumer* computation model, i.e. the results of tac_1 are *consumed* by tac_2, as they are *produced* (by tac_1) . Here are some scenarios where a data-driven computation model can benefit (compared to the demand-driven execution model described above) :

1. Computation times of a producer (input data) is irregular
2. Computation times of a consumer function is irregular
3. The consumer function can process each input independently, thus making it unnecessary to wait for the entire input to be available, before it can process the first candidate

Examples

In this section, we present two examples (using the notation described above), where the sequential mode of execution can inhibit the proof search process:

Example 7.1 Irregular data size

With the sequential execution described above, tac_1 `--` tac_2 (x) ,

though the application of tac_2 on each y_i, is an independent computation, because of the sequential nature of the execution and the demand-driven aspect of lazy evaluation, computation of $tac_2(y_i)$ gets triggered only after the computation of $tac_2(y_{i-1})$ is completed.

tac_1 `--` tac_2 (x) produces results in the following order:

$$y_{11}, y_{12} \ldots, y_{21}, y_{22}, \ldots, \cdots$$

This can prove to be limiting in many scenarios. For instance, if y_1 is an infinite sequence, then, computation of $tac_2(y_1)$ will never complete and so, application of tac_2 to y_2 will never start. Thus, even if application of tac_2 to the very first element of y_2 solves the (sub-)goal, i.e. y_{21} is a solution, it will never be reached.

Example 7.2 Irregular computation times

Using `--`, the sequential composition operator, consider the scenario where $tac_2(y_k)$ is a solution; but, application of tac_2 on the predecessors is taking a significantly long time to complete i.e. suppose,

$$tac_1(x) = y_1, y_2, \ldots, , y_{k-1}, y_k, \cdots$$
where, *k-1*: large

and some or all of $tac_2(y_i), i = 1 \ldots (k-1)$ are taking a significantly long time to be computed. Thus, using `--`, there will be a significant delay in reaching the solution, $tac_2(y_k)$. This is the case despite the computation of the solution being independent of the computation of the predecessors.

7.6.1.2 Implementation of distComp and related programming abstractions

In the previous section, we highlighted the limitations of --. We have used asynchronous execution to enable a *data-driven* evaluation model. Using this evaluation model, we have implemented an asynchronous composition operator for tactics, called *distComp*. It composes two functions, but the results are computed in a different order, compared to the ones produced by the sequential composition operator, --.

Implementation of control structures to apply tactics in HAL, is dependent on the associated operations of the *Seq* structure. We have added new functions to the *Seq* structure, to allow the following asynchronous operations, while still adhering to the functional aspect. The code fragment in Listing 7.4 show these implementations. Inline comments explain the purpose of each function. The new functions are:

***distMap*, distributed function application:** The function *distMap* applies a given function on a sequence, in an asynchronous manner. It has been realised using the *spawn* library function of Alice ML. (`Recap`: *spawn e* returns a *future*, a placeholder for the result of the concurrently evaluated expression, *e*.)

Data-driven evaluation:

byTime The function *byTime* performs the merger of two sequences; it returns the members, as they are computed, rather than in an order defined by the concatenation operation.

sequencesByTime Takes a sequence of sequences, and merges in the order in which items in the sequences are computed.

The data-driven evaluation model implemented by these functions share similarities with the *pipeline* and *producer-consumer* programming abstractions found in the parallel programming literature (described in §5.4.2.2 and §5.4.2.1 respectively).

Using the asynchronous operations for *Seq* mentioned above, *distComp* is implemented as follows:

```
(*sequential composition operator*)
fun (tac1 -- tac2) x = Seq.concat (Seq.map tac2 (tac1 x))
(*distComp operator*)
fun lazy (tac1 distComp tac2) x = Seq.sequencesByTime ((Seq.distMap tac2) (tac1 x))
```

Listing 7.4: Code for adding asynchronous operations to the Sequence structure in HAL

```
structure Seq = struct
 datatype 'a t = Nil | Cons of 'a * 'a t; exception Empty;
 val empty = Nil; fun cons(x,xf) = Cons(x, xf); (*constructors*)

 (*New functions to allow asynchronous operations on "Seq"*)
 fun lazy distMap _ Nil = Nil |   distMap f (Cons (x,g) ) = Cons (spawn f x, distMap f g)

(*This fn returns the fastest results from two given sequences*)
 fun lazy byTime (s1, s2) = let
  fun lazy hdTl l = let val a=hd l and b=tl l in SOME(a,b) end handle Empty => NONE
  val h2 = spawn hdTl s2; val h1 = spawn hdTl s1
 in
  case (Future.awaitEither (h2,h1)) of
    SND (SOME (a,b)) => lazy cons(a, byTime (b, s2)) | SND NONE => s2
  | FST  (SOME (a,b)) => lazy cons(a, byTime (s1, b)) | FST NONE => s1
 end

(*Merges a sequence of sequences, in the order in which items are computed*)
fun sequencesByTime (Cons(s1,ss)) = byTime (s1, (sequencesByTime ss))  | sequencesByTime
     Nil = Nil
end
```

7.6.1.3 Utility of *distComp*

As explained in §7.6.1, the *distComp* operator implements a *data-driven* evaluation model, to implement the producer-consumer scenario. Thus, using the *distComp* operator (as $distComp(tac_1,tac_2)\ (x)$), the results of tac_1 can be consumed by tac_2 as they are produced, unlike the sequential composition operator --.

tac_2 generates its results in an asynchronous manner too. Thus, in the case of Example 7.1, when y_1 is an infinite sequence, because of the asynchronous nature of the *distComp* operator, the computation $tac_2(y_2)$ can be spawned independently without waiting for $tac_2(y_1)$ to complete.

Also, the results of tac_1 get picked up asynchronously. In the case of Example 7.2, as $(tac_1 x)$ is being produced, it is being consumed by tac_2 . So, even though, the computation of $tac_2(y_i), i = 1 \ldots \mathbf{k}$ is taking too long, the computation $tac_2(y_k)$ would have been spawned already.

A concrete proof attempt using this operator, in HAL, is given in §7.10.1.

7.6.2 Fastest-first: a novel choice operator using asynchronous concurrent execution

As mentioned earlier in the thesis in §5.2.1, the use of asynchronous execution modes opens up approaches that are not possible in a sequential mode of execution. One such possibility is of enabling a novel choice operator to address scenarios where evaluation of one candidate suffices. The novel form of choice is as follows: spawn the evaluation of the choices simultaneously and choose the one whose computation terminates earliest (a.k.a *Fastest-first*).

We have introduced the following new choice operators in HAL: **FF** and **FFOnList**. The code fragment in Listing 7.5 outlines how these have been realised, using the asynchronous operations of the *Seq* structure that we have developed (described earlier in §7.6.1.2). **FFOnList** works on a list of tactics and commits to the first tactic that comes back with a result.

Listing 7.5: Implementation of fastest-first tactic in HAL

```
(*Uses the byTime function defined in Seq structure, which returns the
   fastest results from two given sequences*)
fun FF (tac1, tac2) x = Seq.byTime (tac1 x, tac2 x)
(*Uses the sequenceByTime function defined in the Seq structure, which returns
  the elements of a sequence of sequences in the order of their computation  *)
fun FFOnList tList s = Seq.sequencesByTime (Seq.fromList (List.map (fn t => t s) tList))
```

7.7 Integrating a SAT solver into HAL: counterexample finder and simultaneous proof and refutation attempts for propositional goals

Incorporation of a SAT solver into an LCF prover, to solve propositional (sub-)goals has been addressed in the literature, in a sequential setup, e.g. [Weber, 2006] reports work on integrating *zchaff*, a DPLL based propositional solver [Yogesh Mahajan, 2004], into Isabelle [Nipkow et al., 2002]. We have integrated an external SAT solver into HAL and engineered two new tactics as described below. We have used the SAT solver as an oracle, i.e., its result is not independently verified by HAL. We have used

the DPLL-based SAT solver (augmented with conflict-driven learning), implemented in the SAT case study described in chapter 6, as the external SAT solver.

Counter example finder

We have engineered a new tactic called *findCounterEx_propGoal_SAT*. This tactic tries to find a counter-example for a given propositional (sub-)goal, by invoking an external SAT solver. Listing 7.6 shows the code fragment for the same. For a given goal, , this tactic performs the following steps:

Is goal propositional? If *ithG* is not propositional, the tactic returns a status, denoting failed application of the tactic, else, proceed, to try to find a counterexample.

Try to find counterexample: Invoke the external SAT solver, *dpllCDCLSAT*, with the negation of the propositional formula for the goal, *ithG* and return a list of possible next goals, based on the outcome of the SAT-solver-call.

Counter-ex found: If the SAT solver call comes back with a true status, i.e., a counter example has been found for the given goal, then, print an appropriate message, with the goal and the counter example; raise an exception with the goal and the counter example; this information can be potentially used in other ways; in the current implementation, the exception is handled merely by returning an empty sequence.

No counter-ex found: As an exhaustive search to find a counterexample for the propositional goal (*ithG*) has been unsuccessful, *ithG* is true.

Listing 7.6: Code fragment for the SAT-based counterexample finding tactic in HAL; *dpllCDCLSAT* is the external SAT solver.

```
exception notPropGoal; exception counterexample of (Fol.form * (Fol.form list))
fun findCounterEx_propGoal_SAT i s = let
 fun findCounterEx_getGoalList ithG=if not(isGoalProp ithG) then (raise notPropGoal
     ) else
 let val propF = getPropFormForGoal ithG
      val neg_propF = Fol.Conn("~", [propF])
      val (boolRes, assgn) = dpllCDCLSAT neg_propF (*find counter example*) in
 if boolRes then
 (print "Counter example found, goal, counter-Ex:";DisplayFol.goal i ithG;
  List.app (DisplayFol.form) assgn; (raise counterexample(propF,assgn)))
 else (print "No counter examples found, for the goal:"; DisplayFol.goal i ithG;[])
 end
 handle notPropGoal => [] | counterexample(_,_) => []
in  propRule "findCounterExample" findCounterEx_getGoalList i s end
```

Simultaneous proof-refute attempts

Even when a (sub-)goal is propositional, it is hard to judge the likelihood of the (sub-)goal's truth-status, i.e. if a proof attempt should be attempted or an attempt should be made to search for a counter-example. To address this challenge, we have used asynchronous execution to synthesise a new tactic, called *proveAndDisprove_SAT*. Listing 7.7 gives a code-fragment describing the high-level design of this new tactic. For a propositional goal, this tactic:

- Spawns the following two simultaneously:

  Try to prove: via an LCF-style proof attempt, using the automatic tactic, *depth*, for this sub-goal (Note that the *depth* tactic is not invoked on the entire proof-state).

  Try to refute: using an external SAT solver, using the tactic, *findCounterEx_propGoal_SAT*, described above.

- If counter example returns faster, it terminates the proof attempt by *depth*. However, for propositional goals, the *depth* tactic can terminate, when no more inference rules can be applied. So, the completion of the tactic does not guarantee that the goal has been solved. So, the counter example finder is not terminated[10].

- For non-propositional goals, it returns an empty sequence, i.e., failure status (in line with HAL's convention to denote that a tactic cannot be applied).

Listing 7.7: Code fragment for implementation of SAT-based proof and refutation in HAL;

```
fun proveAndDisprove_SAT i  st = let
 val ithG =   List.nth( (Rule.subgoals(st)), (i-1))
in
 if not(isGoalProp ithG) then (raise notPropGoal) else
 let
  val propF = getPropFormForGoal ithG
  val proofSt = Rule.initial propF (*make a local proof state for this call of
      depth*)
  val (t1,f1) = Thread.spawnThread (fn () => (depth proofSt)) (*Call HAL's depth*)
  val (t2,f2) = Thread.spawnThread (fn () => (findCounterEx_propGoal_SAT i st) ) (*
      counter example*)
 in
  case Future.awaitEither(f1,f2) of
  FST(r) =>   (msgBrd.printToStrm "\n Depth tactic completed!\n"; r)
```

[10]The fastest-first tactic, FF, can be used here. However, in its current form, FF will terminate the counterexample finder, which we don't want.

```
|SND(r) => (msgBrd.printToStrm  "\n Counter ex finder completed\n"; Thread.
    terminate t1;r)
 end
end
```

Example

Example 7.3

Consider the negation of the associativity of the & operator, as follows:

$$goal\ "\neg\ ((P\ \&\ Q)\ \&\ R\ \rightarrow\ P\ \&(Q\&R))"$$

Application of the depth tactic i.e., by (Tac.depth), fails.
Application of *findCounterEx_propGoal_SAT* guides the user by showing a counter example

7.8 Collaborative unification: using communication for unification

We have investigation the scope of using collaborative exchange of partially evaluated information by multiple asynchronous processes, to address a theorem proving scenario. To this end, we have considered the concrete scenario of unification (the *unify* tactic of HAL).

In this section, we describe the limitations of the sequential *unify* tactic in HAL, discuss our proposed solution and its implementation as a new tactic called *crossTalk*.

crossTalk applies *unification* across a given list of sub-goals and orchestrates exchange of partially-evaluated information. A detailed example which uses *crossTalk* within an automatic proof search procedure is provided in Example §A 8.

7.8.1 Limitations of the sequential *unify* tactic in HAL

In the sequential implementation of the *unify* tactic in HAL (described in §7.5.3):

- HAL provides the *unify* tactic to apply unification on a given goal. It attempts to solve a subgoal by applying *unification* and converting it into a *basic* sequent. If it can unify a formula on the left with a formula on the right then it deletes the subgoal and applies the *unifier* thus found, to the rest of the proof state. This new proof state is a possible next-proof-state.

- For a given (sub)goal, say, G_i (a sequent), there may be several different pairs of unifiable formulae, thus giving several possible unifiers and corresponding possible next-proof-states. A next-proof-state corresponding to an unifier is computed by applying the unifier across all the sub-goals of the given state. Given that the next-proof-states are returned as a *sequence*, their computation is performed only on *demand*.

  E.g., applying *unify* to the subgoal

  $$\text{P(?a),P(?b)} \vdash \text{P(f(c)), P(c),}$$

  generates a sequence of four possible next proof states. However, only the first of these is computed, while the others are available upon demand, because of the lazy evaluation of sequences ([Paulson, 1996, pg.423]).

  Note that the unifier list is not implemented as a *lazy* list though. Let the possible unifiers produced by application of *unify* on G_i be: $[U_{i1}, U_{i2} \ldots U_{ip_i}]$.

- Meta-variables can be shared across multiple sub-goals. When this happens, for a U_i to lead to a successful proof, it needs to serve as a unifier for the other sub-goals sharing its meta-variables. We will refer to such a unifier as a *consistent/consensus* unifier, for the rest of this thesis.

- When used within an automatic proof search procedure, a suitable backtracking mechanism will need to be employed to ensure that all the unifiers are considered. In the *depth* tactical in HAL (see §7.5.4 for details), *unify* is applied repeatedly along with other rules, within a depth-first-approach-based search strategy.

  Given the sequential nature of the *depth* tactical, only one candidate from the *unify* tactic gets considered at any given time. And because of the *demand-driven* behaviour of the lazy list of states returned by *unify*, the next state gets produced only when the *depth* tactic finishes computation of the earlier state and requests the next state. When this happens, the *ordering* of the unifiers produced ($[U_{i1}, U_{i2} \ldots U_{ip_i}]$) influences the behaviour of *depth*, as illustrated in

the examples below.

- As explained above, the sequence of next-proof-states are produced by applying the corresponding unifiers to the given proof state. However, application of an unifier in the list (which appears before the consistent unifier) can lead to bottlenecks in the proof attempt during the application of *depth*. Two such scenarios are described below.
  - Make one or more sub-goals unprovable, either because the unifier is not consistent with the sub-goal or due to other reasons.
  - Lead to a looping situation in the proof state. This happens in the example described in Example §A 8. If this happens, the other unifier candidates never get considered, thus sabotaging a possibly successful proof attempt. The reasons for the occurrence of looping can be varied. Some examples are:
    - Duplications of sub-formulas can get added to the right hand side of the goal sequent; this is the case in Example §A 8
    - The quantification rules, *allL* and *exR* introduce meta-variables. In backward proof, they do not remove any formulae from the goal. They expand a quantified formula, substituting a term into its body; and retain the formula to allow repeated expansion. It is impossible to determine in advance how many expansions of a quantified formula are required for a proof, a consequence of the undecidability of provability in FOL. Thus, when these are applied after an unsuccessful application of the *unify* tactic, it can result in the repetitive applications of the two steps of meta-variable introduction and unification, without ever terminating.

The limitations have particular significance when there are multiple (sub)goals in a given proof state sharing a list of meta-variables, say, *mVL*. In a typical proof attempt (automatic or interactive) inference steps get applied to each sub-goal during different stages of the proof. When *unify* gets applied to a sub-goal, we want the resulting unifier (as applicable to *mVL*) to be consistent with the unifier found for every other sub-goal. Please note that we are not talking of solving the sub-goals, but only finding a unifier that is consistent with all sub-goals. More inference steps may need to be applied to

the individual sub-goals to progress the proof, even after the consistent unifier has been found. This in turn, may result in the instantiation of more meta-variables, possibly calling for further applications of unification.

7.8.2 Gist of our solution: asynchronous evaluation and collaborative use of partially evaluated information

The gist of our solution (implemented as the *crossTalk* tactic) is to use *partially evaluated* information in a *collaborative* manner to compute a list of unifiers each of which is consistent across a given list of sub-goals:

1. The process of finding the unifiers is local to the sub-goal. So, it can be spawned for the *n* sub-goals, in an asynchronous manner. Spawn independent computations in an *asynchronous* manner to apply *unify* on each sub-goal and post the list of unifiers to a common location, say *board*. The unifiers are not applied to the proof-state as yet.

2. The unifiers produced by each sub-goal are *partially evaluated* information as they by themselves cannot guarantee the despatch of all the proof obligations. We introduce a new process called *referee agent*. This uses the information available in *board* to compute the *consensus* candidate(s), i.e. an unifier that will work for all the sub-goals which share the meta-variables involved. This approach has been implemented as the *crossTalk* tactic in HAL and is described in the next section. It performs unification across a given list of sub-goals.

3. A referee agent collects the unifiers produced and computes the list of unifiers that are consistent with all the sub-goals. If an agreement can't be reached, then, an empty list is returned.

4. The referee agent can choose to wait for each sub-goal to compute its entire list of possible unifiers or it can choose to act as and when they are produced. We have implemented only the former option, as with the latter option, the type signature of tactic will not be adhered to.

Thus, *crossTalk* has the following advantages:

- The lazy aspects and the related problem of the order in which the unifiers are produced are addressed

- It allows us to make use of multiple threads/processes in an effective way.
- It offers a *novel* way of employing asynchronous computation techniques and collaboration to compute unifiers, consistent across a list of sub-goals. Such an implementation is not possible in a *sequential* mode of execution.
- In general, the search space is smaller when there are fewer meta-variables. A goal which has a higher proportion of instantiated meta-variables can thus be considered to be generally easier to solve. *crossTalk* returns only the consistent unifiers. Thus, for subsequent steps after the application of the *crossTalk* tactic, the sub-goals are easier to solve.
- By eliminating the unsuccessful candidates for the consensus, *crossTalk* prunes the subsequent search space for the sub-goals.
- *crossTalk* is available at the top level as a tactic for interactive use and can be used interactively and to code automatic proof searh procedures.

In Example §A 8, we describe an example where HAL's *depth* tactic (which uses *unify*) loops whereas *depthCrossTalk* (which is identical to *depth* with *unify* replaced by *crossTalk*) succeeds in finding the proof.

7.8.3 CrossTalk: a new proof tactic implementing collaborative unification

In this section, we describe our implementation of the solution described above, as a tactic: *crossTalk*. This tactic computes the consensus unifiers across a given list of sub-goals and a given proof state. If given an empty list as the first argument, it computes the consensus across all the pending sub-goals of the given proof state. The type signature of *crossTalk* is given below [11]. The Alice ML code fragment that describes the implementation in detail is given in §7.8.3.2.

```
(*Takes an integer list as parameter; This gives the indexes of the sub-goals to consider
to compute the consensus unifier. If it is empty, all pending sub-goals are considered*)
  val crossTalk : (int list) -> tactic
```

[11] crossTalk takes an integer list and returns a tactic. But, in this work, we refer to crossTalk as a tactic, as the behaviour is essentially the same as that of taking a state and returning a sequence of states

In §7.8.3.1, we describe *refereeAgent*, the programming abstraction that we have implemented. This abstraction addresses the generalised scenario for computing the list of collaborative consensus candidates. In Listing 7.8, we describe the Alice ML code fragment that implements the *referee* abstraction.

7.8.3.1 Referee abstraction

The approach described above can be viewed as a particular case of the following more general case: There are p *worker* processes: $G_1, G_2, \ldots G_p$. All the p agents need to agree on something, say *consensus*. Each process produces some possible candidates for the *consensus* as follows:

$$\begin{aligned} G_1 &: [U_{11}, U_{12} \ldots U_{1k_1}] \\ G_2 &: [U_{21}, U_{22} \ldots U_{2k_2}] \\ &\vdots \\ G_p &: [U_{p1}, U_{p2} \ldots U_{pk_p}] \end{aligned}$$

The task is to come up with a list of all *consensus* candidates. Clearly, we need to consider all possible combinations, i.e. we need to consider $(k_1 * k_2 * k_3 * \ldots * k_p)$ combinations. This naturally fits into a model of information-sharing and asynchronous execution. We have implemented the functionality of computing the consensus candidates as a programming abstraction, *refereeAgent*. This scenario also holds potential for applying constraint satisfaction techniques, which in turn, may offer scope for applying distributed techniques. This is a possible option for future work.

The implementation of the *refereeAgent* programming abstraction is as follows:

- Each *worker* process posts its results to a location, say *board*.
- A process, *refereeAgent*, monitors all the results posted by the agents.
- Each *worker* needs to know only the location of the *board*. It does not need to know any information about the *refereeAgent* or other worker agents.
- The *refereeAgent* does not need to know any information about the identity of the *worker* agents. Only the results of the agents are needed and their identity is not needed.
- This allows for scope of distributing these processes over cluster-like network

Listing 7.8: Code fragment for the referee abstraction in HAL

```
fun refereeAgent intersectionFn filterFn workList = let
 val ch = Channel.channel();
 val resList =  List.map  (fn (f,x) => spawn f ch x ) workList
 do List.app (fn r => await r) resList
 val consensus = getConsensus  filterFn ch
in  consensus end
```

architectures that are designed for large scale distribution of work without high levels of communication traffic. The multithreaded implementation shares the same computational model. We have implemented a multithreaded implementation in our current prototype.

The *refereeAgent* abstraction, given in the code fragment in Listing 7.8, is parametrised by the following:

workList: List of *workers*, provided as a list of functions. Each function performs a computation and posts the results to the given location, say, *board*.

filterFn, getConsensus: The functionality of computing the consensus is abstracted using the function *getConsensus*; it takes a location as argument, say, *board* and performs the operation of computing the possible consensus candidates from the data on *board*. *filterFn* allows for filters to be applied to the data considered for processing the consensus.

In our current implementation, the referee waits for every process to finish and then generates the list of consensus candidates. The results of all the agents are indeed needed, as we are trying to compute the consensus candidates. So, there is no efficiency loss by waiting for a potential candidate from each process. In terms of computational models, message-passing may not be useful in most cases as it will lead to a lot of traffic, when 'n' is large. Instead, information sharing achieved by posting to a common location that is monitored by the referee agent autonomously is better suited to address the scenario.

Referring to the code given in Listing 7.8, *refereeAgent* (i) waits for all agents to finish (ii) pools all the results (iii) applies *getConsensus* and returns the list.

7.8.3.2 CrossTalk: code

The Alice ML code for the *crossTalk* tactic is given in Listing 7.9. The high level design of the same is as follows:

- Take an integer list (*gNumList*) and proof state (*st*) as arguments. *gNumList* gives the indices of the sub-goals that the consensus unifiers has to be found for. It returns the possible next-states as a sequence of states.
- Collect all the meta-variables (as *mVList*) in the list of sub-goals. If *mVList* is empty, then, *crossTalk* returns an empty sequence.
- Compute the list of consensus unifiers and hence states by instantiating the *refereeAgent* abstraction defined earlier, using a work function list and an appropriate intersection function
- The function *pseudoUnifyNonTactic*, has type signature as given below. This corresponds to the *unifiable* function of the sequential implementation of unification in HAL (see Listing 7.3). One important difference is that it does not apply the unifier(s) to the goal (Hence the "pseudo" in the function name!). It posts the unifier(s) using the *myBroadcastFn* to the given location: *ch*.

```
val pseudoUnifyNonTactic:((Fol.term StringDict.t) list->unit)-> int ->tactic
```

Listing 7.9: Code fragment for crossTalk:collaborative unification in HAL

```
fun getListOfConsUnifier mVList gNumList st = let

 val sGList = subgoals st (*sub-goal list of given proof state*)
 fun myBroadcastFn ch x = Channel.put(ch,x)  (*Broadcast fn, parametrised by broadcast
     location*)
 fun workFn ch g = pseudoUnifyNonTactic (myBroadcastFn ch) g (*ch: broadcast location, g
     :(sub-) goal*)

 val gList = List.map (fn i=> List.nth (sGList, (i-1))) gNumList
 val workList =  List.map  (fn g =>(workFn, g)) gList
 val filterFn = (not o (StringDict.isEmpty))
in
 refereeAgent (getConsensusForEnvList mVList) filterFn workList
end

fun crossTalk gNumList st = let (*st: current proof state; gNumList: indices of sub-goals
     to be unified*)
 val State(sGList,g,i) = st (*sGList:subgoal list,g: main goal*)
 val mVList = List.foldr (*meta-variable list for all sub-goals*)
 (fn (sGoal, mVListAccum) => Fol.goalVars (sGoal,mVListAccum)) [] sGList;
```

```
 val numOfSGoals = List.length sGList (*num of subgoals; STEP-1*)

(*Get the list of consensus proof states using above abstraction*)
 val newSt = if (List.null mVList) then Seq.empty else let
 val gNumListLocal=if (List.null gNumList) then(*all sub-goals*)
    (List.tabulate (numOfSGoals,(fn i=>(i+1)))) else gNumList

(*Get list of consensus unifiers using abstraction STEP-2.2,2.3*)
 val unifEnvList=getListOfConsUnifier mVList gNumListLocal st(**)
 fun instSubGoals x=List.map (fn sG => Unify.instGoal x sG) sGList
 val stList=List.map(fn env =>State ((instSubGoals env),g,i))unifEnvList
 in  (Seq.fromList stList)  end(*STEP-3*)
in  newSt end
```

7.8.3.3 Possible improvements

In this section, we outline some possible improvements that can be done in the implementation of *crossTalk*, which we consider as possible future enhancements to the prototype.

A better implementation of *crossTalk* will be where

- The referee agent posts its results as and when they are found, instead of waiting for the entire list to be computed. I.e. the implementation of *refereeAgent* can be as follows: To compute the consensus, the referee need not wait for a worker process to finish computing all the possible candidates. It can start generating consensus candidates as and when one full list of contributions from all workers is available.
- Another possible improvement is for the referee to post the results as they are being generated instead of returning them as a list. This will help the reaction of subsequent steps which are dependent on the results being generated by the referee.

Including these features changes the type signature of *crossTalk* to be different from that of a tactic. This is because *crossTalk* will now be returning individual states rather than a *sequence* of states and thus the type signature of *tactic* of returning a *sequence* of states will not be adhered to. This can limit its plug-and-play usage as a tactic.

Listing 7.10: Code fragment for *depthCrossTalk*: collaborative unification based automatic proof search

```
(*Standard depth-first based automatic tactic in HAL*)
val depth = depthFirst final (safeSteps 1 || unify 1 || quant 1);

(*Use the crossTalk tactic instead of unify in the above line; Pass [] as the first
    argument
  to crossTalk to unify across all pending sub-goals*)
val depthCrossTalk =   depthFirst final (safeSteps 1 || crossTalk [] || quant 1);
```

7.9 Novel automatic search procedures employing concurrent and collaborative approaches

In §7.6.1, §7.6.2, §7.8.3 and §7.7, we described novel concurrent tacticals that we have implemented in our prototype. One of the key objectives of our multilayered approach has been to encapsulate the use of concurrent techniques as proof-tactics, so that they can be use along with other sequential/concurrent tactics using the sequential/concurrent tacticals. To demonstrate this, in this section, we describe a few novel automatic proof search procedures, which we have developed using the new proof-tactics and concurrent tacticals described earlier.

7.9.1 Using *crossTalk* in an automatic search procedure

The modified version of unification, which we call *crossTalk*, is available at the top level as a tactic for interactive use. It has also been used to generate a new automatic search procedure based on the existing *depth* automatic tactic. The *depth* tactic employs the depth-first approach using a combination of propositional inference rules, unification (the *unify* tactic) and quantification (the *quant* tactic) to generate the next states (see §7.5.4).

We have used *crossTalk* instead of *unify* giving a new automatic tactic, which we call *depthCrossTalk*. As we will see in §7.10.3, *depthCrossTalk* solves a problem which the sequential automatic tactic, *depth* does not solve, as it gets stuck in a non-terminating loop, during the search.

7.9.2 New depth-first automatic search procedures, using the distComp and FF operators

The *depth* tactic, described earlier in §7.5.4, uses the composition operator, in its implementation, via the function, *depthFirst*. To study the scope of the utility of the *distComp* operator here, we implemented*dist_depthFirst* and *distDepth*, using the *distComp* operator described earlier. It is worth observing that the *distDepth* operator does not implement the depth-first strategy any more, as the tree can grow depth and breadth wise simultaneously due to the asynchronous operators.

```
(* Sequential version*)
fun depthFirst pred tac x = if pred x then all else (tac -- depthFirst pred tac)
val depth = depthFirst final (safeSteps 1 || unify 1 || quant 1);

(*Using distComp*)
fun dist_depthFirst pred tac x = if pred x then all else distComp(tac,(depthFirst pred
    tac))
val distDepth =  dist_depthFirst final (safeSteps 1 || unify 1 ||quant 1); 11)
```

Extending this line of exploration, we have synthesised a suite of experimental automatic proof search procedures, using the primitives developed here. These listed in §7.10.4.

7.9.3 Using SAT-based tactics in an automatic proof search procedure

In this section, we describe an automatic tactic incorporating the proof-refutation tactic described in §7.7. While the counterexample finding tactic can be quite useful in the interactive mode, to guide the user, it can be embedded in automatic proof search procedures as well. We describe two such procedures here.

Including the SAT helper as a safe tactic in depth: depth_SAT

We have used the *proveAndDisprove_SAT* tactic described in §7.7 to implement a depth-first-approach-based automatic tactic. To accomplish this, we have followed the same pattern of the implementation of the sequential *depth* tactic in HAL, with the following modifications:

- We have included *proveAndDisprove_SAT* as a *safe* rule, referred to as *safeSteps_SAT*, in the code below.
- We have replaced *safeSteps* with *safeSteps_SAT*, in the implementation of the sequential *depth* tactic in HAL.
- This helps to deal with a propositional (sub-)goal, by simultaneously trying to *prove* it by using HAL's inference rules as well as to *refute* it, by invoking the external SAT solver, to find counter examples.
- Thus, the automatic tactic, *depth_SAT* makes the counterexample finder available for use, within an automatic proof search procedure.

```
val safe_SAT =
      firstF [basic,
conjL, disjR, impR, negL, negR, exL, allR, proveAndDisprove_SAT_tactic, (*1 subgoal*)
               conjR, disjL, impL, iffL, iffR(*2 subgoals*)];

fun safeSteps_SAT i = safe_SAT i — repeatDeterm (safe_SAT i);
val depth_SAT = depthFirst final (safeSteps_SAT 1 || unify 1 || quant 1);
fun step_SAT i = safeSteps_SAT i || (unify i |@| allL i |@| exR i);
fun depthIt_SAT d = depthIter (final, d) (step_SAT 1);
```

7.10 Evaluation

As described in this chapter, we have developed a multilayered approach, to enable sound, programmable extensions, incorporating concurrent approaches in an LCF style prover, enabling engineering of concurrent proof search procedures and implemented the same in the first-order LCF context.

There are arguably many possible approaches to applying concurrent techniques to an LCF style theorem prover. Published research in this area focuses on approaches which use a combination of heterogeneous provers and other systems (e.g., use of SAT solvers and/or first-order theorem provers to tackle higher-order problems). The objectives of these approaches are primarily geared towards tackling harder problems and/or solving problems faster. Our investigation has been geared towards programmability and experimentation with new concurrent techniques to address difficult theorem proving scenarios. While incorporation of external solvers is achievable using our approach (see §7.7), it has not been the ultimate goal of our investigation. In this section, we discuss how the objectives of our investigation have been met.

As described in 2, the *object-level* hypothesis for the LCF case-study is as follows:

> A multilayered approach to application of concurrent techniques to an LCF style first-order prover, using concurrent LCF-style tacticals, realised via programming abstractions enables:
>
> 1. Programmable extensions (to the prover), incorporating concurrent programming techniques, retaining the soundness guarantees.
> 2. Easy prototyping and evaluation of novel proof search techniques, applying concurrent programming techniques, that can be tailored to a given theorem proving application.
> 3. The novel proof search procedures use concurrent approaches to deal with theorem proving tasks and in the process, address some of the shortcomings of their sequential counterparts and fare better in some test cases.

In §7.6, §7.7 and §7.8, we described the concurrent, programmable extensions that we have implemented in the HAL system, using our multilayered approach. Each of these extensions illustrates a different possibility of using asynchronous execution, to synthesise new tacticals, as summarised below. In the rest of this section, we provide examples of concrete theorem proving scenarios, which demonstrate the utility of these extensions, by solving problems that cannot be solved by their sequential counterparts

and/or where the sequential counterparts do a lot of unnecessary search to find the proof.

At this point, it is worth pointing out to the reader, that these examples provide generic scenarios, to give a sense of how these new concurrent extensions can be utilised. For a given problem scenario, customised solutions can be tailored using these extensions. However, the starting point for our investigation has not been the analysis of problem scenarios. Analysis of a wider class of problems to find suitable concurrent approaches for them merits further investigation and is a topic for future research. During the exploratory investigatory phase, one can follow an iterative process of:

- finding examples which can potentially benefit from concurrent/parallel approaches;
- implementing the concurrent approaches, using the extensions developed;
- empirically studying the performance of the concurrent approaches and
- refining the concurrent approaches and/or their implementation.

Implementing existing functionality in a different way, using asynchronous execution: In the case of *distComp*, the starting point for our development was an analysis of the limitations of the corresponding sequential operator and how an asynchronous execution mode can be used to address the same. We used a *data-driven* asynchronous execution model to address the limitations.

Introducing new approaches, using asynchronous execution: *Fastest-first* is a novel choice operator for tacticals, returning the fastest-computing tactic (and terminating the others), from a list of tactics, all of which are simultaneously working on the (sub-)goal. This is an example of introduction of a new approach, not necessarily based on an existing tactic/operator.

Simultaneous proof-refutation attempts on a propositional (sub-)goal: Use the power of asynchronous execution to tackle a propositional (sub-)goal, by spawning proof and refutation attempts simultaneously, returning the fastest. An external SAT solver is used to perform the refutation attempt on the propositional (sub-)goal.

Introducing new approaches, using collaborative exchange of (partially-evaluated) information, between asynchronous processes:

When sub-goals share meta-variables,

- the sequential *unify* tactic in HAL performs unification for each sub-goal, producing a list of unifiers, corresponding to possible choices for making the left and right sides of the goal (sequent) the same (i.e. a basic sequent); the *sequence* of proof-states returned by *unify* is generated by applying a unifier from the list to the given proof state; compatibility of its unifiers with other sub-goals (which share meta-variables with it) is addressed in subsequent steps;
- *crossTalk* spawns independent unification attempts on each sub-goal, but does not apply them to the proof-state; collects the unifiers produced by each sub-goal (these are partially-evaluated information); produces a list of unifiers that are compatible with all the sub-goals, if any (we refer to these as *consistent* unifiers); returns a sequence of proof-states corresponding to this list of *consistent* unifiers.

As mentioned in 2 above, ease of prototyping new concurrent proof search techniques has been one of our claims, in this work. While it is hard to quantify *ease of prototyping* as a performance-metric, in §7.10.4, we illustrate this aspect, with a list of novel concurrent proof search procedures that we have engineered. These demonstrate the range of experimentation, ease of prototyping and incremental development of new proof search procedures using the asynchronous tacticals and operators implemented in this prototype. In §7.10.5, we provide a brief discussion of how the developmental hypothesis of this work has been demonstrated via this case study.

7.10.1 Utility of the distributed composition operator

As explained in §7.6.1, the *distComp* operator implements an asynchronous *data-driven* evaluation model, enabling on-the-fly application of composition, as and when the candidates are available, without waiting for the previous computations to complete. In Example 7.4, we give a concrete case of a proof attempt performed in HAL, where the distComp operator performs better than HAL's sequential composition operator.

Example 7.4 Composition operation involving a time consuming tactic

Consider the following goal:

$$\textbf{G: } (g(a) \wedge f(a)) \rightarrow (f(a) \wedge g(a))$$

Application of the sequent rule *impR* on this gives the following sequent as the new goal.

$$g(a) \wedge f(a) \vdash f(a) \wedge g(a)$$

Applications of the following inference rules: conjL, conjR gives the following two subgoals:

$$\textbf{g1: } g(a), f(a) \vdash f(a)$$
$$\textbf{g2: } g(a), f(a) \vdash g(a)$$

We want to illustrate the utility of the distComp operator compared to HAL's composition operator.

1. HAL's composition operator: `--`,
2. The new distributed composition operator, **distComp**

For this purpose, we create a scenario of tactic application, using the following new tactics (created from existing tactics):

Simulation of a time consuming tactic:

In HAL, a state includes a sub-goal list; a goal is a sequent, which in turn, is a pair of formula lists, say lhs and rhs.

For a given string x, a number n and a tactic, t, we define a new tactic with the following behaviour: If it finds x as a predicate symbol in any of the formulae in the rhs of any of the goals, then, it sleeps for n seconds and then applies the tactic t; Else it applies the tactic straight away. This is implemented by the function *newTac*, as described below:

```
fun newTac n x t s  = let
 fun IsStrInForm x (Fol.Pred(f,tList))=(f= x)|IsStrInForm x _ = false
 fun IsStrInRhsOfGoal x (g as (ls,rs))=List.exists(IsStrInForm x) rs
 fun IsStrInState x st=List.exists(IsStrInRhsOfGoal x)(subgoals st)
in
 if (IsStrInState x s) then (sleepForNSeconds n;(t s)) else (t s)
end
```

We instantiate *newTac* with the *basic* tactic (see §7.5) and an arbitrary symbol "f", to give a new tactic *newTacBasic*. Thus, we now have a tactic that simulates the behaviour of a tactic which can take a long time to complete, for some cases, and finishes immediately for others.

```
val newTacBasic = newTac 200 "f" (Rule.basic 1);
```

Possible proof states after application of *basic*:

Application of the tactic, *basic*, on **g1** eliminates **g1**, leaving **g2** as the only pending sub-goal; this application returns a singleton sequence of next-possible-proof-state, say, **next-proof-state-1**, as:

$$\textbf{next-proof-state-1: } State(G, [g(a),\ f(a) \vdash g(a)], \_)$$

Application of the *basic* tactic on **g2** eliminates **g2**, leaving **g1** as the only pending sub-goal; this application returns a singleton sequence of next-possible-proof-state, say, **next-proof-state-2**, as:

$$\textbf{next-proof-state-2: } State(G, [g(a),\ f(a) \vdash f(a)], \_)$$

Note that the proof state, **next-proof-state-2** has a sub-goal which satisfies the criteria of the presence of the predicate symbol 'f' on the rhs and hence the application of *newTacBasic* to **next-proof-state-2** will cause a delayed application of *basic*, whereas, application of *newTacBasic* to the proof state, **next-proof-state-1** will not be delayed.

Append two applications of *basic*:

We now create a tactic, *append2BasicApplications* which appends these two applications of the *basic* tactic and thus will give the following sequence of proof states: **next-proof-state-2, next-proof-state-1**.

```
fun append2BasicApplications s=(Rule.basic 2 s) |@| (Rule.basic 1 s)
```

When *append2BasicApplications* is composed with *newTacBasic*, the first element of the sequence, i.e., **next-proof-state-2**, causes a delayed application of *basic*.

Comparing – and distComp:

We now consider the following two cases of applying the composition of the two tactics: append2BasicApplications and newTacBasic:

```
val tacOldComp = append2BasicApplications -- newTacBasic
val tacDistComp = append2BasicApplications distComp newTacBasic
```

When given a time duration of 5 seconds, *tacOldComp* fails to complete, whereas *tacDistComp* completes successfully, solving all the goals. This is because, in the case of *tacOldComp*, **next-proof-state-2** delays the application of *newTacBasic*, which when applied on the second element, i.e., **next-proof-state-1** will solve the goal. Note that in this example, *tacOldComp* will complete successfully when given enough time.

In the case of *tacDistComp*, we have used the *distComp* operator, which addresses this scenario effectively and allows the application of *newTacBasic* on the second element, **next-proof-state-1**, even though the application of the first tactic on **next-proof-state-2** has not yet completed.

```
val goalStr = "(g(a)&f(a))-->((f(a)&g(a)))"; goal goalStr;
by (Rule.impR 1);by (Rule.conjL 1);by (Rule.conjR 1);

(*Using HAL's composition operator*)
timeout(fn () => by tacOldComp, 5);(*fails, for 5s timeout*)

(*Using the new distributed composition operator*)
timeout(fn () => by tacDistComp, 5); (*Succeeds*)
```

The example provided illustrates a scenario, which can manifest in many other theorem proving problem domains. An example from the inductive theorem proving domain where the distributed composition operator can be applied gainfully is the frequently occurring scenario of inductive rule synthesis. Here, the steps involved in the synthesis need to be applied in a compositional fashion.

- Let tac_1 be a tactic to identify induction rules, tac_2 the induction strategy.
- Let y_1, the first candidate produced by tac_1 be a structural induction, which is inadequate for the conjecture and does not terminate, and
- y_2 the second candidate produced by tac_1 be a more complex induction rule, one that succeeds.

We are interested in the composition of tac_1 and tac_2. To accomplish this, consider the following two scenarios:

Using the sequential composition operator, – : Using this operator, tac_2 will be applied on y_2, only after application of tac_2 on y_1 has completed. As this is a non-terminating computation, computation of $tac_2(y_2)$ will never be performed.

Using the asynchronous composition operator, distComp : Because *distComp* executes an asynchronous, *data-driven* evaluation model, $tac_2(y_1)$ and $tac_2(y_2)$ will be spawned simultaneously. Thus, though $tac_2(y_1)$ will never terminate, the successful computation, $tac_2(y_2)$ will still be performed.

7.10.2 Utility of the fastest-first tactical

As explained in §7.5.2, HAL provides two choice operators to address scenarios where evaluation of any one candidate suffices, || and |@|. In §7.6.2, we described **FF**, the fastest-first tactical for two or more tactics, a novel choice operator, which returns the fastest computing tactic and terminates the others. In Example 7.5, we provide a simple example, where use of each of the choice operators of HAL fails, whereas, **FF** succeeds.

Example 7.5 Choice involving a time consuming tactic, FF succeeds, whereas || and |@| fail

Consider a simple example, with the following goal

$$\textbf{G:}\ a \rightarrow a$$

Application of the sequent rule impR on this gives the following sequent as the new goal.

$$\textbf{g1:}\ a \vdash a$$

g1 above has identical left and right sides and so is provable by applying the *basic* tactical of HAL, explained in §7.5.2.

We now create a contrived scenario of tactic application to illustrate the utility of the FF choice operator compared to HAL's choice operators. As mentioned before, we use the tactic, *sleepTactic*, to simulate the behaviour of a time consuming tactic. It may be recollected that the tactic *all*, returns the state unchanged. Now, consider the following tactic, *delayed_all*, where, the tactic *all* is applied after a time delay. *delayed_all* gives the effect of a time consuming tactic, but one that returns the state unchanged and thus does not help in proving the goal.

```
fun delayed_all n = sleepTactic n all.
```

Consider a scenario where a choice has to be made between application of *delayed_all 100* and *basic*. Obviously, application of *basic* is the faster tactic among these two and in the case of **g1**, it helps to solve the goal as well. A choice operator which picks the faster tactic among two given tactics can be helpful to address this scenario, in a way not possible using the sequential choice operators, || and |@|.

To illustrate the utility of our new choice operator, *FF*, compare the outcome of a proof-attempt involving use of HAL's choice operators and FF as follows:

Using the FF operator, (delayed_all 100) **FF** basic

```
fun all_Basic_FFversion n = (delayed_all n) FF (basic1);
by (all_Basic_FFversion 10) (*Proves*)
```

Using ||, which commits to the first successful tactic with no backtracking; (delayed_all 100) || basic

```
fun all_Basic_HALversion n = (delayed_all n) || (basic1);
by (all_Basic_HALversion 10) (*Fails*)
```

Using the |@| operator, which combines the results of two tactics with possibilities for backtracking; though this operator allows for backtracking, as *delayed_all* eventually applies *all* successfully, no backtracking happens in this case. (delayed_all 100) |@| basic

```
fun all_Basic_HALversion2 n = (delayed_all n) |@| (basic1);
by (all_Basic_HALversion2 10);(*Fails*)
```

Using HAL's choice operators returns the result of *delayed_all* and *basic* is never applied, leaving the goal **g1** unsolved. On the other hand, the *FF* operator picks the faster tactic, which happens to be *basic* in this example and applies the same to **g1** and solves the goal.

Though this is an artificially synthesised example, it illustrates a scenario where the computation times are irregular and where the **FF** operator leads to a successful proof attempt, whereas the sequential choice operators of HAL do not.

Irregularity in computation times can manifest itself in many theorem proving situations, as it is hard to predict the time taken by a tactic on a (sub-)goal. A slow (and possibly unsuccessful) tactic can block a potentially successful proof attempt. **FF** can be optimally used to address such scenarios, by simultaneously considering all the possible options and choosing the fastest.

7.10.3 Utility of the crossTalk tactic

In §7.9.1, we described *depthCrossTalk*, a depth-first-search-based automatic tactic. *depthCrossTalk* is identical to *depth*, HAL's depth-first-search-based automatic tactic, except that it uses *crossTalk* instead of HAL's unify tactic. A recap of *depth* and *depthCrossTalk* is as follows:

```
fun depthFirst pred tac x = if pred x then all else (tac -- depthFirst pred tac)
val depth = depthFirst final (safeSteps 1 || unify 1 || quant 1);
(*Use the crossTalk tactic instead of unify in the above line; to unify across
all pending sub-goalsPass [] as the first argument to crossTalk*)
val depthCrossTalk =   depthFirst final (safeSteps 1 || crossTalk [] || quant 1);
```

We now provide an example illustrating the scenario of performing unification on a proof-state, which has sub-goals with shared meta-variable(s). *depthCrossTalk* solves the problem whereas *depth* does not. The detailed workings of the following example are given in Appendix §A 8.

Example 7.6 Collaborative unification

GIVEN: For constants, p, q, r,

1. $\forall x\, Q(x) \wedge R(x) \rightarrow P(x)$
2. $\forall x\, S(x) \rightarrow Q(x)$
3. $\forall x\, Q1(x) \wedge R1(x) \rightarrow P1(x)$
4. $R(p) \wedge R(q) \wedge R(r)$
5. $S(p) \wedge S(q)$
6. $Q1(q) \wedge R1(q)$

GOAL: $\exists x.(P(x) \wedge P1(x))$

Proof state: Applying propositional and quantification rules on the above problem, we get the following proof state with 6 sub-goals and meta-variables: `?_a`,`?_b`,`?_c`:

Listing 7.11: Example illustrating the utility of crossTalk, the collaborative unification tactic. 'connective'-L/R to the left and right sequent calculus rules for 'connective'; variables preceded with the '?' symbol denote meta-variables.

```
MAIN GOAL: (ALL x. S(x) ⟶ Q(x)) & ((ALL x. R1(x) & Q1(x) ⟶ P1(x)) &
        (S(p) &
         (S(q) &
          (R(p) &
           (R(q) &
            (R(r) &
             (Q(p) &
              (Q(q) &
               (Q1(q) &
                (R1(q) &
                 (ALL x. R(x) & Q(x) ⟶ P(x))))))))))) ⟶
                 (EX x. P(x) & P1(x))
SUB-GOALS:
 1. P(?_b), R1(q), Q1(q), Q(q), Q(p), R(r), R(q),
    R(p), S(q), S(p),
    ALL x. R1(x) & Q1(x) ⟶ P1(x),
    ALL x. S(x) ⟶ Q(x),
    ALL x. R(x) & Q(x) ⟶ P(x)
    |-  P(?_a), EX x. P(x) & P1(x)
 2. R1(q), Q1(q), Q(q), Q(p), R(r), R(q), R(p),
    S(q), S(p), ALL x. R1(x) & Q1(x) ⟶ P1(x),
    ALL x. S(x) ⟶ Q(x),
    ALL x. R(x) & Q(x) ⟶ P(x)
    |-  R(?_b), P(?_a), EX x. P(x) & P1(x)
 3. R1(q), Q1(q), Q(q), Q(p), R(r), R(q), R(p),
    S(q), S(p), ALL x. R1(x) & Q1(x) ⟶ P1(x),
    ALL x. S(x) ⟶ Q(x),
    ALL x. R(x) & Q(x) ⟶ P(x)
    |-  Q(?_b), P(?_a), EX x. P(x) & P1(x)
 4. P(?_c), R1(q), Q1(q), Q(q), Q(p), R(r), R(q),
    R(p), S(q), S(p),
    ALL x. R1(x) & Q1(x) ⟶ P1(x),
    ALL x. S(x) ⟶ Q(x),
    ALL x. R(x) & Q(x) ⟶ P(x)
    |-  P1(?_a), EX x. P(x) & P1(x)
 5. R1(q), Q1(q), Q(q), Q(p), R(r), R(q), R(p),
    S(q), S(p), ALL x. R1(x) & Q1(x) ⟶ P1(x),
    ALL x. S(x) ⟶ Q(x),
    ALL x. R(x) & Q(x) ⟶ P(x)
    |-  R(?_c), P1(?_a), EX x. P(x) & P1(x)
 6. R1(q), Q1(q), Q(q), Q(p), R(r), R(q), R(p),
    S(q), S(p), ALL x. R1(x) & Q1(x) ⟶ P1(x),
    ALL x. S(x) ⟶ Q(x),
    ALL x. R(x) & Q(x) ⟶ P(x)
    |-  Q(?_c), P1(?_a), EX x. P(x) & P1(x)
```

Next step: apply unification: Find a suitable unifier for the list of meta-variables: `?_a` , `?_b`, `?_c`, which satisfies all the 6 sub-goals. As can be worked out easily, the possible unifier(s) for each sub-goal (i.e. which make the left and right sides of the sequent identical) are as follows:

1. ?_a = ?_b
2. ?_b = (r,q,p) i.e. 3 candidates:(_b,r), (_b,q), (_b,p)
3. ?_b = (q,p) i.e. 2 candidates: (_b,q), (_b,p)
4. Unification cannot be applied successfully
5. ?_c = (r,q,p) i.e. 3 candidates: (_c,r), (_c,q), (_c,p)
6. ?_c = (q,p) i.e. 2 candidates: (_c,q) (_c,p)

HAL's *depth* tactic results in a non-terminating search:

When HAL's sequential depth-first search tactic, *depth* is applied, the *unify* tactic is used to tackle unification. As explained earlier, this tackles unification for each sub-goal. In our example here, the first unifier produced by sub-goal-2, (_b = r), results in a looping situation, resulting in a non-terminating proof search. In particular, here, the looping happens because new disjuncts are added to the right hand side of the sequent.

Given the lazy nature of the list of states returned by the *unify* tactic used by *depth*, (_b = r) is applied across all sub-goals and execution of the *depth* tactic is continued. This in turn, means application of the *quant* and *safe* tactics in succession, on the state produced after the application of (_b = r).

Even if just a re-ordering of variables may suffice to circumvent the problem faced in our contrived example, it is easy to see that the problem can be rearranged in a way that still poses the same problem. Furthermore, the effect of ordering illustrates a problem that can appear in many other forms.

***depthcrossTalk* solves the goal:**

Application of *depthCrossTalk*, the depth-first-approach-based automatic tactic which uses the collaborative unification tactic, *crossTalk* (see §7.9.1), successfully solves the goal. A summary of the workings of the proof attempt by *depthCrossTalk* is provided below. This illustrates the process of finding the

consensus unifiers, using the *crossTalk* tactic.

The unifiers are printed as Key-val pairs. e.g., for sub-goal 3, the two unifiers are: [Key= _b,Val= p] and [Key= _b,Val= q]. Only the successful attempts at finding a consensus are included in the listing below. The names of the native inference rules being applied at each step are also included, should the reader wish to work through the example.

The *STEP* numbers included can be tracked with the same in the crossTalk code fragment given earlier (see Listing 7.9). Also, for sub-goal 4, unification cannot be applied. *crossTalk* deals with such a situation by ignoring the sub-goal for the purpose of finding the consensus unifiers. But, when the next-states are returned, the unifier gets applied to all the sub-goals, including sub-goal 4.

Listing 7.12: Execution-trace of crossTalk, for given example; Finding the consensus unifiers

```
****Applying crossTalk******
STEP-2.3 Consensus is of length..3;  (_a , _b , _c) = (q, q, q)
STEP-2.3 Consensus is of length..3;  (_a , _b , _c) = (q, q, p)
STEP-2.3 Consensus is of length..3; (_a , _b , _c) = (p, p, q)
STEP-2.3 Consensus is of length..3; (_a , _b , _c) = (p, p, p)
STEP-3 Num of consensus unifiers: 4 ****
```

Finding more consensus unifiers: As observed in the description of *crossTalk* earlier, the states are returned as a sequence, to adhere to the type definition of a tactic. Thus, in the rest of this trace, after the application of *crossTalk*, the state corresponding to the first unifier in the list of consensus unifiers is used to generate the corresponding next-proof-state. This proof state is used for the subsequent inference steps.

Referring to the trace given above, the first candidate in the sequence of next-proof-states is generated by applying the unifier (_a , _b , _c) = (q, q, q). This is applied to all the 6 sub-goals and execution of *depthCrossTalk* is continued on the resulting state.

Listing 7.13: Execution-trace of crossTalk, for given example; Finding more consensus unifiers

```
**** Applying safe ******
[basic , basic , basic , basic , &-L,  |-R,  ->-R, ~-L,  ~-R,  exL , allR , &-R,
|-L,  ->-L,  <->-L,  <->-R, basic , &-L,  |-R,  ->-R, ~-L,  ~-R,  exL ,
allR ,  &-R,  |-L,  ->-L, <->-L,  <->-R ]

**** Applying crossTalk ******
```

```
*** !!! STEP-1 Num of sub goals..!!!3;; !!! Meta-variable list: [] !!!

**** Applying quant ****** [allL, exR]
**** Applying safe ******
[basic &-L, |-R, ->-R, ~-L, ~-R, exL, allR, , &-R basic &-L, |-R, ->-R,
 ~-L, ~-R exL, allR, &-R, |-L, ->-L, basic, &-L, |-R, ->-R, ~-L, ~-R,
exL, allR, &-R, |-L, ->-L, <->-L, <->-R, basic, &-L, |-R, ->-R, ~-L,
~-R, exL, allR, , &-R, |-L, ->-L, <->-L, <->-R]

**** Applying crossTalk ******
*** !!! STEP-1 Num of sub goals..!!!5; !!! Meta-variable list: _d , _e , !!!
STEP-2.3 Consensus is of length..2 ; (_d , _e ) = (q, q);
STEP-3 Num of consensus unif envs is 1****
```

Using the consensus unifiers : From the above, we get

$$(\_a\ ,\ \_b\ ,\ \_c,\ \_d,\ \_e) = (q,\ q,\ q,\ q,\ q)$$

Listing 7.14: Execution-trace of crossTalk, for given example; Using the consensus unifiers

```
**** Applying safe ******
[basic, basic, basic, &-L, |-R, ->-R, ~-L, ~-R, exL, allR, &-R, |-L,
->-L, basic, basic, &-L, |-R, ->-R, ~-L, ~-R, exL, allR, &-R, basic,
basic, basic, basic, basic, &-L, |-R, ->-R, ~-L, ~-R, exL, allR, &-R,
|-L, ->-L, <->-L, <->-R]

(ALL x. S(x) --> Q(x)) & ((ALL x. R1(x) & Q1(x) --> P1(x)) &
 (S(p) &
  (S(q) &
   (R(p) &
    (R(q) &
     (R(r) &
      (Q(p) &
       (Q(q) &
        (Q1(q) &
         (R1(q) &
          (ALL x.
             R(x) & Q(x) --> P(x))))))))))) --> EX x. P(x) & P1(x))

                                                    No subgoals left!
```

Thus, *depthCrossTalk*, via application of the *crossTalk* tactic to perform unification across the 6 sub-goals, with shared meta-variables has circumvented the looping situation caused by an incompatible unifier, which led the sequential *depth* tactic of HAL (which uses the sequential *unify* tactic to perform unification) to a non-terminating search.

7.10.4 Programmability: new concurrent proof search procedures

As mentioned in 2, ease of prototyping new concurrent proof search techniques has been one of our claims in this work. While it is hard to quantify *ease of prototyping* as a performance-metric, we illustrate this aspect, via a list of novel concurrent proof search procedures which we have engineered, using the concurrent operators implemented in our prototype. Some simple examples of the same were provided earlier in §7.9.2. In this section, we provide more examples to demonstrate the range of experimentation, ease of prototyping and incremental development of new proof search procedures using the asynchronous tacticals and operators implemented in this prototype.

Distributed quantifier tactic and corresponding depth-first search: The sequential tactic *quant* uses the composition operator -- to compose the unsafe quantifier rules. Thus, it suffers from the limitations imposed by the demand-driven lazy evaluation model. We have replaced -- with the distComp operator, to synthesise a new tactic called *dist_quant*. This applies the unsafe quantifier rules using a data-driven evaluation model and thus, composition operations are performed as and when the candidates are available, instead of waiting for the previous computation(s) to complete. To enable use of this tactic in an automatic setting, a depth-first search procedure has been engineered, by replacing the *quant* tactic in the implementation of *depth* with the *dist_quant* tactic.

```
fun quant i = (allL i -- try (exR i)) || exR i; (*Original quant tactic*)
fun dist_quant i =  ((allL i) distComp (try(exR i))) || (exR i)

val depth=depthFirst final (safeSteps 1 || unify 1 || quant  1); (*Original
    depth tactic*)
val depth_dist_quant=depthFirst final (safeSteps 1 || unify 1 || dist_quant  1);
```

Order of applying the unsafe quant rules: The order of application of the unsafe quantifier rules is crucial for a successful proof search. However, this order cannot be pre-determined. To address this, we have used the FF operator to try two automatic searches with different orders of application of the unsafe quantifier rules.

```
fun quant i = (allL i -- try (exR i)) || exR i; (*Original quant tactic*)
fun quant_diffOrder i=(exR i distComp try (allL i)) ||(allL i); (*Different
    order of rules*)
val depth_quant_diffOrder=depthFirst final (safeSteps 1||unify 1||
    quant_diffOrder 1);
```

```
val depth=depthFirst final (safeSteps 1 || unify 1 || quant 1); (*Original
    depth tactic*)
val depth_fastest_UnsafeQuantOrderings = depth FF depth_quant_diffOrder
```

Safe and unsafe tactics: One of the benefits of parallelisation is the scope to isolate choice points where the order of application matters and use parallelisation at such choice points. We use parallelisation to formulate a new way of applying the *safe* and *unsafe* tactics, *safeAndParallelUnsafe*. It applies the safe rules till they can't be applied any more and then uses *distComp* to compose unify and *dist_quant* (also implemented using distComp). A depth-first-approach-based automatic procedure has been synthesised using *safeAndParallelUnsafe*.

```
fun safeAndParallelUnsafe i=(safeSteps i)||((dist_quant i) distComp (unify i))
val depth_safeAndParallelUnsafe = depthFirst final (safeAndParallelUnsafe 1)
(*Using distComp to handle interaction between unify and quant*)
val depth_distComp=depthFirst final ((safeSteps 1)|| (unify 1)distComp(quant 1))
```

Depth-first search using crossTalk and safeAndParallelUnsafe: To bring together the benefits of *crossTalk*, the collaborative unification tactic and a data-driven implementation of the unsafe rules, we have synthesised *safeAndParallelUnsafe_crossTalk* and a corresponding depth-first search.

```
val safeAndParallelUnsafe_crossTalk=(safeSteps 1) || ((dist_quant 1) distComp (
    crossTalk []))
val depth_safeAndParallelUnsafe_crossTalk=depthFirst final (
    safeAndParallelUnsafe_crossTalk)
```

Fastest-first of depth and depth_SAT In §7.9.3, we described *depth_SAT*, the depth-first search procedure, which uses the SAT-based counterexample checker for propositional (sub-)goals. As it may be hard to judge the applicability of depth_SAT (i.e. the presence of propositional sub-goals), it may be hard to predict the performance and/or success rate of *depth_SAT*, in comparison to *depth*. So, we have synthesised a new tactic, *FF_depth_Or_depth_SAT* using the FF operator

```
(*Using the FF tactical to return the fastest returning strategy*)
val FF_depth_Or_depth_SAT = depth FF depth_SAT
```

The new tactics described here demonstrate the multilayered approach of encapsulating concurrent approaches as concurrent tactics and tacticals and using them incrementally to prototype new procedures. As can be seen from the code samples, implementation of these new concurrent search procedures, requires very little developmental effort.

Furthermore, as the details of concurrent programming have been abstracted away, the user can focus on using these concurrent tools to tailor customised concurrent solutions for their individual problem scenarios. As these extensions are guaranteed to be sound, the proof guarantees of the LCF approach still hold, for the concurrent proof search procedures.

If a user comes up with a requirement of a new concurrent technique to address her problem scenario, an attempt can be made to use the existing concurrent machinery to implement the same. Else, the abstraction based, multilayered approach demonstrated in our prototype can be used to implement the new technique.

7.10.5 Developmental methodology

The multilayered approach that we have used to implement concurrent, sound, programmable extensions, for an LCF-style prover demonstrates our developmental methodology of using a functional programming language and programming abstractions. The use of these features is an excellent fit for the LCF class of provers, in particular, given the functional programming origins and the programmability focus of the LCF school of theorem proving.

As described in previous sections, (§7.6, §7.8.3), keeping in line with our multilayered approach, we have developed programming abstractions encapsulating the concurrent techniques employed. We have kept our implementations purely functional. Since these abstractions have been implemented as higher-order ML functions, they should be readily portable, to other concurrent ML platforms, with little or no modification. The portability should be extendable to other LCF systems as well, as long as the implementation has avoided non-functional aspect or has abstracted them adequately and they can be ported to an ML dialect with language-based concurrency support.

7.11 Related work

In the work described in this chapter, we have addressed the topic of applying parallelisation to LCF-style theorem provers, by developing and implementing a multilayered approach to incorporate concurrent techniques into an LCF-style theorem prover. As demonstrated, this approach promotes *programmability* and *ease of experimentation*

with new concurrent search procedures and these criteria have been the foci of our investigation. Implementation of the multilayered approach, in turn, is embedded within the LCF prover and is coded in the implementation language of the LCF prover. As explained earlier, the framework addresses our objectives in the following manner:

Use of programming abstractions Our multilayered approach employs programming abstractions encapsulating concurrent techniques. This has been done to facilitate incremental development, portability and programmability.

Programmability, customisability, incremental development Allows for users to develop their own extensions incorporating concurrent programming techniques. The abstractions that form the first layer can be used by the user/developer to build on the already implemented suite of techniques and to implement new techniques.

Portability Porting to other development platforms and deploying on different computing architectures.

Ease of prototyping and experimentation The multilayered approach helps to isolate design and implementation and enables faster experiment set-up with minimal developmental effort.

As discussed in chapter 2, the field of automatic theorem proving has seen a fair amount of published work related to the application of concurrent/parallel/distributed approaches. For LCF systems, the focus has been primarily on automatic provers and heterogenous systems (e.g., use of SAT solvers and/or first-order theorem provers to tackle higher-order problems). Systems like OANTS (discussed in §2.2.3) provide frameworks for combining heterogeneous systems like computer algebra systems, constraint solvers, automatic first-order provers and higher-order provers.

The objectives of these approaches are primarily geared towards tackling harder problems and/or solving problems faster. Our investigation has been geared towards incorporating concurrent, programmable, useful extensions to an LCF-style prover. While incorporation of external solvers is achievable using our approach (see §7.7), it has not been the ultimate goal of our investigation.

An orthogonal observation is that the majority of published research in this area have involved significant developmental costs, which is not surprising, as concurrent-distributed programming is notoriously hard to program, debug and to carry out performance anal-

ysis. A detailed discussion of these aspects was provided in chapter 5. These huge developmental costs can be prohibitive and can stifle development of and experimentation with novel techniques that were previously not possible in a sequential setup.

In §7.11.1 and §7.11.2, we try to draw out the similarities with and differences between our work and two other systems which adopt a similar (though not identical) treatment as ours, to address the parallelisation of LCF systems:

- the metaPRL system [Hickey, 1999] and
- the Isabelle-PolyML project [Matthews and Wenzel, 2010; Wenzel, 2009].

7.11.1 MetaPRL: similarities and differences

The work discussed in [Hickey, 1999] implemented in the MetaPRL proof environment [12] is the only work we found in the literature, which shares some similarities with our implementation, though the objectives were slightly different. It uses the *ensemble* communication system, a proprietary communication system layer implemented in the functional programming language, OCaml, to distribute the load of tactic implementation across processors. The distribution is handled as a separate layer above the MetaPRL logical framework. The focus of the work has been to achieve *fault tolerance* in the context of a distributed computing environment and this has been addressed by using the fault tolerance capabilities of the *ensemble* communication layer.

They replaced the tactic implementation of the MetaPRL logical framework with a functionally equivalent distributed tactic scheduler. This allowed for compatibility with the other (sequential) members of the tactic base. Our approach achieves the same effect, as all our distributed tacticals adhere to the same type as the sequential tacticals. Thus, it allows for interoperability and compatibility.

From a developmental point of view, one drawback in the MetaPRL implementation is the presence of multiple implementation layers to engineer the scheduling. Also, the only main parallelisation technique used is *scheduling* of jobs and its application is restricted. The distribution module is implemented as an independent layer with a view towards not requiring the tactic library to be modified. But, this also restricts possibilities in which the distributed tactical could be used: (i) there is no way that the distributed tacticals can be used in an equivalent way as the original sequential tactical

[12] http://metaprl.org/

as part of a proof (ii) this in turn inhibits the possibilities of the distributed tactical being used as a primitive either to incorporate more sophisticated parallelisation techniques and/or to use them to create novel proof search procedures and to reengineer existing proof search procedures.

The key differences between our work and that of MetaPRL are as follows:

Communication between tactics: Given our use of Alice ML with its language-integrated parallelism support, as opposed to using a communication layer for handling parallelisation, we have been able to achieve communication between tactics more effectively. MetaPRL allows only for raw parallelisation with a pre-decided form of work distribution across processors. The unit work is a tactic application. The emphasis is on capitalising the scheduling and fault-tolerant capabilities of the *ensemble* communication layer. Our example of using the *crossTalk* tactic within a depth-first based proof search procedure is an illustration of a scenario where we have implemented communicating tactics.

Granularity: Term level concurrency: As explained above, the only form of parallelisation allowed is distribution of workload across processors. The *crossTalk* tactic is an example where we have been able to implement term level concurrency as the instantiation of meta-variables spanning multiple goals is done using a collaborative consensus mechanism.

Scope for user to develop extensions: Given the extra communication layer which handles scheduling, there is very little scope for users to develop their own extensions. With our multilayered approach of programming abstractions for concurrent techniques, concurrent tacticals and novel proof search procedures, entities from each layer can be used in a mix-and-match mode to build new tacticals and/or novel proof search procedures.

7.11.2 Isabelle-PolyML: similarities and differences

Our efforts to port Isabelle to Alice ML to develop the prototype of our multilayered approach were described earlier in the chapter (see §7.4). As mentioned there, an early work on the ideas of our project was presented in an Isabelle workshop which led to a useful collaboration with one of the key Isabelle developers to facilitate changes in the Isabelle architecture to address the issues that we had identified during our efforts

to port Isabelle to Alice ML. However, despite these modifications, we realised that porting Isabelle-HOL to Alice ML would still require a lot of effort, due to the way Isabelle builds HOL as a *heap*, a dump of the bindings in the top-level environment of the ML platform. Also, there was not enough enthusiasm for supporting Alice ML from the Isabelle developer's side, because of Alice ML's lack of support for *truly parallel system threads*. It was observed that Alice ML's runtime system does not support this feature [Matthews and Wenzel, 2010].

The Isabelle-PolyML project that started subsequently shares some of our objectives: of providing the concurrency support via the implementation language. The work discussed in [Matthews and Wenzel, 2010; Wenzel, 2009] report the details on the significant reworking of the ML layers undertaken to facilitate support for parallelism in Isabelle for the PolyML platform. A detailed review of this project was given in §2.2.2. Crucially, this work has involved significant modifications to both the Poly/ML language as well as the Isabelle architecture. Poly/ML and Isabelle are big and complex software systems that have evolved over more than two decades. Thus, it is hardly surprising that an effort to port Isabelle to a version of Poly/ML with parallelisation support required comprehensive knowledge of the internals of each of the systems. The work reported in [Matthews and Wenzel, 2010] has involved architects of both the Isabelle and Poly/ML systems. And as reported in the work, the Poly/ML side of the work has required reworking of many infrastructure layers: from low-level system threads to high-level principles of value-oriented programming. Substantial reorganisation of the Isabelle architecture has also been required.

The work aims to address the *multicore* architecture specifically. It is useful to point out here that this project has addressed the development of the ML language level (PolyML) support and the Isabelle modifications. Their approach has been to provide support for concurrency primitives (which is richly supported already by Alice ML) alongside many other tweaks on the PolyML design for optimal utilisation of multicore architectures. However, they do not address distributed architectures like clusters and also lack message-passing provisions, both of which are supported by Alice ML as discussed earlier in the thesis. In our view, the concurrency primitives provided in PolyML are not very user-friendly from a developer's perspective and lack many features found in Alice ML. This in turn, can be possibly due to the evolving nature of the parallel PolyML, given that this effort is the first version of parallel-PolyML. However, more crucially, there is no support for distribution in their current version of

parallel PolyML. This can be a serious limitation if one wants to perform experiments on a distributed network.

Another key difference is that their main goal has been to provide parallel proof checking capabilities in Isabelle via the PolyML platform. They have tried to leverage on the proof structure present in *Isar* documents to facilitate *implicit* parallelism. On the other hand, we have tried to provide a multilayered approach that can serve as an experimental workbench that gives the user the flexibility to quickly prototype experiments and develop their own novel search techniques incorporating concurrency and distribution.

As discussed earlier in §7.2.3, development of and experimentation with concurrency techniques is required in order to enable effective use of the same to engineer better theorem provers. Given the varied nature of problem domains of theorem proving problems and their differing structures, difficulty levels etc, it is important to tailor the concurrency techniques to a given problem. Thus, we have aimed to address different goals in our work, different from the objectives of the Isabelle-PolyML project.

7.12 Summary

In the work discussed here, we have proposed and explored the scope and utility of a previously unexplored approach to use parallelisation for LCF provers: a multilayered approach for developing sound, concurrent extensions to an LCF style theorem prover. We have demonstrated a proof-of-concept prototype for the same and have set the context for it to be ported to other LCF provers as well as allowing for incremental development and further research. Our framework allows for rapid prototyping and experimentation and incremental development of novel approaches to theorem proving exploring parallel and co-routining possibilities and approaches.

LCF style provers are particularly well suited for a modular approach, given the modularity present in the well established techniques of tactics and tacticals. Tactics and tacticals are an integral part of every LCF style prover and apart from the modularity, they are also designed to guarantee soundness. Thus, an approach to incorporate concurrent-distributed techniques while retaining the soundness aspects is particularly well suited to the LCF style provers. In this case study, our focus has been to achieve these objectives in the context of a concrete example of a prototypical LCF style first-order prover.

We have developed a proof-of-concept prototype framework for HAL, an LCF style first-order prover (without equality) that allows *programmable, sound extensions, incorporating concurrent programming techniques and enables novel proof search procedures*. This in turn, has been achieved by a clearly defined multilayered approach of developing programming abstractions (see §7.2.1), using the abstractions in turn, to implement distributed tacticals and using the distributed tacticals to implement novel proof search procedures. We have discussed earlier in the chapter (in §7.6) the distributed tacticals and novel proof search procedures that have been implemented. We have used Alice ML as the implementation language.

Our framework and the abstractions and novel tacticals developed have opened up some novel approaches like fastest-first, data-driven asynchronous execution and collaborative unification that are not possible in a sequential setting. We have illustrated scenarios where concurrency can be of use: by highlighting the limitations posed by some of the sequential tactics in HAL (§7.6.1, §7.5.3) and how an asynchronous mode of execution can help address the same. However, our current implementation runs the

processes on the same machine. Enabling distribution is one of the priorities on our agenda for further work.

Furthermore, our approach promotes *programmability*: offering users a set of concurrency primitives and abstractions, enabling them to use the same to synthesise new tactics and new search procedures. Further research is required to rigorously ascertain problem classes that will benefit from specific abstractions and concurrent techniques.

(§7.10) discusses some examples of theorem proving scenarios where our approaches can benefit. The examples also illustrate the efficacy of the *programmable* aspects of our framework by illustrating the ease of prototyping one's own search techniques incorporating concurrent techniques, to suit a particular theorem proving scenario.

We have also provided scenarios where collaborative approaches can be gainfully employed. We have implemented a novel tactic, *crossTalk* (§7.8.3) that performs unification across multiple goals, comes up with a list that serve as unifiers for all of the sub-goals. It uses collaborative exchange of partially evaluated information from asynchronous computations. The information exchanged in this implementation is fairly fine-grained, being at the term level and as such this is a good illustration of the utility of Alice ML. *crossTalk* has been used within the *depth* automatic search procedure of HAL to give a new automatic search procedure, which we have called *depthCrossTalk* and solves an example that is not solvable by HAL's *depth* tactic (§7.10.3s).

We have developed distributed programming abstractions (§7.10.5) that encapsulate the different forms of parallelism and co-routining employed in developing the multi-layered approach. The use of Alice ML has enabled us to develop these as higher-order functions. The abstractions allow for portability to other LCF settings. Also, they can be potentially reused to apply concurrent techniques to tackle other theorem proving scenarios, both within and outwith LCF style provers. E.g., with appropriate parametrisation, the *refereeAgent* abstraction can be used to tackle scenarios where a task can be decomposed as independent computations which need to agree on one or more things. The modular nature of the abstractions allows us to modify the concurrency implementations (e.g., changing the target architecture) without having to modify other parts of the system. The rich language support for concurrency and distribution provided in Alice ML has helped us greatly to do rapid prototyping of and experimentation with, these approaches.

The approach demonstrated in this prototype is applicable for any LCF prover. And

this is where our key contribution lies: using a simple prototype and a functional programming language with language-based concurrency support, we have demonstrated a previously unexplored approach to create an exploratory workbench that can be adopted by even a highly sophisticated LCF style theorem prover.

Theorem proving problems come from a variety of domains and can vary a lot in problem structure, proof hardness, solution distribution etc and each of these can benefit from application of different concurrent programming techniques. In view of this, a one-solution-fits-all approach may not work always for attempts to use concurrent programming techniques for engineering better theorem provers. Moreover, given the nascent nature of the field, it will stand to benefit greatly by prototyping novel proof search techniques, evaluating them empirically and using the feedback to reassess the prototypes. Such experimentation is also important to evaluate and choose the right technique. E.g., in the case of HAL, the application of the *unsafe* tactics : application of the unsafe quant rules *allL* and *exR* and *unify*. If order of execution does not matter, then applying an abstraction like fastest-first is obviously not going to help. This is contrast to scenarios where in the number of applications is itself potentially huge and there in fact, the lack of dependence on ordering paves the way for bulk parallel processing.

The rapid-prototyping and experimentation pros are the obvious advantages. Also, such a setup is easier to reason about and to enable a plug-and-play style of experimentation as well as to be able to exploit co-routining possibilities at the lowest levels of granularities like what we have implemented in our *crossTalk* proof tactic.

We believe that we have merely scratched the surface of the spectrum of possibilities that can be potentially realised with our multilayered approach. There are at least as many ways of using these as there are distributed algorithms and techniques. New primitives can be developed; new search procedures can be designed; the concurrent features can be made available in an interactive setting. Developments from the field of concurrent-distributed programming can be effectively employed to identify latent opportunities for concurrency/distribution and to implement techniques to leverage them. In the next section, we outline some ideas for possible future work.

7.13 Ideas for future work

Here are some possible next steps relating to the implementation details of the prototype: In §7.8.3.3 and §7.8.3.1, we outlined some possible improvements that can be done in the implementation of *crossTalk*, *depthCrossTalk* and the *refereeAgent* abstraction respectively. Earlier in the chapter, we had described how our current implementation uses multiple threads on a single machine to perform the asynchronous processes. One of the next steps for the prototype is for the computational model to be extended to work on a distributed architecture, thereby opening up more options for implementation and evaluation.

Provision of concurrent, parallel, distributed tacticals at the kernel level offers immense potential, especially for a generic-prover-framework architecture like Isabelle as these tacticals can be used by all the logics that are built on top of the (Pure) kernel. The HAL architecture shares a lot in common with the Isabelle/Pure kernel in design and many of the features implemented in our work can be readily ported to Isabelle. As discussed earlier, the Isabelle-PolyML project aims to address something very similar, specifically tuned for multicore architectures. Also, the PolyML concurrency support is not very user-friendly and development of programming abstractions as we have done in our work will be extremely tedious to program. Furthermore, there is no support for distribution in PolyML, which is a serious limitation. As discussed in §7.4, porting Isabelle-HOL(higher-order logic) to Alice ML (despite the modifications done to Isabelle's bootstrapping issues highlighted by us) may require considerable effort. We experimented with some ways around to address this by using the Alice ML component system. But, the dependencies and the bootstrapping process were too arbitrary to manage. But, this is still a promising option to follow through. There could be limitations to this depending on the latest developments and reorganisations done to the Isabelle architecture. Porting Isabelle-FOL(first-order logic) to Alice ML may still be possible and would be a good vehicle to port the abstractions developed in our current prototype.

Porting HOL-light to Alice ML is a better possibility along these directions. HOL-light is a more compact system than Isabelle and can hence be more amenable to modifications. It will require porting HOL-light to Alice ML. We have already done a OCaml-Alice ML porting exercise in our porting of HAL to Alice ML and did not face any irreconcilable incompatibilities.

Many theorem provers like *Isabelle* allow for possibilities for the user to interact with the system using ML syntax directly. Using our approach, the user can extend such a working style to a concurrent setting as well, e.g., by spawning a sub-goal to another machine or by spawning a potentially resource-heavy computation or even a refutation finder to another machine in an asynchronous manner while continuing with the rest of the proof. This holds for the automatic setting as well for re-implementing existing proof search procedures using the concurrent tacticals.

Another possible line of research is the identification of classes of problems and characteristics of problems that are most likely to benefit by incorporating concurrency and distribution. This can be part of an evolving iterative process of the study informing the development of the primitives and tacticals and novel proof procedures and finding new classes of problems that will fit the bill well for a given novel proof procedure. A particular case of the above can be the investigation of mathematical formalisations and mathematical proofs and identifying latent co-routining and parallelisation opportunities therein and coming up with specific abstractions and tacticals to tap those opportunities.

The work on *scientific community metaphor* [Kornfeld and Hewitt, 1981] and the account of the dynamics of how a famous mathematical proof was discovered by the synergetic interaction of a team of mathematicians as described in [Waerden, 1971] provide a cognitive motivation to investigate this particular class of problems. This could then make an interactive theorem prover, in its incarnation of mathematical assistant to become more attractive to human mathematicians as it can potentially do things that a human mathematician cannot do; e.g. of pursuing 10,000 possibilities all at once and allowing for inter-process communication !

Chapter 8

Conclusions

In this thesis, we have proposed an implementation methodology for application of concurrent techniques to theorem provers. The methodology is oriented towards facilitating, ease of prototyping of and experimentation with concurrent techniques, to engineer novel proof search procedures. Our methodology advocates the use of :

- a functional programming language with language-based support for concurrency and distribution and
- programming abstractions to encapsulate the concurrent techniques used, enabling effective separation of design and implementation.

The advantages of the individual components of this methodology, are widely known and acknowledged. However, published research in the theorem proving field does not show evidence of widespread adoption of such an approach, in the context of theorem proving systems incorporating concurrent techniques. We hope that the work discussed in this thesis adequately highlights and reiterates the advantages of these features and initiates a move towards a wider adoption of these features in the implementation methodology used to parallelise theorem provers. Our approach is particularly relevant, in the exploratory investigation phase of developing parallel theorem provers, because of the ease of prototyping and experimentation, facilitated by it. As the approach facilitates an effective separation of design and implementation, once a near-optimal approach has been identified, it can be ported/reimplemented in other (production-quality) systems as well.

We used a concrete instance of this methodology: using Alice ML as the implementation language and encapsulating the concurrent techniques used, as programming abstractions, via higher-order Alice ML functions. We demonstrated the utility of our methodology, by applying it to explore some previously unexplored parallelisation approaches/opportunities, in two diverse theorem proving flavours (SAT, LCF style first-order theorem proving), developing proof-of-concept prototypes for the new approaches developed.

Each case study has been explored in line with the two-fold hypothesis of this work (§3.1): developmental/implementation level and object-level. The object-level investigation relates to the scope and efficacy of using concurrent approaches for the theorem proving scenario considered and the consequent gains made. The developmental level investigation focuses on illustrating the utility of our implementation methodology, in terms of: ease of prototyping, experimentation, exploratory investigation and portability.

Propositional satisfiability (SAT)

Hybrid SAT solver: We investigated the potential to synergetically use two complementary SAT approaches (DPLL and Stalmarck), in a co-operative manner. This approach has been implemented using an asynchronous mode of execution and so, it enables dynamic exchange of information and allows the solvers to be run independently on different machines, enabling optimal utilisation of distributed architectures. Empirical data showed performance gains over sequential counterparts. The programming abstraction developed encapsulates the asynchronous interaction of the two solvers. This abstraction was (re)used to prototype and experiment with two further hybrid solvers, thus illustrating the utility of our implementation methodology.

Concurrent Stalmarck: As a piece of exploratory research, a novel concurrent algorithm for SAT was developed, by applying various concurrent techniques to an established algorithm, the Stalmarck algorithm. A new approach to work partitioning for SAT (different from the guiding-path based ones found in parallel SAT literature) has been realised in this implementation. Using the data-driven behaviour of Alice ML, work-consumption has been implemented such that the overheads of communication are avoided

and programming abstractions have been developed for the same. Empirical data showed performance gains over sequential counterparts. An abstraction encapsulating the concurrent approach adopted to implement the *saturation* technique has been developed, allowing for it to be reused to address similar scenarios.

LCF style first-order theorem proving: A multilayered approach, to incorporate concurrent techniques for proof search, in an LCF prover. The approach focuses particularly on programmability and portability. The programmability aspects allow users to *program* their own novel proof search procedures using the facilities provided via the framework. Our proposed approach consists of the following three layers:

Programming abstractions: encapsulating concurrent techniques

Concurrent tacticals: i.e. concurrent control structures for applying tactics, implemented using the respective programming abstractions

Novel proof search procedures: engineered using the concurrent tacticals and the sequential ones

8.1 Why and how to parallelise a theorem prover

In chapter 5, we provided arguments for the need for parallelisation of theorem proving, from the perspective of imperatives of the hardware field as well as the theorem proving-domain perspective. We listed some of the specific challenges for parallelisation, posed by the theorem proving domain, e.g. irregular search spaces, which in turn, make effective work-partitioning and load-balancing difficult. In the same chapter, we provided a prescriptive analysis of desirable criteria for implementing the novel approaches enabled by the use of concurrent techniques. These can be summarised as follows:

Exploratory research, programmability, easy prototyping: Theorem proving problems come from a variety of domains and even within the same formalism, problem classes can differ greatly in their structure, hardness and solution distribution. Potentially, each can benefit from employing a different set of concurrent technique(s). Thus an iterative process of implementation and empirical eval-

uation on particular problem classes aimed at achieving an optimal design can be of immense use. These call for the following two things: ease of experimentation and programmability[1]. The need for the exploratory research phase is further accentuated both by the sensitivity of parallel applications to implementation efficiency and the relatively nascent status of the field of parallel theorem proving.

However, concurrent applications are notoriously hard to program, debug and carry out performance analysis on. Thus, an implementation approach that aids modularity and easy prototyping of application of concurrent techniques can greatly aid the enterprise of exploration. Furthermore, it can also aid the evaluation phase by facilitating easy prototyping of multiple systems, applying different techniques, thereby enabling the analysis of the relative performance of the systems.

Isolation of design and implementation, portability, incremental development: It is important for the utility of a concurrent approach for a given theorem proving scenario, to be investigated, separately from the effectiveness of a particular implementation of that approach. This calls for an implementation approach that aids effective isolation of design and implementation. An added advantage of such an approach is the potential for portability of an approach to different platforms, with minimal developmental effort. The rapid pace at which parallel architectures are evolving further accentuates the need for portable implementations. Another desirable criteria is incremental development: to facilitate building on existing functionality to incrementally develop variations and new features.

These implementation considerations were addressed in our developmental hypothesis given in §3.1, where we claimed that use of a functional language with language-based support for concurrency enables easy prototyping of application of concurrent techniques to theorem proving and the use of programming abstractions to implement the concurrency techniques promotes portability, incremental development and aids effective isolation of design and implementation.

In chapter 5, we explained the suitability of a functional programming language for implementing concurrency. In particular, one that provides language-based (as opposed

[1]By programmability, we mean the provision of concurrency primitives and giving the user the flexibility to easily prototype their own experiments with minimal developmental effort.

to API based) support for concurrency and distribution. Another powerful feature that fits naturally in a declarative concurrency setting is that of implicit dataflow synchronisation which addresses the task of data synchronisation, a challenging issue for concurrent programmers. We explained how Alice ML, our implementation language in this work, is a concrete example of one such language.

8.2 Novel concurrent approaches for SAT: knowledge-sharing, lateral-thinking, co-operative frameworks combining complementary approaches, large scale parallelism

In chapter 6, the novel approaches developed for SAT were explained along with details of implementations of proof-of-concept prototypes (developed in Alice ML), for the same.

8.2.1 Hybrid SAT solvers

In the hybrid solver investigation, we had set out to investigate the utility of the Stalmarck clause learner, when used along with the DPLL algorithm, with the two algorithms working as independent processes, in an asynchronous manner, thus allowing for dynamic exchange of learned clauses and potential pruning of search spaces. The motivation behind the development of this solver was to

1. explore the scope of employing an asynchronous approach;
2. enable utilisation of multiple workstations by using a clause learner that can work independently from the DPLL algorithm;
3. study the utility of the Stalmarck clause learner, to address DPLL's inability to leverage on implicit structural information; and
4. extract a co-operation framework (as a programming abstraction) where multiple clause learners (possibly based on different approaches, but all working on the same problem) can be used simultaneously.

We used asynchronous message-passing, for communication, in our implementation and used an iterative (as opposed to recursive) implementation of the DPLL algorithm. For the empirical tests, for comparing the behaviour of the hybrid SAT solver(s), we considered three different problem classes: Pigeon-hole, Urquhart and Random3SAT.

The first two problem classes are known to be difficult for the stand-alone DPLL algorithm, despite possessing implicit structural information, which the DPLL algorithm fails to leverage on, an aspect where the Stalmarck algorithm is known to fare better. Thus, a hybrid architecture where the Stalmarck clause learner works with the DPLL solver, in a co-operative set up can benefit this scenario. This is confirmed by our empirical data, where the hybrid solver shows uniform performance (time, size of search space) gains for these two problem classes.

To further explore this topic of comparison of solvers, we developed another hybrid solver, by replacing the DPLL solver, with DPLL-CDCL, a DPLL solver, augmented with CDCL, an established clause-learning technique. This is an illustration of the exploratory investigation aspect, one of the running themes of this thesis. Prototyping of this solver was done using the abstraction (*doDPLLwithHelper*, §6.3), and required minimal concurrent programming developmental effort.

We carried out further experiments to study the comparison of this solver with and without the Stalmarck clause-learner and concluded that for these two problem classes, the DPLL-CDCL-Stalmarck was the fastest, among the 3 solvers considered. However, there was not a significant difference between the DPLL-Stalmarck and DPLL-CDCL-Stalmarck solvers. This tells us that the performance gains for these two problem classes, should be primarily attributed to the Stalmarck clause-learner, rather than CDCL. This empirical behaviour can be explained as follows: the CDCL technique is embedded within the DPLL algorithm, and thus suffers from the same limitations as DPLL, with respect to these two problem classes.

With the Random3SAT, the behaviour was not uniform. And in some cases, the DPLL algorithm performs better without the use of the CDCL technique, thus raising questions about the utility of a clause learner at all, for these problem instances. Management of learned clauses is a known challenge for the use of clause-learners, with the DPLL algorithm. Our focus in this work has been to compare the utility of the clause learners, particularly in an asynchronous setup. To this end, the empirical data from this problem class did not provide us a clear picture, with respect to the utility of the

clause-learners.

Another orthogonal issue is that clause-learners that can work independently and autonomously, can promote the utilisation of distributed architectures, e.g., a cluster of workstations, as they can be run on different machines and the DPLL can be run on a different machine. In our hybrid solver implementations, the Stalmarck clause-learner works independently and autonomously. It works in an independent thread in the multithreaded version and on a different machine, in the distributed implementation. The CDCL learning mechanism computes its learned clauses, by analysing the conflicts arrived at, in the DPLL search tree. Thus, it is embedded within the DPLL algorithm and it will not be effective to decouple the CDCL algorithm to work as an independent clause learner, working on the same problem. Moreover, it spans the search space in the same way, as the DPLL algorithm does and so, the learned clauses will arise from the same search tree.

We can draw the following conclusion, from the DPLL-based, hybrid SAT solver(s) investigation, that we have conducted:

- Use helpers (clause-learners) that address DPLL's limitations
- Use helpers (clause-learners) that can work independently and autonomously
- When focusing on a particular problem class, identify an algorithm(s) that are known to be successful for that class and try to build a clause-learner, based on that algorithm, ensuring that the algorithmic features that favour that particular problem class are retained.
- Use an abstraction based approach to building the hybrid architecture, so that prototyping of new solvers can be done, with minimal effort, aiding the exploratory investigation phase

In these efforts, it was important to be able to easily prototype new solvers that combined different techniques, and to quickly set up and easily analyse experiments. Our methodology of using programming abstractions achieved their purpose in the incremental development of new solvers, e.g. when there was a need to investigate the behaviour of problem classes by adding a new technique, as in the DPLL-CDCL-Stalmarck case mentioned earlier. The separation of design from implementation, enabled by the methodology, helped in the performance analysis of the solvers as well.

A possible option for future work is to port the high-level design of the hybrid solver(s), to a C-based state-of-the-art solver and study the empirical behaviour. Our empirical data was collected using multithreaded versions of the hybrid solvers. In future, we hope to run experiments for the distributed versions of the hybrid solvers. Of particular interest is the DPLL-ConcurrentStalmarck implementation, described in §6.4. This uses our novel algorithm, *ConcurrentStalmarck* (instead of the Stalmarck algorithm) in the DPLL-Stalmarck architecture. *ConcurrentStalmarck* is amenable to large-scale parallelism and thus is well-placed to utilise a distributed architecture.

8.2.2 Concurrent Stalmarck

Effective task-partitioning (for parallelisation) and work-allocation (load-balancing) are crucial issues for optimal utilisation of distributed architectures, e.g. a cluster of workstations. Most parallel SAT solver implementations, reported in published research, are DPLL-based systems. DPLL is a tightly-coupled, state-based algorithm, making effective task-partitioning, a challenging task. A vast-majority of DPLL-based systems, use *guiding-path* as the task-partitioning technique, where the tasks are essentially sub-trees, of the depth-first-search tree (of the DPLL algorithm). The irregularity of the search spaces and inability to predict completely the hardness of a (sub-) problem, makes effective load-balancing very difficult, for this form of task-partitioning. Our new algorithm, *ConcurrentStalmarck*, addresses these twin aspects.

ConcurrentStalmarck is a new algorithm that we have developed, by applying concurrent techniques, using asynchronous message-passing style communication. This new algorithm is amenable to large scale parallelism and demonstrates a new form of task partitioning and work-consumption. We implemented a proof-of-concept prototype of this algorithm, using Alice ML. In this prototypical implementation, the new form of task partitioning and work-consumption were realised by using the features of dataflow synchronisation and the data-driven behaviour facilitated by the incremental evaluation feature of Alice ML. These features are central to functional languages that provide language-based concurrency support, as opposed to API-based support.

The Stalmarck algorithm is a tautology checking algorithm. It assumes the given formula to be true and tries to derive a contradiction. §4.5.3 gives a detailed description of the algorithm. The key insight for the design of our new algorithm has been the fact that the recursive applications of the branch-merge rule can be flattened, as

the operations are associative and thus independent of the order of execution. This flattening step gives a pool of tasks (new form of *task-partitioning* for SAT). As the order of execution of these tasks does not matter, we have implemented a simple form of work-consumption (a.k.a work-stealing, in the literature), where by, any process which becomes free, picks up a task from the pool of tasks mentioned earlier. Work-consumption (load-balancing) related communication is thus negligible.

We carried out empirical tests by comparing the performance of the multithreaded version of the concurrent prototype, with the sequential counterpart and observed significant performance gains for the same. Given that the algorithm's design is well-placed to utilise a distributed architecture, in future, we hope to perform empirical tests on a distributed implementation of this prototype.

The *saturation* technique, tries to extract new information, until no more new information can be found, by applying a set of inference rules. In each round, it absorbs the newly found information. However, in a sequential setting, application of the saturation technique, involves waiting for the completion of the computation of all candidates being considered in an iteration, before deciding to perform the next iteration.

We extracted a programming abstraction, from our concurrent implementation of the *saturation* technique, used in the Stalmarck algorithm. This can be potentially used to implement concurrent approached to tackling other saturation-based algorithms or similar scenarios.

8.3 A multilayered approach to develop programmable, sound extensions, for an LCF prover

In §7.3.1, our multilayered approach to tackle the specific issues for addressing usage of concurrent techniques in an LCF prover were explained and implementation of a proof-of-concept prototype of this approach was described. The importance of programmability was mentioned earlier. The LCF style of theorem proving, is a particularly good candidate for providing concurrent *programmable* extensions. Programmability forms a core focus of the LCF school of theorem proving. It provides tactics and control structures to apply them to support interactive theorem proving, enabling the user to program their own proof search procedures. This philosophy can be extended

to include incorporation of concurrent techniques as well.

As mentioned before, we have accomplished this via a multilayered approach: use programming abstractions to implement the concurrent techniques; use these to develop novel tacticals (control structures to apply tactics), incorporating concurrent techniques; and use these to develop novel proof search procedures.

It should be pointed out that the multilayered approach can be applied, to address any LCF prover. In this work, this multilayered approach has been implemented as a proof-of-concept prototype, implemented in Alice ML. The approach has been applied to HAL, a prototypical LCF style first-order prover and provides a suite of concurrent tacticals and novel proof search procedures. The following new concurrent tacticals are provided:

Fastest first approach *FF* is a choice operator for applying tactics. It spawns concurrent evaluation of the options and picks the tactic that finishes first (a.k.a fastest-first approach)

Distributed composition The data-driven behaviour facilitated by the incremental evaluation feature of Alice ML was used to implement *distComp*, a novel operator for composing two tactics, say t1, t2. Tactics take a state and return a sequence (lazy list) of states, each of which is a possible next-state. Composition of t1 and t2 involves application of t2 on the sequence of states returned by t1. However, in the sequential case, application of t2 on the i^{th} element can start only after application of t2 on all the previous elements have been completed. Using *distComp*, t2 gets applied as and when data is available and thus application of t2 on the i^{th} element starts as soon as it is available and does not depend on the status of evaluation of the application of t2 on other elements of the sequence.

Simultaneous proof-refutation attempts on a propositional (sub-)goal: Use the power of asynchronous execution to tackle a propositional (sub-)goal, by spawning proof and refutation attempts simultaneously, returning the fastest. An external SAT solver is used to perform the refutation attempt on the propositional (sub-)goal. The SAT solver developed in the SAT case study has been (re)used here.

CrossTalk: using information exchange for unification In HAL, the *unify* tactic is provided to perform unification on a (sub-)goal. It tries to unify a formula on

the left with a formula on the right of the sequent. There may be several such pairs and hence several possible unifiers. Thus, one goal can potentially produce multiple candidates for the unifier. When there are multiple sub-goals involved with shared meta-variables, a consensus unifier(s) needs to be computed that will serve as a unifier for each sub-goal. The *crossTalk* tactic tackles this by spawning the unification attempts concurrently and using a referee to compute the consensus candidate.

This application is representative of the generic situation where a meta-variable (a shared datum) is shared between multiple goals. Each goal can pose certain constraints on and/or post suggestions for the potential instantiation and the suggestions have to be mutually consistent. The conflict resolution/consistency decision is made in our implementation by a *referee* (§7.8.2). A similar approach can be extended to tackle other scenarios, where a shared resource requires instantiation. A variation of this, is a situation where partial instantiations are communicated either in a peer-to-peer style or via a referee.

These tacticals were used within automatic proof search procedures based on the depth first approach. The *crossTalk* tactic was used within a depth-first approach based automatic proof search procedure and fared better than the same procedure which used HAL's *unify* tactic.

Examples were provided to illustrate the power of programmable extensions. Some of these are contrived examples, designed to illustrate the specific features of a concurrent tactical or proof search procedure and the possible proof search procedures that can be *programmed* to address the example. Nevertheless, they illustrate typical scenarios, encountered often in interactive theorem proving.

As an example, consider a scenario where the user encounters a non-obvious choice of inference rule-application. Our prototype allows the user to *program* their own concurrent proof search procedure to tackle this scenario, with minimal developmental effort, using the concurrent tacticals and operators provided in the prototype. There are various ways proposed in published research for tackling the task of inference rule selection, e.g. command-suggestion mechanism, proposed in [Benzmüller and Sorge, 2000; Benzmüller et al., 2008] (discussed in §2.2.3). However, we are using this scenario, merely as an example to illustrate the power of programmable extensions, realised via our multilayered approach and do not aim or claim to address inference

rule-application choice, as such.

Our experience with this case study shows that given an ML language with language-based support for concurrency and distribution, and an LCF prover, the prototyping and implementation is relatively easy once a basic group of concurrent tacticals have been implemented. In fact, the harder question is: how and where to use the concurrent tacticals and what novel proof search procedures to engineer using them? This in turn, is dependent on the theorem proving scenario being tackled and the problem class being considered, thus reiterating the importance of programmability and exploratory investigation.

8.4 Utility of our implementation approach

The use of Alice ML and abstractions has greatly enabled easy prototyping in both the case studies. Furthermore, it greatly helped to setup experiments and analyse the empirical behaviour, with relative ease. The modularity aspects and ease of prototyping, promoted by the use of abstractions, allowed for fast setup of experiments, comparing the performance of variants of a system, i.e. use the prototype of a system and set up variants of the systems differing in certain specific conditions.

In §5.5.2, we explained how concurrency and distribution features can be included in a declarative model and how Alice ML implements them. In particular, the Channels feature of Alice ML is useful for implementing message-passing mechanisms. This was used to implement message-passing mechanisms in *DPLL-Stalmarck*, *Concurrent Stalmarck* and *crossTalk*.

In §5.5.2.1, we explained the *data-driven* behaviour exhibited by the incremental evaluation feature, a consequence of implicit data-flow synchronisation. In §5.6.2, we explained how this is supported in Alice ML. The data-driven behaviour has been used to address the following scenarios:

Concurrent Stalmarck: To implement work allocation, with minimal communication;

Distributed composition operator: As explained above.

Interaction of the solvers in the hybrid solver and implementation of the *saturation* technique of the Stalmarck algorithm in a parallel setting were encapsulated as pro-

gramming abstractions, thus promoting clarity of design, modularity and portability and can thus potentially be ported to other platforms with minimal developmental effort.

The use of Alice ML is well-suited for the LCF case study given that it is an ML based language. However, for the SAT case study, it may be better to make use of the easy prototyping facilities to carry out an iterative process of development and evaluation and when an optimal design has been arrived at, then, port the implementations to optimised state-of-the-art (possibly C-based) SAT solvers, which given their advanced state of optimisations can handle really large problems.

8.5 In a nut shell...

In this thesis, we have proposed an implementation methodology to incorporate concurrent techniques in theorem provers, by using a functional programming language, with language-based support for concurrency and distribution and programming abstractions to encapsulate the concurrent techniques employed.

We have shown the scope and efficacy of applying concurrent techniques, to synthesise novel proof search procedures in two diverse theorem proving settings. The approaches developed are better placed to utilise large scale parallel processing resources and employ novel computational patterns that are not possible in a sequential setting. The novel procedures show performance gains, compared to their sequential counterparts, in many cases and no significant slow down, in the other cases considered.

The proof-of-concept prototypes implementing these novel approaches, were developed in Alice ML and showed performance gains in some cases. These prototypes and the exploratory investigations made feasible by them, illustrate the utility of our proposed methodology, in terms of ease of prototyping, (with minimal developmental effort) and ease of experimentation, by enabling fast prototyping of variants of a system, to carry out relative performance evaluations of the same. The separation of design and implementation, facilitated by the use of programming abstractions, enables the design of these approaches to be ported to other systems as well, e.g. once a near optimal design has been arrived at, in the exploratory investigation phase, the design can be ported to other (possibly state-of-the-art) systems.

8.6 Directions for future research

8.6.1 Ideas for future work related to the case studies of SAT and LCF

In §6.10, we outlined some future research possibilities arising out of the work carried out as part of the SAT case study. These included ideas for the approaches adopted as well as the implementation aspect. In §7.13, we outlined some possible ideas for extending the approach implemented for LCF style proving. In the next few sections, we provide some thought experiments for further applications of concurrent techniques to theorem proving.

8.6.2 Proof and refutation

There are many situations in which the current (sub)conjecture is false and time spent trying to prove it is wasted. This can happen even when the initial conjecture is true. For instance,

- An intermediate lemma is speculated, say, by a critic, and instantiated in a way that makes it false.
- A conjecture is over-generalised, say, by a critic.
- A case split, e.g. $P \vee \neg P$ is made even though one of these cases, say P, is already true in the current case. The $\neg P$ case now contains contradictory hypotheses, $P \wedge \neg P$,. So, it is false. If this contradiction can be detected then this contradictory case can be concluded. A similar behaviour has been implemented in our hybrid solver, *DPLL-Stalmarck*, where the DPLL solver can possibly abandon the search of a sub-tree corresponding to a case-split, because that particular literal has already been found to be false by the Stalmarck solver.

  This pattern can be extended more generally to address scenarios where a proof step is made that could have resulted in a false (sub-)conjecture. Then, it is worth investing some effort in detecting this falsity without investing a lot of wasted effort in trying to prove it. In a sequential system, it is usually worth investing only a small amount of refutation effort. However, if the proof and refutation

attempts can be spawned as asynchronous processes, preferably in different processors, then the overheads introduced by the asynchronous refutation attempt can be kept minimal.

Isabelle's counter-example finder, *Quickcheck* can be used to address this scenario. But, by itself, it does not provide an ideal solution. Firstly, Quickcheck may not find the counterexample, whereas a more sustained search would. A lot of work may now be spent on a doomed proof attempt. Secondly, Quickcheck will only work for purely universally quantified (sub-)conjectures. False conjectures containing existential quantification will require refutation, i.e., proof of their negations, not just counterexamples. For both these reasons, a better solution would be to set up two parallel tasks: one to continue the proof of the (sub-)conjecture and one to refute it, i.e. prove its negation. By providing for resource management facilities, threads can be assigned priorities. Now, a variety of heuristics could be used to decide how much resource should be devoted to each task. For instance, Quickcheck might be given a high priority for purely universal conjectures. If it fails to find a counter-example, the search for one might continue, but with a very low priority. Moreover, these two complementary tasks could be allowed to interact. For instance, if some cases of a proof attempt are successful, then the search for a counter-example should be focused on the outstanding cases.

8.6.3 A society of agents for inductive theorem proving

Many inductive proofs involve coming up with a well-founded ordering, which in turn, involves speculation of the order and then proving the well-foundedness part. This is then used for the application of induction. Coming up with the well-founded relation is a key step in such scenarios, and not an obvious one, at that. So, it will be extremely useful if this process can be semi-automated. The distributed paradigm can be used gainfully here in the following manner:

1. Independent well-founded relation suggesting agent (this can be user-input to start with or picked from a library of well-founded orderings); Communicates with (2),(3)

2. Independent well-foundedness proving agent; Communicates with (1); Even if it fails, can inform the rest of the proof.

3. Hypothesis forming agent; Communicates with (4),(1)
4. Society of agents: {Base-case, step-case}
5. The interesting point to note here is that these can be independent and one can spawn many threads of such speculations each of which can communicate with the other threads.

8.6.4 Co-routining scope in Middle-out reasoning

The work reported in the thesis titled *Proof Planning for Logic Program Synthesis* [Kraan, 1994] demonstrates how the combined techniques of middle-out reasoning and proof planning [Bundy, 1998] can be exploited to automate logic program synthesis. *Middle-out reasoning* is a term used to refer to the technique of representing unspecified objects in the proof with meta-variables (as a least commitment mechanism) and instantiating them via unification in the course of planning. Kraan's project implemented middle-out reasoning in proof planning for program synthesis, as an extension of the proof planning system CLAM [Bundy et al., 1990], a sequential proof planning system and the extended system, *Periwinkle* was used to synthesise a variety of programs.

Middle-out-reasoning gives a lot of opportunities for co-routining, dynamism and distribution, as explained below. Hence, a task involving application of middle-out reasoning and proof planning can be tackled more efficiently when a distributed proof planning system is used.

The induction critics make a lot of use of meta-variables as a least commitment mechanism. For instance, consider the proof planning critic that does the job of lemma speculation. Referring to the *Rippling* technique Bundy et al. [2005a], analysis of a failed ripple can determine a lot of the structure of a missing wave-rule, but not all of it. Those bits that cannot be fully determined are represented by meta-variables. These meta-variables are subsequently instantiated by higher-order unification during subsequent rippling. Similar remarks apply to the generalisation critic.

When the lemma speculation (or generalisation) critic is invoked there are two subgoals

to be proved:

1. The original goal must now be proved with the aid of the lemma (generalised goal).

2. The lemma (generalised goal) must be solved.

Either of these sub-proofs could cause the meta-variables to be instantiated. Note that an instantiation that would suit the proof of one of the sub-goals might make the other sub-goal unprovable (e.g. false). It is necessary to find instantiations that are compatible with both sub-proofs. In a sequential planner, it is easiest to arrange for one sub-proof to be completed before the other is started. This can lead to a lot of wasted effort if the instantiations made during the first sub-proof are incompatible with those needed for the second sub-proof. Moreover, there might be a large branching factor associated with some of these instantiations, leading to the need to search each branch in turn,. It is more efficient to do the instantiations in the sub-proof with the lowest branching factor, since this keeps the search space small. More generally, we might want to co-routine between the two sub-proofs, instantiating meta-variables incrementally and picking the sub-proof that offers the lowest branching factor at each phase of the instantiation. For instance, $F(x)$ might be instantiated to $g(F'(x))$ by the first process, then to $g(h(x))$ by the second. This kind of co-routining can be organised as a multi-agent process with one agent for each of the sub-proofs. Note that the agents must communicate their meta-variable instantiations to each other, with failure reporting if the instantiations made by one agent cause another agent's sub-proof to fail.

Another opportunity is that often more than one critic may be applicable to a failed ripple and these options can be tried simultaneously, e.g. speculating a lemma or trying a different induciton rule.

8.6.5 The Dynamic Creation of Induction Rules Using Proof Planning

This section discusses the distribution opportunities in the work reported in the thesis tittled *The Dynamic Creation of Induction Rules Using Proof Planning* [Gow, 2004] and the work on program construction via deductive synthesis [Bundy et al., 2005b].

The former was implemented as the *Dynamis* system, which used *Lambda Clam*, a sequential proof planning system, as its proof planner. For the purpose of program construction via deductive synthesis, the induction rules to be used cannot be determined from the usual heuristics. So, middle-out induction, as developed in the *Dynamis* system is used. That is, a meta-variable is used to stand for the induction term. This meta-variable is instantiated during rippling, fertilization, etc. A well-founded induction rule is then constructed that is based on the instantiated induction term. Details can be found in the aforementioned citations.

This process effectively determines the step case of the induction. It is then necessary to construct the remaining cases to cover the data-type; we will require base cases and sometimes additional step cases. The induction rule must be shown to be well-founded. Sub-syntheses may be required to construct sub-routines needed for the original program. All these sub-goals may share meta-variables. To summarise, some of the distribution possibilities are:

1. The process is initiated by the development of a step case with meta-variables arising from the induction term being instantiated. In *Dynamis*, processing was sequential, with this step case being completed before the other tasks were tackled. But we could, alternatively, start some of the other subgoals, in parallel, as soon as the meta-variables are partially instantiated. Indeed, if there are choices in this step case, we can explore them concurrently. Some subset of them might be combined to form the final induction rule. There may be several alternative ways of combining these step cases to provide alternative induction rules.

2. The rule must be proved well-founded. This involves finding a measure under which the induction variable and the various induction terms are well-ordered. This process can start as soon as the meta-variables standing for the induction terms begin to be instantiated. This process can interact with process 1 above, since the failure to show an induction term strictly well-ordered w.r.t the induction variable might cause a step case proof attempt to be rejected.

3. Missing cases must be detected and proved. Again, these detection and proof processes can start as soon as the induction term meta-variables are partially instantiated. In *Dynamis*, there was a lack of symmetry between the first step case and the remainder, but this need not be the case in a concurrent implementation.

4. Sub-synthesis of co-routines may be necessary. This is triggered by any properties that are required to be proved of residual meta-variables left after fertilization. These properties become specifications to be satisfied. These sub-syntheses can be initiated as soon as fertilization occurs and will contribute to the completion of the base and step cases.

All four of these processes involve shared meta-variables. Co-routining between them provides the advantages discussed in §8.6.4. But it introduces the problem of exchanging information about the instantiations of these meta- variables and the success and failure of these instantiations.

Considering the scenario of inductive rule synthesis: in general, the steps involved need to be applied in a compositional fashion and the distributed composition operator described in §7.6.1 can be applied gainfully to address the same. One example of such an application is as follows: Let tac_1 be a tactic to identify induction rules, tac_2 the induction strategy, y_1, the first candidate produced by tac_1 is a structural induction which is inadequate for the conjecture and y_2 the second candidate produced by tac_1 is a more complex induction that succeeds. We are interested in the composition of tac_1 and tac_2. The distributed composition operator can help to lead to a successful proof whereas the sequential composition operator cannot, assuming tac_1 does not terminate.

A 1 Parallel programming terminology

Threads, Pthreads, Java threads The term *thread* is used to describe an independent flow of control within a process and is essentially an operating system resource. Programming involving multiple threads is referred to as *multithreaded* programming.There are two main ways in which threads can be managed:

- Use them as an operating system resource for multithreaded programming, using standardised libraries like Pthreads (POSIX threads) which provides a suite of C library functions for thread management and synchronisation.
- Use them via higher level language objects as found in Java which offers the *thread* class, which provides thread management and synchronisation methods as part of the class. However, with high level languages, the multithreaded programming model has to be adapted to the programming paradigm of the language. For instance, in the case of Java, access and management of threads has to be via an *object*.

Compiler directives: OpenMP The OpenMP API provides a set of compiler directives and library routines to express shared memory parallelism by providing bindings for different languages via low level APIs. However, it requires a high level of expertise to program these. Typically, the programmer is required to annotate their sequential program with compiler directives, flagging the parts of the program that must be executed concurrently and specifying synchronisation points explicitly.

Sockets, MPI, PVM Sockets and their associated functions are mechanisms to establish channels of communication between two computers thus paving the way for engineering distributed computing applications. Although programming with sockets directly with the aid of C-based interfaces is possible, it is error-prone and requires understanding of the low-level characteristics of the network. Furthermore, it does not provide any mechanism for features like process management, fault tolerance, task migration, which have become crucial for modern parallel applications. Other options to handle socket programming are through high-level programming languages like Java or by using APIs like MPI or PVM, which are briefly described below:

PVM Parallel Virtual machine (PVM) consists of a runtime environment and related APIs that support different concurrent programming paradigms including the message passing paradigm with the distinguishing feature of support for a heterogeneous network of machines. The APIs are language specific and are in the form of primitives that have to be embedded in the program. The supported languages are C, C++ and Fortran. This is not being actively developed anymore.

MPI based APIs Message passing interface (MPI) is the de facto standard for APIs providing inter process communication. Programs written in sequential languages are augmented with the API directives to enable sending and receiving messages. In general, APIs are language-specific.

RPC Remote procedure call (RPC) is a technique by which a program can cause the execution of a non-local function, i.e. a function residing in another computer/processor/address space; The target program need not be specifically altered to enable this though as long as the language in question allows for RPC. RPC is generally very expensive in terms of the communication traffic as the calls are typically transmitted over a network.

HTTP Hyper Text Transfer Protocol is an application layer protocol for distributed systems. It is a request-response standard, typical of the *client-server* model of computation (a *server* provides a service which is used by one or more *clients* with the clients and server computing over a network or possibly residing in the same machine)

Data parallelism It is characterised by the parallel execution of the same operation on different data or different parts of a large data set. Implemented on varied architectures, the focus of data parallelism as a technique, is to be able to utilise a huge array of processing units by introducing appropriate parallelisation into the algorithms. The inter process communication is meant to be minimal. The technique is targeted at extremely fine grained parallelism and is thus suited for tightly coupled architectures as opposed to architectures that have slower communication machinery like distributed systems. Functional programming languages like Haskell are increasingly being used to implement data parallelism as are other APIs like Map-Reduce. These are discussed later in the chapter.

Work stealing Technique used in concurrent programming where when a process becomes idle, it tries to take over part of the work of another busy process

Serialisation/Marshalling Serialisation/Marshalling also referred to as *marshalling*, refers to the process of converting a data structure into a format such that it can be stored in a memory and/or can be transmitted over a network to be reassembled into the original data structure in a similar or different environment. It is a very useful feature in the context of message passing based distributed systems.

Open programming the development of programs that support dynamic exchange of higher-order values with other processes

A 2 Alice features

This is an enumeration of some of the language features of Alice ML that are relevant to the work discussed in this thesis. For more details please see the webpage for the Alice ML manual

Futures Concurrent programming in Alice ML is uniformly based on the model of futures. A concurrent thread can be initiated by means of the spawn expression, e.g.,: *spawn 45*68* initiates the computation in a new thread and returns a future, a place-holder for the result of the concurrent computation. Once the result becomes available, the future will be globally replaced by the result. Threads

are said to be functional, in the sense that they have a result. Futures impose an implicit form of *dataflow synchronisation* (see §5.6)

Data-flow synchronization Futures can be passed around as values. Once an operation actually requests the value the future stands for, the corresponding thread will block until the future has been determined. This is known as data-flow synchronisation and is a powerful mechanism for high-level concurrent programming.

Channel A channel is a simple imperative message queue that allows *asynchronous* communication between processes. An arbitrary number of messages may be sent to a channel using *put*. The *get* operation takes the oldest message out of the channel and blocks if none is available.

Component Alice ML introduces the notion of *component* as the unit of compilation as well as deployment. A component contains a module expression that, when evaluated, potentially imports modules from other components.

Package A package is a value encapsulating an arbitrary (higher-order) module and its signature. Packages enrich the static type system of ML with a dimension of dynamic typing: unpacking a package performs a dynamic type check. This basic mechanism is used to make all kinds of dynamic operations safe, particularly exchange of higher-order data structures between different processes or export to a file system (*pickling*). A *save* operation writes a package to a given file.

Pickles A pickle is a self-contained platform independent representation of the saved *package*

Inter process communication: Pickling To allow for export and import of data between processes, Alice ML supports pickling, also known as Serialisation/Marshalling, for export of language data structures. Using pickling, arbitrary data can be pickled, including code and entire modules. Pickles are platform-independent and are hence suitable for exchange across heterogenous networks, especially the Internet. By pickling first-class functions, Alice ML processes can exchange behaviour.

Distribution: Tickets Components and pickles already provide a primitive form of distributed programming, since they may be imported or loaded from arbitrary locations across a local network or even the Internet. But Alice ML also provides high-level means for processes at different sites to communicate directly using a feature called *tickets*. Tickets are in the form of ASCII strings and act as a global means to access any language entity

The first mechanism that allows sites to establish peer-to-peer connections is *offer* and *take*. A process can create a package and make it available to other processes. Offering a package opens a communication port and returns an URI for that port. The URI is called a *ticket*. Other processes can then obtain the available package using the ticket. In general, *take* establishes a connection to the communication port denoted by the *ticket*, and retrieves the offered package.

Distribution: Proxies Tickets are intended merely as a means to establish an initial

connection between processes. All subsequent communication should be dealt with by the functions in the offered package. Alice ML provides a very simple feature to enable this idiom: *proxies*. A proxy is basically an RPC(remote-procedure-call) stub, a mobile reference to a stationary function that can be used in place of the function it references.

Thread execution There is no explicit mechanism for implementing prioritisation of thread execution. A thread will execute unless dependency is programmed explicitly.

Virtual machine Alice ML uses a virtual machine constructed on top of the SEAM infrastructure (Simple Extensible Abstract Machine), a portable infrastructure for building virtual machines which implements generic services like memory management, thread management, pickling etc. SEAM implements threads purely in software, using its own scheduling mechanism. It does not yet enable employment of system threads. Consequently, an Alice ML program cannot yet take advantage of multi-processor machines and multi-core processors. SEAM and the Alice VM have been implemented in C++, while the rest of the system is almost entirely bootstrapped in Alice ML.

Components Components are the unit of compilation as well as the unit of deployment in Alice ML. A program consists of a potentially open set of components that are created separately and loaded dynamically. Each Alice ML source file defines, and is compiled into, a component: the contained sequence of SML declarations is interpreted as a structure body, forming the export module. The respective export signature is inferred by the compiler. A component can access other components through a prologue of *import* declarations that specify the name of the module to be imported and the location of the module. Loading of imported components is performed *lazily*, and every component is loaded at most once. Loading implies evaluation of the respective component.

A 3 Alice ML code for hierarchical threads

Listing 1: Code for implementing a variant of the Alice ML threads to accommodate termination of child threads

```
(*This module is as an abstraction of threads, designed to
  handle the child-thread termination correctly.
  Type: (thread, list of threads):(baseThread, its child threads)
  Termination of the baseThread triggers termination of
  all child threads held in the list of threads*)

signature H_THR = (* hierarchical threads *)
sig
 type hThrType
 val mySpawnThread :  hThrType * (unit -> 'a) -> hThrType * 'a
 val myTerminate :  hThrType -> unit
 val baseThread :  hThrType
end

structure h_thr  :> H_THR  = struct
 type  hThrType = Thread.thread * Thread.thread list ref
 fun myTerminate(th,thList)=Thread.terminate th handle Thread.Terminated=>()

 fun wrap (f, children) () =   f ()
  handle (Thread.Terminate) =>
  (List.app
     (fn ht =>  if (Thread.state ht <> Thread.TERMINATED)
       then  (Thread.terminate ht) else  ())
  (!children);   raise Thread.Terminate)
 fun mySpawnThread ((self, children), f) =   let
   val grandchildren = ref []
   val (child, x) = Thread.spawnThread (wrap (f, grandchildren))
  in
   ( children := child :: !children; ((child, grandchildren),x) )
  end
 val baseThread =   let
   val children = ref ([] : Thread.thread list)
   val (t,_) = Thread.spawnThread (wrap (fn () => (* lazy  *) (), children ))
  in
   (t,children)
  end
end
```

A 4 Alice ML code for the DPLL solver

Listing 2: Code for sequential, iterative implementation of the DPLL algorithm, with non-chronological backjumping and learning

```
(* ==================================== *)
(* The  Davis-Putnam-Loveland-Logemann procedures.          *)
(*                                                                  *)
(* Copyright (c) 2003-2007, John Harrison
(* ========================================== *)
(* The Davis-Putnam and Davis-Putnam-Loveland-Logemann procedures.  *)
(*                                                                  *)
(* Copyright (c) 2003-2007, John Harrison.*)

(* All rights reserved.
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

* Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.

* The name of John Harrison may not be used to endorse or promote
products derived from this software without specific prior written
permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
                                                                  *)
(* ========================================== *)


import "lib";
import "intro";
import "formulas";
import "prop";
import "propexamples";
import "defcnf";

open Lib;
open Intro;
open Formulas;
open Prop;
open DefCNF;

fun one_literal_rule clauses = let
  val u = hd (valOf (List.find (fn cl => length cl = 1) clauses));
  val u' = negate u ;
  val clauses1 = List.filter (fn cl => not (mem u cl)) clauses
in
  smap (fn cl => subtract cl [u'] ord_forms)  clauses1   ord_fl
end
handle Option => clauses;
```

```
fun affirmative_negative_rule clauses  = let
  val (neg',pos) = partition negative (unions clauses ord_forms)
  val neg =  smap negate neg' ord_forms
  val pos_only = subtract pos neg ord_forms and neg_only =
  subtract neg pos ord_forms
  val pure = union pos_only (smap negate neg_only ord_forms)
  ord_forms
in
  if (pure = []) then raise (Failure "affirmative_negative_rule")
    else
  (List.filter
  (fn cl => ((intersect cl pure ord_forms) = []) ) clauses)
end;


fun posneg_count cls l =   let
  val m = List.length(List.filter (mem l) cls)
  val n = List.length(List.filter (mem (negate l)) cls)
  in
  m + n  ;
end;

(* ------------------------------------------------------------------------- *)
(*Iterative implementation with explicit trail instead of recursion*)
(* ------------------------------------------------------------------------- *)

datatype trailmix = Guessed | Deduced;;

fun unassigned cls trail = let
  fun litabs p =   case p of Not q => q | _ => p
  fun smap_temp f ord s = smap f s ord
in
  subtract (  unions (smap_temp (smap_temp litabs ord_forms)
                           ord_fl cls)  ord_forms  )
    (smap_temp (litabs o fst) ord_forms trail) ord_forms
end;;


fun unit_subpropagate (cls, partialFn ,trail) = let
  val cls' = List.map (List.filter
  ((not) o (defined partialFn) o   negate)) cls ;

  val uu = fn [c] =>  if not(defined partialFn c) then
                      [c] else raise (Failure "")
               | _ => raise (Failure "") ;
  val newunits = unions(mapfilter uu cls') ord_forms
in
  if newunits = [] then (cls',partialFn,trail) else
  let
    val trail' = itlist (fn p =>
                  fn t => (p,Deduced)::t) newunits trail
    val partialFn' = itlist (fn u => (u |-> ()))
                        newunits partialFn
  in
    unit_subpropagate (cls',  partialFn',trail')
   end
end

fun unit_propagate (cls,trail) = let
  val partialFn  =  itlist (fn (x,_) => (x |-> ()))
                        trail undefined
  val (cls',partialFn',trail') =
              unit_subpropagate (cls,partialFn,trail)
in
   (cls',trail')
end;;

fun backtrack trail =
  case trail of
    (p,Deduced)::tt => backtrack tt
```

```
  | _ => trail;;

fun dpli cls trail = let
  val (cls',trail') = unit_propagate (cls,trail)
in
  if mem [] cls' then
    case (backtrack trail) of
      (p,Guessed)::tt =>
                    dpli cls ((negate p,Deduced)::tt)
      | _ => false
  else
      case (unassigned cls trail') of
        [] => true
      | ps => let
                val p = maximize (posneg_count cls') ps
                in  dpli cls ((p,Guessed)::trail') end
end

fun dplisat fm = dpli (defcnfs fm) [];;

fun dplitaut fm = not(dplisat(Not fm));;

(* ------------------------------------------------------------------------- *)
(* With simple non-chronological backjumping and learning                    *)
(* ------------------------------------------------------------------------- *)

fun backjump cls p trail =
  case (backtrack trail) of
    (q,Guessed)::tt =>
        let val (cls',trail') =
        unit_propagate (cls,(p,Guessed)::tt) in
        if mem [] cls' then backjump cls p tt else trail end
  | _ => trail;;

fun dplb cls trail = let
   val (cls',trail') = unit_propagate (cls,trail) in
  if mem [] cls' then
    case (backtrack trail) of
      (p,Guessed)::tt => let
        val trail' = backjump cls p tt
        val declits = List.filter (fn (_,d) => d = Guessed) trail'
        val conflict = insert (negate p)
        (smap (negate o fst) declits ord_forms)  ord_forms
        in
         dplb (conflict::cls) ((negate p,Deduced)::trail')
        end
    | _ => false
  else
    case (unassigned cls trail') of
      [] => true
    | ps => let val  p = maximize (posneg_count cls') ps in
            dplb cls ((p,Guessed)::trail') end
end;

fun dplbsat fm = dplb (defcnfs fm) [];;

fun dplbtaut fm = not(dplbsat(Not fm));;
```

A 5 Alice ML code for the Stalmarck solver

Listing 3: Code for sequential, iterative implementation of the Stalmarck tautology checking algorithm, with modifications to make it to work as a clause-learning tool

```
(* ========================================================== *)
(* Simple implementation of Stalmarck's algorithm.            *)
(*                                                            *)
(* NB! This algorithm is patented for commercial use          *)
(*  (not that a toy version like this would actually be useful in  *)
(*  practice). See US patent 5 276 897, Swedish patent 467 076 and *)
(*  European patent 0403 454 for example.                     *)
(*                                                            *)
(* Copyright (c) 2003, John Harrison.                         *)
(* All rights reserved.
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

* Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.

* The name of John Harrison may not be used to endorse or promote
products derived from this software without specific prior written
permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
                                                              *)
(* ========================================== *)

(* to time tests  *)

import structure BasicTimer from "basicTimer"
import signature MAP from "x-alice:/lib/data/MAP-sig"

import "lib";
import "intro";
import "formulas";
import "prop";
import "defcnf";

structure Stalmarck =
struct

open Lib
open Intro
open Formulas
open Prop
open DefCNF

(* ---------------------------------------------------------- *)
(* Triplet transformation, using functions defined earlier.   *)
(* ---------------------------------------------------------- *)
```

```
fun triplicate fm =
  let val fm' = nenf fm
      val n = 1 + (overatoms (max_varindex "p_" o pname) fm' 0)
      val (p,defs,_) = maincnf (fm',undefined,n)
  in
      (p,map (snd o snd) (funset defs))
  end

(*  ------------------------------------------------------------------  *)
(* Automatically generate triggering rules to save writing them out *)
(*  ------------------------------------------------------------------  *)
fun atom lit = if negative lit then negate lit else lit;;

fun align (p,q) =
  if ord_forms (atom p) (atom q) then align(q,p) else
  if negative p then (negate p,negate q) else (p,q);;

fun equate2 (p,q) eqv = equate (negate p,negate q) (equate (p,q) eqv);;

fun irredundant rel eqs =
    case eqs of
  [] => []
    | (p,q)::oth =>
  if canonize rel p = canonize rel q
    then irredundant rel oth
  else insert (p,q) (irredundant (equate2 (p,q) rel) oth) ord_pair

fun consequences ((p,q) as peq) fm eqs =
  let
     fun follows(r,s) = tautology(Imp(And(Iff(p,q),fm),Iff(r,s)))
  in
      irredundant (equate2 peq unequal) (List.filter follows eqs)
  end

fun triggers fm =
  let val poslits = insert True (map (fn p => Atom p) (atoms fm)) ord_forms
      val lits = union poslits (map negate poslits) ord_forms
      val pairs = allpairs (fn p => fn q => (p,q)) lits lits
      val npairs = List.filter (fn (p,q) => atom p <> atom q) pairs
      val eqs = setify(map align npairs) ord_pair
      val raw = map (fn p => (p,consequences p fm eqs)) eqs
  in
      List.filter (fn (p,c) => c <> []) raw
  end


(* to show computed triggers  *)
val showTriggs =
    let fun indent () = print "␣␣␣␣"
        fun showPair (f,g) = (print "("; pr f; print "␣,␣"; pr g; print ")")
        fun pr_pair_list l =
      let fun ppl [] = ()
      | ppl [h] = showPair h
      | ppl (h::t) = (showPair h; print "␣,␣"; ppl t)
      in
                print "\n␣␣␣["; ppl l; print "]\n"
      end
        fun showLine (a,b) =
      (showPair a; pr_pair_list b)
    in
        List.app showLine
    end

(* ------------------------------------------------------------------ *)
(* An example.                                                        *)
(* ------------------------------------------------------------------ *)

(*
START_INTERACTIVE;;
triggers <<p <=> (q /\ r)>>;;
```

```
END_INTERACTIVE;;
*)

(* ------------------------------------------------------------------ *)
(* Precompute and instantiate triggers for standard triplets.         *)
(* ------------------------------------------------------------------ *)


fun trigger z =
    let
        val p = Atom (P "p")
  and q = Atom (P "q")
  and r = Atom (P "r")

  val f1 = Iff ( p, And (q,r) )
  and f2 = Iff ( p, Or (q,r) )
  and f3 = Iff ( p, Imp (q,r) )
  and f4 = Iff ( p, Iff (q,r) )

  val [trig_and, trig_or, trig_imp, trig_iff] =
      map triggers [f1,f2,f3,f4]

  fun ddnegate fm = case fm of Not(Not p) => p | _ => fm

  fun inst_fn [x,y,z] =
      let
              val  subfn = fpf2 [P"p" |--> x, P"q" |--> y, P"r" |--> z]
      in
    ddnegate o propsubst subfn
      end
  fun inst2_fn i (p,q) = align(inst_fn i p,inst_fn i q)
  fun instn_fn i (a,c) = (inst2_fn i a,map (inst2_fn i) c )
  val inst_trigger = map o instn_fn
    in
  case z of
     (Iff(x,And(y,z))) => inst_trigger [x,y,z] trig_and
               | (Iff(x,Or(y,z))) => inst_trigger [x,y,z] trig_or
               | (Iff(x,Imp(y,z))) => inst_trigger [x,y,z] trig_imp
               | (Iff(x,Iff(y,z))) => inst_trigger [x,y,z] trig_iff
    end


(* ------------------------------------------------------------------ *)
(* Compute a function mapping each variable/true to relevant          *)
(* triggers.                                                          *)
(* ------------------------------------------------------------------ *)
fun relevance trigs =
let
  fun insert_relevant p trg f = (p |-> insert trg (tryapplyl f p) ord_trig) f;;
  fun insert_relevant2 (((p,q),_) as trg) f  =
    insert_relevant p trg (insert_relevant q trg f)
in
  itlist insert_relevant2 trigs undefined
end

(* ------------------------------------------------------------------ *)
(* Merging of equiv classes and relevancies.                          *)
(* ------------------------------------------------------------------ *)
fun equatecons ord ord2 (p0,q0) ((eqv,rfn: (''a * ''c list) list FormMap.map ) as erf) =
  let
      val p = canonize eqv p0 and q = canonize eqv q0
  in
      if p = q then ([],erf)
      else
    let
        val p' = canonize eqv (negate p0) and q' = canonize eqv (negate q0)
        val eqv' = equate2(p,q) eqv

        and sp_pos = tryapplyl rfn p and sp_neg = tryapplyl rfn p'
        and sq_pos = tryapplyl rfn q and sq_neg = tryapplyl rfn q'
```

```
        val rfn' =  ( (canonize eqv' p)  |-> (union sp_pos sq_pos ord) )
       (( (canonize eqv' p') |-> (union sp_neg sq_neg ord) ) rfn)

        val nw = union (intersect sp_pos sq_pos ord)
           (intersect sp_neg sq_neg ord) ord
    in
        ( itlist (fn (x,y) => fn z => union y z ord2) nw [],
         (eqv',rfn')
         )
    end
  end

(* ------------------------------------------------------------------------- *)
(* Zero-saturation given an equivalence/relevance and new assignments.       *)
(* ------------------------------------------------------------------------- *)
fun zero_saturate  erf assigs =
  case assigs of
    [] => erf
  | (p,q)::ts =>
      let val (news,erf') = equatecons ord_trig ord_pair (p,q) erf
      in
    zero_saturate  erf' (union ts news ord_pair)
      end

(* ------------------------------------------------------------------------- *)
(* Zero-saturate then check for contradictoriness.                           *)
(* ------------------------------------------------------------------------- *)
fun zero_saturate_and_check erf trigs =
  let
      val ((eqv',rfn') as erf') = zero_saturate erf trigs
      val vars = List.filter positive (equated eqv')
  in
      if (List.exists (fn x => canonize eqv' x = canonize eqv' (Not x)) vars)
    then snd(equatecons ord_trig ord_pair (True,Not True) erf')
      else erf'
  end

(* ------------------------------------------------------------------------- *)
(* Now we can quickly test for contradiction.                                *)
(* ------------------------------------------------------------------------- *)

fun truefalse pfn = canonize pfn (Not True) = canonize pfn True;;

(* ------------------------------------------------------------------------- *)
(* Iterated equivalencing over a set.                                        *)
(* ------------------------------------------------------------------------- *)

fun equateset ord1 ord2 s0 eqfn =
  case s0 of
    a::(b::s2 as s1) =>
      equateset ord1 ord2 s1 (snd(equatecons ord1 ord2 (a,b) eqfn))
  | _ => eqfn


(* ------------------------------------------------------------------------- *)
(* Intersection operation on equivalence classes and relevancies.            *)
(* ------------------------------------------------------------------------- *)

fun inter els ((eq1,_) as erf1) ((eq2,_) as erf2) rev1 rev2 erf =
    case els of
    [] => erf
  | x::xs =>
      let
    val b1 = canonize eq1 x and b2 = canonize eq2 x
    val s1 = apply rev1 b1  and s2 = apply rev2 b2

          fun ord x y = true
    val s = intersect s1 s2 ord_forms
      in
```

```
    inter (subtract xs s ord_forms) erf1 erf2 rev1 rev2
          (equateset  ord_trig ord_pair s erf)
      end


(* ------------------------------------------------------------------ *)
(* Reverse the equivalence mappings.                                  *)
(* ------------------------------------------------------------------ *)

fun reverseq ord domain eqv =
  let val al = map (fn x => (x,canonize eqv x)) domain
  in
      itlist (fn (y,x) => fn f => (x |-> insert y (tryapplyl f x) ord) f)
         al undefined
  end


(* ------------------------------------------------------------------ *)
(* Special intersection taking contradictoriness into account.        *)
(* ------------------------------------------------------------------ *)

fun stal_intersect ((eq1,_) as erf1) ((eq2,_) as erf2) erf =
  if truefalse eq1 then erf2
  else if truefalse eq2 then erf1 else
  let
      val dom1 = equated eq1 and dom2 = equated eq2
      val comdom = intersect dom1 dom2 ord_forms
      val rev1 = reverseq ord_forms dom1 eq1 and rev2 = reverseq ord_forms dom2 eq2
  in
      inter comdom erf1 erf2 rev1 rev2 erf
  end

(* ------------------------------------------------------------------ *)
(* General n-saturation for n >= 1                                    *)
(* ------------------------------------------------------------------ *)
    fun saturate n erf assigs allvars =
    let val ((eqv',_) as erf') = zero_saturate_and_check erf assigs
    in
        if n = 0 orelse truefalse eqv' then erf'
        else
      let val ((eqv'',_) as erf'') = splits n erf' allvars allvars
      in
          if eqFP (eqv'', eqv') then erf''
          else saturate n erf'' [] allvars
      end
    end
      and splits n ((eqv,_) as erf) allvars vars =
    case vars of
        [] => erf
      | p::ovars =>
              if canonize eqv p <> p then splits n erf allvars ovars else
              let
                  val erf0 = saturate (n - 1) erf [(p,Not True)] allvars
      and erf1 = saturate (n - 1) erf [(p,True)] allvars
      val ((eqv',_) as erf') = stal_intersect erf0 erf1 erf
        in
      if truefalse eqv' then erf'
      else splits n erf' allvars ovars
        end

(* ------------------------------------------------------------------ *)
(* Saturate up to a limit.                                            *)
(* ------------------------------------------------------------------ *)

val showPartition = ref false

fun saturate_upto vars n m trigs assigs =
  if n > m then
    failwith("Not-"^(Int.toString m)^"-easy")
  else
```

```
    (print_string("*** Starting "^(Int.toString n)^"-saturation");
     print_newline();
     let
      val (eqv,_) = saturate n (unequal,relevance trigs) assigs vars
      val _ = if !showPartition then print_pn eqv else ()
     in
      (truefalse eqv) orelse (saturate_upto vars (n + 1) m trigs assigs)
     end
    )

(* ------------------------------------------------------------------ *)
(* Overall function.                                                  *)
(* ------------------------------------------------------------------ *)
fun stalmarck fm =
  let
   fun include_trig (e,cqs) f =
   (e |---> union cqs (tryapplyl3 f e) ord_pair) f;;
   val fm' = psimplify(Not fm)
  in
      if fm' = False
    then true
      else if fm' = True
    then false
      else
    let val (p,triplets) = triplicate fm'
        val trigfn = itlist (itlist include_trig o trigger)
                    triplets undefined3
              val vars =  map (fn p => Atom p) (unions (map atoms triplets) ord_prop)
    in
        saturate_upto vars 0 2 (funset3 trigfn) [(p,True)]
    end
  end

(* ------------------------------------------------------------------ *)
(* Try the primality examples.                                        *)
(* ------------------------------------------------------------------ *)

(*
START_INTERACTIVE;;
do_list (time stalmarck)
  [prime 5;
   prime 13;
   prime 23;
   prime 43;
   prime 97];;
END_INTERACTIVE;;
*)

(* ------------------------------------------------------------------ *)
(* Artifical example of Urquhart formulas.                            *)
(* ------------------------------------------------------------------ *)
fun urquhart n = let
      fun uptoN 1 = [1]
  | uptoN n = if n < 1
      then failwith "negative number not allowed"
        else n :: (uptoN (n-1))
      val pvs = map (fn n => Atom(P("p_"^(Int.toString n)))) (uptoN n)
in
      end_itlist (fn p => fn q => Iff(p,q)) (pvs @ pvs)
end

val time = BasicTimer.time;
fun testUrquhart()=let fun f n=time (fn()=>stalmarck (urquhart n))
in  map f [1,2,4,8,16]     end
(*START_INTERACTIVE;;
map (time stalmarck ** urquhart) [1;2;4;8;16];;
END_INTERACTIVE;;*)
end
```

A 6 Alice ML code for the DPLL-Stalmarck solver

Listing 4: Code for the hybrid solver, DPLL-Stalmarck

```
(* ========================================== *)
(* Hybrid solver based on the Davis-Putnam-Loveland-Logemann(DPLL) *)
(* and Stalmarck algorithms.
    *)
(* The code for these sequential solvers has been borrowed from the SML versions of the
    same, found in the code repository accompanying a recent textbook on automated
    reasoning, entitled, \textit{Handbook of Practical Logic and Automated Reasoning} \
    citep{harrison-book}. The code can be found in the following web pages: \href{http://
    www.cl.cam.ac.uk/~jrh13/atp/OCaml/dp.ml}{SML code for sequential DPLL} and \href{http
    ://www.cl.cam.ac.uk/~jrh13/atp/OCaml/stal.ml}{SML code for sequential Stalmarck
    tautology checker} and is in turn protected by the
following copyright notice *)
                                                                                *)

(* Copyright (c) 2003-2007, John Harrison.*)
(* All rights reserved.
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

* Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.

* The name of John Harrison may not be used to endorse or promote
products derived from this software without specific prior written
permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
                                                                                *)
(* ========================================== *)

(* ------------------------------------------------------------------------- *)
(* The DP procedure.                                                         *)
(* ------------------------------------------------------------------------- *)

import signature CHANNEL from "x-alice:/lib/data/CHANNEL-sig"
import structure Channel from "x-alice:/lib/data/Channel"
import structure Remote from "x-alice:/lib/distribution/Remote"
import structure Inspector from "x-alice:/lib/tools/Inspector"

import "lib";
import "intro";
import "formulas";
import "prop";
import "propexamples";
import "defcnf";
import "stal-lib";

open Lib;
open Intro;
```

```
open Formulas;
open Prop;
open DefCNF;

fun one_literal_rule clauses = let
  val u = hd (valOf (List.find (fn cl => length cl = 1) clauses));
  val u' = negate u ;
  val clauses1 = List.filter (fn cl => not (mem u cl)) clauses
in
  smap (fn cl => subtract cl [u'] ord_forms)  clauses1   ord_fl
end
handle Option => clauses;

fun affirmative_negative_rule clauses  = let
  val (neg',pos) = partition negative (unions clauses ord_forms)
  val neg =  smap negate neg' ord_forms
  val pos_only = subtract pos neg ord_forms and neg_only = subtract neg pos ord_forms
  val pure = union pos_only (smap negate neg_only ord_forms) ord_forms
in
  if (pure = []) then raise (Failure "affirmative_negative_rule")
    else
  (List.filter  (fn cl => ((intersect cl pure ord_forms) = []) ) clauses)
end;

fun resolve_on p clauses = let
  val p' = negate p and (pos,notpos) = partition (mem p) clauses
  val (neg,other) = partition (mem p') notpos
  val pos' = smap ( List.filter (fn l => l <> p) )  pos ord_fl
  and neg' = smap (List.filter (fn l => l <> p') ) neg ord_fl
  fun temp_union a b = union a b ord_forms
  val res0 = allpairs temp_union pos' neg'
in
  union other (List.filter (noN trivial) res0) ord_fl
end;;

fun resolution_blowup cls l = let
  val m = List.length(List.filter (mem l) cls)
  and n = List.length(List.filter (mem (negate l)) cls)
 in
  m * n - m - n
end;;

fun resolution_rule clauses = let
  val pvs = List.filter positive (unions clauses ord_forms)
  val p = minimize (resolution_blowup clauses) pvs
in
  resolve_on p clauses
end;;

(* ------------------------------------------------------------------------- *)
(* Overall procedure.                                                        *)
(* ------------------------------------------------------------------------- *)

fun dp clauses =
  if clauses = [] then true else if mem [] clauses then false else
    dp (one_literal_rule clauses)
    handle Failure _ =>
         dp (affirmative_negative_rule clauses)
         handle Failure _ =>
            dp(resolution_rule clauses);;

(* ------------------------------------------------------------------------- *)
(* Davis-Putnam satisfiability tester and tautology checker.                 *)
(* ------------------------------------------------------------------------- *)

fun dpsat fm = dp(defcnfs fm);;

fun dptaut fm = not(dpsat(Not fm));;

(* ------------------------------------------------------------------------- *)
```

```
(* Examples.                                                                 *)
(* ------------------------------------------------------------------------ *)
(*
START_INTERACTIVE;;
tautology(prime 11);;

dptaut(prime 11);;
END_INTERACTIVE;;*)

(* ------------------------------------------------------------------------ *)
(* The same thing but with the DPLL procedure.                              *)
(* ------------------------------------------------------------------------ *)

fun posneg_count cls l =   let
  val m = List.length(List.filter (mem l) cls)
  val n = List.length(List.filter (mem (negate l)) cls)
  in
  m + n  ;
end;

fun  dpll clauses =
  if clauses = [] then true else if mem [] clauses then false else
   dpll(one_literal_rule clauses)
   handle Failure _ =>
     dpll(affirmative_negative_rule clauses)
    handle Failure     _ =>
  let
   val pvs = List.filter positive (unions clauses ord_forms)
   val p = maximize (posneg_count clauses) pvs
  in
   dpll (insert [p] clauses ord_fl) orelse dpll (insert [negate p] clauses ord_fl)
  end;;

fun dpllsat fm = dpll(defcnfs fm);;

fun dplltaut fm = not(dpllsat(Not fm));;

(* ------------------------------------------------------------------------ *)
(* Example.                                                                 *)
(* ------------------------------------------------------------------------ *)
(*
START_INTERACTIVE;;
dplltaut(prime 11);;
END_INTERACTIVE;;
*)
(* ------------------------------------------------------------------------ *)
(* Iterative implementation with explicit trail instead of recursion.       *)
(* ------------------------------------------------------------------------ *)

datatype trailmix = Guessed | Deduced;;

fun printTrail t  = let
val litsFromTrail = (List.map (fn (a,_) => a)) t
fun printLiteral l = case l of  (Not(p)) =>  (print "~" ; (pr p); print "\n") | p => (
    pr p;print "\n")
in
 (print "Beg\n"; List.app printLiteral litsFromTrail;print "Beg\n")
end;

fun unassigned cls trail = let
  fun litabs p =   case p of Not q => q | _ => p
  fun smap_temp f ord s = smap f s ord
in
  subtract (  unions (smap_temp (smap_temp litabs ord_forms)  ord_fl cls)  ord_forms  )
    (smap_temp (litabs o fst) ord_forms trail) ord_forms
end;;


fun unit_subpropagate (cls, partialFn ,trail) = let
  val cls' = List.map (List.filter ((not) o (defined partialFn) o negate)) cls ;
```

```
  val uu = fn [c] =>  if not(defined partialFn c) then [c] else raise (Failure "")
                    | _ => raise (Failure "") ;
  val newunits = unions(mapfilter uu cls') ord_forms
in
  if newunits = [] then (cls',partialFn,trail) else
  let
    val trail' = itlist (fn p => fn t => (p,Deduced)::t) newunits trail
    val partialFn' = itlist (fn u => (u |-> ()))    newunits partialFn
  in
    unit_subpropagate (cls',  partialFn',trail')
   end
end

fun unit_propagate (cls,trail) = let
  val partialFn  =  itlist (fn (x,_) => (x |-> ())) trail undefined
  val (cls',partialFn',trail') = unit_subpropagate (cls,partialFn,trail)
in
   (cls',trail')
end;;

fun backtrack trail =
  case trail of
    (p,Deduced)::tt => backtrack tt
  | _ => trail;;

fun dpli cls trail = let
(*++++++++++Search space tracking ++++++++++*)
val (dpll_visitedNodes : (prop formula) Channel.channel) = Channel.channel ();

fun dpll_visitedNodesPut1 elt = Channel.put(dpll_visitedNodes, elt)

fun dpll_visitedNodesGet1 () = let
val tempCh = Channel.clone dpll_visitedNodes
in Channel.toListNB(tempCh) end;
(*++++++++++Search space tracking ++++++++++*)

(*Timing*)
val dpllTime = Timer.startRealTimer();

      fun dpli_main cls trail = let
  val (cls',trail') = unit_propagate (cls,trail)
      in
  if mem [] cls' then
    case (backtrack trail) of
      (p,Guessed)::tt => ( dpll_visitedNodesPut1 (negate p); (*search space tracking*)
                           dpli_main cls ((negate p,Deduced)::tt) )
    | _ => false
  else

      case (unassigned cls trail') of
        [] => (printTrail trail'; true)
       | ps => let
                  val p = maximize (posneg_count cls') ps;
                  do dpll_visitedNodesPut1 p;
                in
                        dpli_main cls ((p,Guessed)::trail')
                    end
      end;
in
  (dpli_main cls trail, dpll_visitedNodesGet1(), (Time.toString(Timer.checkRealTimer
       dpllTime) ))
end

fun dplisat fm = dpli (defcnfs fm) [];;
fun dplitaut fm = let val (res, srchSpc,t) = (dplisat (Not fm)) in (not(res), srchSpc,t)
     end;;


(* ------------------------------------------------------------------------- *)
```

```
(* With simple non-chronological backjumping and learning.                  *)
(* ------------------------------------------------------------------------ *)

fun backjump cls p trail =
  case (backtrack trail) of
    (q,Guessed)::tt =>
        let val (cls',trail') = unit_propagate (cls,(p,Guessed)::tt) in
        if mem [] cls' then backjump cls p tt else trail end
  | _ => trail;;

fun dplb cls trail = let
(*++++++++++Search space tracking ++++++++++*)
val (dpll_visitedNodes : (prop formula) Channel.channel) = Channel.channel ();

fun dpll_visitedNodesPut1 elt = Channel.put(dpll_visitedNodes, elt)

fun dpll_visitedNodesGet1 () = let
val tempCh = Channel.clone dpll_visitedNodes
in Channel.toListNB(tempCh) end;
(*++++++++++Search space tracking ++++++++++*)

(*Timing*)
val dpllTime = Timer.startRealTimer();

fun dplb_main cls trail = let
   val (cls',trail') = unit_propagate (cls,trail) in
  if mem [] cls' then
    case (backtrack trail) of
      (p,Guessed)::tt => let
        val trail' = backjump cls p tt
        val declits = List.filter (fn (_,d) => d = Guessed) trail'
        val conflict = insert (negate p) (smap (negate o fst) declits ord_forms)
             ord_forms
        (*search space tracking*)
        do dpll_visitedNodesPut1 (negate p);
        in
         dplb_main (conflict::cls) ((negate p,Deduced)::trail')
        end
    | _ => false
  else
    case (unassigned cls trail') of
      [] => (printTrail;true)
    | ps => let
     val p = maximize (posneg_count cls') ps;
     do dpll_visitedNodesPut1 p;
        in
    dplb_main cls ((p,Guessed)::trail')
        end
end;

in
 (dplb_main cls trail, dpll_visitedNodesGet1(), (Time.toString(Timer.checkRealTimer
      dpllTime) ))
end

fun dplbsat fm = dplb (defcnfs fm) [];;

fun dplbtaut fm = let val (res, srchSpc,t) = (dplbsat (Not fm)) in (not(res), srchSpc,t)
     end;;

(* ------------------------------------------------------------------------ *)
(* Examples.                                                                *)
(* ------------------------------------------------------------------------ *)
(*
START_INTERACTIVE;;
dplitaut(prime 101);;
dplbtaut(prime 101);;
END_INTERACTIVE;;
*)
```

```
(*-------------------------DPLL Stalmarck-----------------------------------------*)

type inboxElt  = prop formula list

(*
rSaturation_withEqvReturn;
val it :
   string -> bool -> 'a -> (Formulas.prop Lib.formula list list -> unit) ->
      Formulas.prop Lib.formula -> 'b -> int -> bool -> bool list * Time.time
*)
fun tempHelperFun_remote varInd putTkt  helperTime fm   = let
fun putDednsAsEqFn () = ()
val p = Remote.take putTkt
structure st = unpack (p): (val dpllInboxPut : prop formula list  list  -> unit);
val putDednsAsClFn = st.dpllInboxPut
val r =
    if (helperTime <0) then
       let  val (t,f) = Thread.spawnThread (fn()=>rSaturation_withEqvReturn varInd true "
            temp" putDednsAsClFn fm putDednsAsEqFn 2 false)
       in f end

    else
      (    valOf(timeout ( (fn()=>rSaturation_withEqvReturn varInd true "temp"
           putDednsAsClFn fm putDednsAsEqFn 2 false), helperTime) )
            handle option => ([], Time.now())
       )
in
()
end

fun bootstrapHelper_remote varInd putTkt  helperFn fm helperTime = tempHelperFun_remote
     varInd putTkt   helperTime fm;

(*++++++++++++++++++++++++++++++++++*)
fun tempHelperFun varInd putDednsAsClFn helperTime fm   = let
fun putDednsAsEqFn () = ()
val (thrHandle,r) =
    if (helperTime <0) then
       let  val (t,f) = Thread.spawnThread (fn()=>rSaturation_withEqvReturn varInd true "
            temp" putDednsAsClFn fm putDednsAsEqFn 2 false)
       in (SOME t, f) end
    else
      (
        let
          val (t2,r2) = timeout_thr ( (fn()=>rSaturation_withEqvReturn varInd true "temp
               " putDednsAsClFn fm putDednsAsEqFn 2 false), helperTime)
        in
          ( t2,  (valOf(r2) handle option => ([], Time.now()) ) )
        end
       )
in
 thrHandle
end

fun bootstrapHelper varInd putDednsAsClFn helperFn fm helperTime = tempHelperFun  varInd
     putDednsAsClFn  helperTime fm;

fun wrapUpHelper thrHandle = if isNone thrHandle then () else Thread.terminate (valOf
     thrHandle) handle _ => ();
(*++++++++++++++++++++++++++++++++++*)

fun dpli_stal helperFun helperTime fm cls trail = let

(* ++++++++++++ STEP 1 +++++++++++++++ *)
(* make the local dpllInbox channel; Any external agent can post a
message to this; As long as they know the appropriate ticket:
which here is is XdpllTkt X;
For example , the external agents can be the Stalmarck agents *)

val (dpllInbox :  inboxElt  Channel.channel) = Channel.channel ();
```

```
(* Function to insert elements to dpllInbox *)
fun dpllInboxPut1 eltList = List.app (fn y => Channel.put(dpllInbox ,y)) eltList ;

(* Allow for REMOTE INVOCATION of the above function *)
val dpllInboxPutPack =
pack (val dpllInboxPut = Remote.proxy dpllInboxPut1 ) :
(val dpllInboxPut : inboxElt list -> unit)
val dpllPutTkt = Remote.offer dpllInboxPutPack;
do print dpllPutTkt;


(* Function to get elements from dpllInbox
Note that this will return a list of clauses*)
fun dpllInboxGet1 () = let
val tempCh = Channel.clone dpllInbox
in Channel.toListNB(tempCh) end;


(* Allow for REMOTE INVOCATION of the above function *)
val dpllInboxGetPack =
pack (val dpllInboxGet = Remote.proxy dpllInboxGet1 ) :
(val dpllInboxGet : unit -> inboxElt list )
val dpllGetTkt = Remote.offer dpllInboxGetPack;
do print dpllGetTkt;

(* ++++++++++ STEP 2 +++++++++++++++ *)
val thrHandle =  bootstrapHelper "p_" dpllInboxPut1 helperFun fm helperTime

(*++++++++++Search space tracking ++++++++++*)
val (dpll_visitedNodes : (prop formula) Channel.channel) = Channel.channel ();

fun dpll_visitedNodesPut1 elt = Channel.put(dpll_visitedNodes, elt)

fun dpll_visitedNodesGet1 () = let
val tempCh = Channel.clone dpll_visitedNodes
in Channel.toListNB(tempCh) end;
(*++++++++++Search space tracking ++++++++++*)

(*Timing*)
val dpllTime = Timer.startRealTimer();

(*DPLL*)
fun dpli_stal_main cls trail = let
val clsListFromInbox = dpllInboxGet1 ();

val relClsFromInbox = dropDuplicatesFromClsList(List.filter (isClRelevant (
    varsInListOfClauses cls) ) clsListFromInbox)
val cls = List.@(cls,relClsFromInbox) (*Add relevant Inbox clauses to problem*)

val (cls',trail') = unit_propagate (cls,trail)
in
  if mem [] cls' then
    case (backtrack trail) of
      (p,Guessed)::tt => ( dpll_visitedNodesPut1 (negate p); (*search space tracking*)
                           dpli_stal_main cls ((negate p,Deduced)::tt))
    | _ => false
  else
      case (unassigned cls trail') of
        [] => (printTrail trail'; true)
      | ps => let
                val p = maximize (posneg_count cls') ps
                (*search space tracking*)
                do dpll_visitedNodesPut1 p;
              in
                dpli_stal_main cls ((p,Guessed)::trail')
              end
end

in
```

```
 ( let
   val res =   (dpli_stal_main cls trail, dpll_visitedNodesGet1(), (Time.toString(Timer.
       checkRealTimer dpllTime)) )
   in
    (wrapUpHelper thrHandle;res)
   end
  )
end

fun dpli_stalsat helperFn helperTime fm =  dpli_stal helperFn helperTime fm (defcnfs fm)
     [];;
fun dpli_staltaut helperFn helperTime fm = let val (res, srchSpc,t) = (dpli_stalsat
     helperFn helperTime (Not fm)) in (not(res), srchSpc, t) end;;
fun dplb_stal helperFun helperTime fm cls trail = let

(* ++++++++++++ STEP 1 +++++++++++++++++ *)
(* make the local dpllInbox channel; Any external agent can post a
message to this; As long as they know the appropriate ticket:
which here is is XdpllTkt X;
For example , the external agents can be the Stalmarck agents *)

val (dpllInbox :  inboxElt  Channel.channel) = Channel.channel ();

(* Function to insert elements to dpllInbox *)
fun dpllInboxPut1 eltList =
List.app (fn y => Channel.put(dpllInbox ,y)) eltList ;

(* Allow for REMOTE INVOCATION of the above function *)

val dpllInboxPutPack =
pack (val dpllInboxPut = Remote.proxy dpllInboxPut1 ) :
(val dpllInboxPut : inboxElt list  -> unit)
val dpllPutTkt = Remote.offer dpllInboxPutPack;
do print dpllPutTkt;

(* Function to get elements from dpllInbox
Note that this will return a list of clauses*)
un dpllInboxGet1 () = let
val tempCh = Channel.clone dpllInbox
in Channel.toListNB(tempCh) end;

(* Allow for REMOTE INVOCATION of the above function *)
val dpllInboxGetPack =
pack (val dpllInboxGet = Remote.proxy dpllInboxGet1 ) :
(val dpllInboxGet : unit -> inboxElt list )
val dpllGetTkt = Remote.offer dpllInboxGetPack;
do print dpllGetTkt;

(* ++++++++++++ STEP 2 +++++++++++++++++ *)
val thrHandle =   bootstrapHelper "p_" dpllInboxPut1 helperFun fm helperTime

(*++++++++++Search space tracking ++++++++++*)
val (dpll_visitedNodes : (prop formula) Channel.channel) = Channel.channel ();

fun dpll_visitedNodesPut1 elt = Channel.put(dpll_visitedNodes, elt)

fun dpll_visitedNodesGet1 () = let
val tempCh = Channel.clone dpll_visitedNodes
in Channel.toListNB(tempCh) end;
(*++++++++++Search space tracking ++++++++++*)

(*Timing*)
val dpllTime = Timer.startRealTimer();

fun dplb_stal_main cls trail = let
   val clsListFromInbox = dpllInboxGet1 ();

   val relClsFromInbox = dropDuplicatesFromClsList(List.filter (isClRelevant (
       varsInListOfClauses cls) ) clsListFromInbox)
   val cls = List.@(cls,relClsFromInbox) (*Add relevant Inbox clauses to problem*)
```

```
    val (cls',trail') = unit_propagate (cls,trail) in
  if mem [] cls' then
    case (backtrack trail) of
      (p,Guessed)::tt => let
        val trail' = backjump cls p tt
        val declits = List.filter (fn (_,d) => d = Guessed) trail'
        val conflict = insert (negate p) (smap (negate o fst) declits ord_forms)
             ord_forms

        (*search space tracking*)
        do dpll_visitedNodesPut1 (negate p);
        in
         dplb_stal_main (conflict::cls) ((negate p,Deduced)::trail')
        end
    | _ => false
  else
    case (unassigned cls trail') of
      [] => (printTrail;true)
    | ps => let
     val p = maximize (posneg_count cls') ps;
     do dpll_visitedNodesPut1 p;
        in
    dplb_stal_main cls ((p,Guessed)::trail')
        end
end;

in
 ( let
   val res = (dplb_stal_main cls trail, dpll_visitedNodesGet1(), (Time.toString(Timer.
       checkRealTimer dpllTime) ))
   in
    (wrapUpHelper thrHandle;res)
   end
  )
end

fun dplb_stalsat helperFn helperTime fm = dplb_stal helperFn helperTime fm (defcnfs fm)
     [];;
fun dplb_staltaut helperFn helperTime fm = let val (res, srchSpc, t) = (dplb_stalsat
     helperFn helperTime  (Not fm)) in (not(res), srchSpc, t) end;;
```

A 7 Code fragment for the abstraction for DPLL working with a helper

Listing 5: Programming abstraction of DPLL solver with helper

```
fun makeInboxAndGetAccessHandles () = let
val (dpllInbox : inboxEltType Channel.channel) = Channel.channel ();
(* Function to insert a list of clauses to dpllInbox *)
fun dpllInboxPut1 eltList = List.app (fn y => Channel.put(dpllInbox ,y)) eltList;
(* Function to get elements from dpllInbox*)
fun dpllInboxGet1()=let val tempCh=Channel.clone dpllInbox in Channel.toListNB tempCh end
(* Allow for remote invocation of the above functions *)
val dpllInboxPutPack = pack (val dpllInboxPut = Remote.proxy dpllInboxPut1)
                            : (val dpllInboxPut : inboxEltType list -> unit)
val dpllPutTkt = Remote.offer dpllInboxPutPack;
val dpllInboxGetPack = pack (val dpllInboxGet = Remote.proxy dpllInboxGet1)
                             : (val dpllInboxGet : unit -> inboxEltType list )
val dpllGetTkt = Remote.offer dpllInboxGetPack;
in
 (dpllInbox, dpllInboxPut1, dpllInboxGet1, dpllPutTkt, dpllGetTkt)
end
fun doDPLLwithHelper dpllSolver inboxEltType bootstrapHelper wrapUpHelper helperFun
     helperTime fm = let
 (* make the local dpllInbox channel; Any external agent (e.g., stalmarck agent)
    can post to this, as long as they know the appropriate ticket*)
 val (dpllInbox, dpllInboxPut1, dpllInboxGet1, dpllPutTkt, dpllGetTkt)=
      makeInboxAndGetAccessHandles()
 val thrHandle = bootstrapHelper dpllPutTkt dpllGetTkt helperFun fm helperTime
 (* ++++++++++ DPLL +++++++++++++++ *)
 val cls = defcnfs fm; (*Converting to CNF*) val trail = [] (*Initial value*)
 val res = dpllSolver dpllInboxGet1 cls trail;do wrapUpHelper thrHandle;
in  res end
```

A 8 Code trace for the working of collaborative unification tactic

In §7.9.1, we described *depthCrossTalk*, a depth-first-search-based automatic tactic. *depthCrossTalk* is identical to *depth*, HAL's depth-first-search-based automatic tactic, except that it uses *crossTalk* instead of HAL's unify tactic. A recap of *depth* and *depthCrossTalk* is as follows:

```
fun depthFirst pred tac x = if pred x then all else (tac -- depthFirst pred tac)
val depth = depthFirst final (safeSteps 1 || unify 1 || quant 1);
(*Use the crossTalk tactic instead of unify in the above line; to unify across
all pending sub-goalsPass [] as the first argument to crossTalk*)
val depthCrossTalk =   depthFirst final (safeSteps 1 || crossTalk [] || quant 1);
```

We now provide an example illustrating the scenario of performing unification on a proof-state, which has sub-goals with shared meta-variable(s). *depthCrossTalk* solves the problem whereas *depth* does not. The detailed workings of the following example are given in Appendix §A 8.

Example 0.1 Collaborative unification

GIVEN: For constants, p, q, r,

1. $\forall x\, Q(x) \wedge R(x) \rightarrow P(x)$
2. $\forall x\, S(x) \rightarrow Q(x)$
3. $\forall x\, Q1(x) \wedge R1(x) \rightarrow P1(x)$
4. $R(p) \wedge R(q) \wedge R(r)$
5. $S(p) \wedge S(q)$
6. $Q1(q) \wedge R1(q)$

GOAL: $\exists x.(P(x) \wedge P1(x))$

Proof state: Applying propositional and quantification rules on the above problem, we get the following proof state with 6 sub-goals and meta-variables: ?_a,?_b,?_c:

Listing 6: Example illustrating the utility of crossTalk, the collaborative unification tactic. 'connective'-L/R to the left and right sequent calculus rules for 'connective'; variables preceded with the '?' symbol denote meta-variables.

```
MAIN GOAL: (ALL x. S(x) --> Q(x)) & ((ALL x. R1(x) & Q1(x) --> P1(x)) &
       (S(p) &
        (S(q) &
         (R(p) &
          (R(q) &
           (R(r) &
            (Q(p) &
             (Q(q) &
              (Q1(q) &
               (R1(q) &
                (ALL x. R(x) & Q(x) --> P(x))))))))))) -->
                (EX x. P(x) & P1(x))
SUB-GOALS:
 1. P(?_b), R1(q), Q1(q), Q(q), Q(p), R(r), R(q),
    R(p), S(q), S(p),
    ALL x. R1(x) & Q1(x) --> P1(x),
    ALL x. S(x) --> Q(x),
    ALL x. R(x) & Q(x) --> P(x)
    |-  P(?_a), EX x. P(x) & P1(x)
 2. R1(q), Q1(q), Q(q), Q(p), R(r), R(q), R(p),
    S(q), S(p), ALL x. R1(x) & Q1(x) --> P1(x),
    ALL x. S(x) --> Q(x),
    ALL x. R(x) & Q(x) --> P(x)
    |-  R(?_b), P(?_a), EX x. P(x) & P1(x)
 3. R1(q), Q1(q), Q(q), Q(p), R(r), R(q), R(p),
    S(q), S(p), ALL x. R1(x) & Q1(x) --> P1(x),
    ALL x. S(x) --> Q(x),
    ALL x. R(x) & Q(x) --> P(x)
    |-  Q(?_b), P(?_a), EX x. P(x) & P1(x)
 4. P(?_c), R1(q), Q1(q), Q(q), Q(p), R(r), R(q),
    R(p), S(q), S(p),
    ALL x. R1(x) & Q1(x) --> P1(x),
    ALL x. S(x) --> Q(x),
    ALL x. R(x) & Q(x) --> P(x)
    |-  P1(?_a), EX x. P(x) & P1(x)
 5. R1(q), Q1(q), Q(q), Q(p), R(r), R(q), R(p),
    S(q), S(p), ALL x. R1(x) & Q1(x) --> P1(x),
    ALL x. S(x) --> Q(x),
    ALL x. R(x) & Q(x) --> P(x)
    |-  R(?_c), P1(?_a), EX x. P(x) & P1(x)
 6. R1(q), Q1(q), Q(q), Q(p), R(r), R(q), R(p),
    S(q), S(p), ALL x. R1(x) & Q1(x) --> P1(x),
    ALL x. S(x) --> Q(x),
    ALL x. R(x) & Q(x) --> P(x)
    |-  Q(?_c), P1(?_a), EX x. P(x) & P1(x)
```

Next step: apply unification: Find a suitable unifier for the list of meta-variables: ?_a , ?_b, ?_c, which satisfies all the 6 sub-goals. As can be worked out easily, the possible unifier(s) for each sub-goal (i.e. which make the left and right sides of the sequent identical) are as follows:

1. ?_a = ?_b
2. ?_b = (r,q,p) i.e. 3 candidates:(_b,r), (_b,q), (_b,p)
3. ?_b = (q,p) i.e. 2 candidates: (_b,q), (_b,p)
4. Unification cannot be applied successfully

5. ?_c = (r,q,p) i.e. 3 candidates: (_c,r), (_c,q), (_c,p)

6. ?_c = (q,p) i.e. 2 candidates: (_c,q) (_c,p)

HAL's *depth* tactic results in a non-terminating search:

When HAL's sequential depth-first search tactic, *depth* is applied, the *unify* tactic is used to tackle unification. As explained earlier, this tackles unification for each sub-goal. In our example here, the first unifier produced by sub-goal-2, (_b = r), results in a looping situation, resulting in a non-terminating proof search. In particular, here, the looping happens because new disjuncts are added to the right hand side of the sequent.

Given the lazy nature of the list of states returned by the *unify* tactic used by *depth*, (_b = r) is applied across all sub-goals and execution of the *depth* tactic is continued. This in turn, means application of the *quant* and *safe* tactics in succession, on the state produced after the application of (_b = r).

Even if just a re-ordering of variables may suffice to circumvent the problem faced in our contrived example, it is easy to see that the problem can be rearranged in a way that still poses the same problem. Furthermore, the effect of ordering illustrates a problem that can appear in many other forms.

***depthcrossTalk* solves the goal:**

Application of *depthCrossTalk*, the depth-first-approach-based automatic tactic which uses the collaborative unification tactic, *crossTalk* (see §7.9.1), successfully solves the goal. A summary of the workings of the proof attempt by *depthCrossTalk* is provided below. This illustrates the process of finding the consensus unifiers, using the *crossTalk* tactic.

The unifiers are printed as Key-val pairs. e.g., for sub-goal 3, the two unifiers are: [Key= _b,Val= p] and [Key= _b,Val= q]. Only the successful attempts at finding a consensus are included in the listing below. The names of the native inference rules being applied at each step are also included, should the reader wish to work through the example.

The *STEP* numbers included can be tracked with the same in the crossTalk code fragment given earlier (see Listing 7.9). Also, for sub-goal 4, unification cannot be applied. *crossTalk* deals with such a situation by ignoring the sub-goal for the purpose of finding the consensus unifiers. But, when the next-states are returned, the unifier gets applied to all the sub-goals, including sub-goal 4.

Listing 7: Execution-trace of crossTalk, for given example; Finding the consensus unifiers

```
****Applying crossTalk******
STEP-2.3 Consensus is of length..3;  (_a , _b , _c) = (q, q, q)
STEP-2.3 Consensus is of length..3;  (_a , _b , _c) = (q, q, p)
STEP-2.3 Consensus is of length..3; (_a , _b , _c) = (p, p, q)
STEP-2.3 Consensus is of length..3; (_a , _b , _c) = (p, p, p)
STEP-3 Num of consensus unifiers: 4 ****
```

Finding more consensus unifiers: As observed in the description of *crossTalk* earlier, the states are returned as a sequence, to adhere to the type definition of a tactic. Thus, in the rest of this trace, after the application of *crossTalk*, the state corresponding to the first unifier in the list of consensus unifiers is used to generate the corresponding next-proof-state. This proof state is used for the subsequent inference steps.

Referring to the trace given above, the first candidate in the sequence of next-proof-states is generated by applying the unifier (_a , _b , _c) = (q, q, q). This is applied to all the 6 sub-goals and execution of *depthCrossTalk* is continued on the resulting state.

Listing 8: Execution-trace of crossTalk, for given example; Finding more consensus unifiers

```
**** Applying safe ******
[basic, basic, basic, basic, &-L, |-R, ->-R, ~-L, ~-R, exL, allR, &-R,
|-L, ->-L, <->-L, <->-R, basic, &-L, |-R, ->-R, ~-L, ~-R, exL,
allR, &-R, |-L, ->-L, <->-L, <->-R ]

**** Applying crossTalk ******
*** !!! STEP-1 Num of sub goals..!!!3;; !!! Meta-variable list: [] !!!

**** Applying quant ****** [allL, exR]
**** Applying safe ******
[basic &-L, |-R, ->-R, ~-L, ~-R, exL, allR, , &-R basic &-L, |-R, ->-R,
 ~-L, ~-R exL, allR, &-R, |-L, ->-L, basic, &-L, |-R, ->-R, ~-L, ~-R,
exL, allR, &-R, |-L, ->-L, <->-L, <->-R, basic, &-L, |-R, ->-R, ~-L,
~-R, exL, allR, , &-R, |-L, ->-L, <->-L, <->-R]

**** Applying crossTalk ******
*** !!! STEP-1 Num of sub goals..!!!5; !!! Meta-variable list: _d , _e , !!!
STEP-2.3 Consensus is of length..2 ; (_d , _e ) = (q, q);
STEP-3 Num of consensus unif envs is 1****
```

Using the consensus unifiers : From the above, we get

$$(\_a , \_b , \_c, \_d, \_e) = (q, q, q, q, q)$$

Listing 9: Execution-trace of crossTalk, for given example; Using the consensus unifiers

```
**** Applying safe ******
[basic, basic, basic, &-L, |-R, ->-R, ~-L, ~-R, exL, allR, &-R, |-L,
->-L, basic, basic, &-L, |-R, ->-R, ~-L, ~-R, exL, allR, &-R, basic,
basic, basic, basic, basic, &-L, |-R, ->-R, ~-L, ~-R, exL, allR, &-R,
|-L, ->-L, <->-L, <->-R]

(ALL x. S(x) --> Q(x)) & ((ALL x. R1(x) & Q1(x) --> P1(x)) &
 (S(p) &
  (S(q) &
   (R(p) &
    (R(q) &
     (R(r) &
      (Q(p) &
       (Q(q) &
        (Q1(q) &
         (R1(q) &
          (ALL x.
             R(x) & Q(x) --> P(x))))))))))) --> EX x. P(x) & P1(x))

                                                    No subgoals left!
```

Thus, *depthCrossTalk*, via application of the *crossTalk* tactic to perform unification across the 6 sub-goals, with shared meta-variables has circumvented the looping situation caused by an incompatible unifier, which led the sequential *depth* tactic of HAL (which uses the sequential *unify* tactic to perform unification) to a non-terminating search.

A 9 Implementation of unification in HAL - code

Listing 10: Code fragment for implementation of unification in HAL

```
structure Unify = struct
exception Failed;
(*Naive unification of terms containing no bound variables*)
fun unifyLists env = let (*Chase variable assignments*)
 fun chase (Fol.Var a)=(chase(StringDict.lookup(env,a))
 handle StringDict.E _ => Fol.Var a) |chase t = t
 fun occurs a (Fol.Fun(_,ts)) = occsl a ts
 |occurs a (Fol.Param(_,bs)) = occsl a (map Fol.Var bs)
 |occurs a (Fol.Var b) = (a=b) orelse (occurs a (StringDict.lookup(env,b))
 handle StringDict.E _ => false) |occurs a _ = false
and   occsl a = List.exists (occurs a)
and   unify (Fol.Var a, t) =  if t = Fol.Var a then env  else
 if occurs a t then raise Failed else StringDict.update(env,a,t)
 |unify (t, Fol.Var a) = unify (Fol.Var a, t)
 |unify (Fol.Param(a,_), Fol.Param(b,_)) = if a=b then env else
 raise Failed  |unify (Fol.Fun(a,ts), Fol.Fun(b,us)) =
 if a=b then unifyl(ts,us) else raise Failed
 |unify _ = raise Failed
and unifyl([],[])=env|unifyl(t::ts, u::us)=
unifyLists(unify(chase t,chase u))(ts,us)|unifyl _ = raise Failed in unifyl
end

(*Unification of atomic formulae
 val atoms: Fol.form * Fol.form -> Fol.term StringDict.t*)
fun atoms (Fol.Pred(a,ts), Fol.Pred(b,us)) =  if a=b then unifyLists
StringDict.empty (ts,us) else raise Failed | atoms _ = raise Failed;

(*Instantiate a term by an environment
val instTerm: Fol.term StringDict.t -> Fol.term -> Fol.term*)
fun instTerm env (Fol.Fun(a,ts)) = Fol.Fun(a, map (instTerm env) ts)
 |instTerm env (Fol.Param(a,bs)) = Fol.Param(a, foldr Fol.termVars []
   (map (instTerm env o Fol.Var) bs))
 |instTerm env (Fol.Var a) = (instTerm env (StringDict.lookup(env,a))
   handle StringDict.E _ => Fol.Var a) |instTerm env t = t;

(*Inst formula: val instForm: Fol.term StringDict.t -> Fol.form -> Fol.form*)
 fun instForm env (Fol.Pred(a,ts))= Fol.Pred(a, map (instTerm env) ts)
 |instForm env (Fol.Conn(b,ps))= Fol.Conn(b, map (instForm env) ps)
 |instForm env (Fol.Quant(qnt,b,p))= Fol.Quant(qnt, b, instForm env p)
 |instForm env (Fol.Equal(t1,t2))=Fol.Equal(instTerm env t1,instTerm env t2);

(*val instGoal: Fol.term StringDict.t -> Fol.goal -> Fol.goal end;*)
 fun instGoal env (ps,qs) =(map (instForm env) ps, map (instForm env) qs);
end
```

References

Andersson, G., Bjesse, P., Cook, B., and Hanna, Z. (2002). A proof engine approach to solving combinational design automation problems. In Proceedings of the 39th annual Design Automation Conference, DAC '02, pages 725–730, New York, NY, USA. ACM.

Andrews, G. R. (2000). Foundations of Multithreaded, Parallel, and Distributed Programming. Addison-Wesley.

Armstrong, J. (1997). The development of Erlang. In ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming, pages 196–203, New York, NY, USA. ACM.

Armstrong, J. (2007). A history of Erlang. In HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages, pages 6–1–6–26, New York, NY, USA. ACM.

Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., and Yelick, K. A. (2006). The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

Avenhaus, J., Denzinger, J., Kuchlin, W., and Sinz, C. (2002). Teamwork-PaReDuX: Knowledge-based search with multipleparallel agents. In Proceedings of the International Conference onMassively Parallel Computing Systems (MPCS 2002), Ischia, Italy. National Technological University Press, Fort Collins, CO, USA.

Beame, P., Kautz, H., and Sabharwal, A. (2003). Understanding the power of clause learning. In In: Proceedings of the 18th International Joint Conference on Artificial Intelligence, pages 1194–1201.

Benzmüller, C. and Sorge, V. (2000). OANTS – an open approach at combining interactive and automated theorem proving. In Kerber, M. and Kohlhase, M., editors, Symbolic Computation and Automated Reasoning, pages 81–97. A.K.Peters.

Benzmüller, C., Sorge, V., Jamnik, M., and Kerber, M. (2008). Combined reasoning by automated cooperation. Journal of Applied Logic, 6(3):318–342.

Biere, A., Heule, M. J. H., van Maaren, H., and Walsh, T., editors (2009). Handbook of Satisfiability, volume 185 of Frontiers in Artificial Intelligence and Applications. IOS Press.

Blochinger, W., Westje, W., Küchlin, W., and Wedeniwski, S. (2005a). ZetaSAT – Boolean satisfiability solving on desktop grids. In Proc. of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2005), volume 2, pages 1079–1086, Cardiff, UK.

Blochinger, W., Westje, W., Küchlin, W., and Wedeniwski, S. (2005b). ZetaSAT – Boolean satisfiability solving on desktop grids. In Proc. of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2005), volume 2, pages 1079–1086, Cardiff, UK.

Böhm, M. and Speckenmeyer, E. (1996). A fast parallel sat-solver - efficient workload balancing. Ann. Math. Artif. Intell., 17(3-4):381–400.

Bonacina, M. P. (1992). Distributed automated deduction. PhD thesis, Department of Computer Science, State University of New York at Stony Brook.

Bonacina, M. P. (1999). Ten years of parallel theorem proving: a perspective (invited paper). In Gramlich, B., Kirchner, H., and Pfenning, F., editors, Notes of the Third Workshop on Strategies in Automated Deduction, Second Federated Logic Conference (FLoC99), pages 3–15.

Borälv, A. (1997). The industrial success of verification tools based on stalmarck's method. In CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification, pages 7–10, London, UK. Springer-Verlag.

Bundy, A. (1998). Proof planning. Technical Report 886, School of Informatics, University of Edinburgh.

Bundy, A., Basin, D., Hutter, D., and Ireland, A. (2005a). Rippling: Meta-level Guidance for Mathematical Reasoning, volume 56 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press.

Bundy, A., Gow, J., Fleuriot, J., and Dixon, L. (2005b). Constructing induction rules for deductive synthesis proofs. In Allen, S., Crossley, J., Lau, K., and Poernomo, I., editors, Proceedings of the ETAPS-05 Workshop on Constructive Logic for Automated Software Engineering (CLASE-05), Edinburgh, pages 4–18. LFCS University of Edinburgh.

Bundy, A., van Harmelen, F., Horn, C., and Smaill, A. (1990). The Oyster-Clam system. In Stickel, M. E., editor, 10th International Conference on Automated Deduction, pages 647–648. Springer-Verlag. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 507.

Butler, R. M. and Lusk, E. L. (1994). Monitors, messages, and clusters: the p4 parallel programming system. Parallel Comput., 20(4):547–564.

Chrabakh, W. and Wolski, R. (2003). Gridsat: A chaff-based distributed sat solver for the grid. In SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing, page 37, Washington, DC, USA. IEEE Computer Society.

Cole, M. (1991). Algorithmic skeletons: structured management of parallel computation. MIT Press, Cambridge, MA, USA.

Cole, M. (2004). Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parallel Comput., 30(3):389–406.

Constable, R. L., Allen, S. F., Bromley, H. M., et al. (1986). Implementing Mathematics with the Nuprl Proof Development System. Prentice Hall.

Cook, S. A. (1971). The complexity of theorem-proving procedures. In Proceedings of the Third IEEE Symposium on the Foundations of Computer Science, pages 151–158.

Cope, M., Gent, I., and Hammond, K. (2001). Parallel heuristic search in Haskell. In Trends in Functional Programming 2, pages 65–76. Intellect.

Davis, M., Logemann, G., and Loveland, D. (1962). A machine program for theorem proving. Communications of the ACM, 5(7):394–397.

Davis, M. and Putnam, H. (1960). A computing procedure for quantification theory. Journal of the ACM, 7(3):201–215.

Dean, J. and Ghemawat, S. (2004). MapReduce: simplified data processing on large clusters. In Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6, OSDI'04, pages 10–10, Berkeley, CA, USA. USENIX Association.

Denzinger, J., Denzinger, O., Kronenburg, M., and Schulz, S. (1996). Discount - a distributed and learning equational prover. Journal of Automated Reasoning, 18:189–198.

Denzinger, J. and Kronenburg, M. (1996). Planning for distributed theorem proving: The teamwork approach. In KI '96: Proceedings of the 20th Annual German Conference on Artificial Intelligence, pages 43–56, London, UK. Springer-Verlag.

Drechsler, R. and Becker, B. (1998). Binary Decision Diagrams: Theory and Implementation. Springer.

Dubois, O. and Dequen, G. (2001). A backbone-search heuristic for efficient solving of hard 3-sat formulae. In Proc. of the IJCAI01, pages 248–253.

Eén, N. and Sörensson, N. (2004). An extensible SAT solver. In Theory and Applications of Satisfiability Testing, pages 333–336.

Falcou, J. (2009). Parallel programming with skeletons. Computing in Science and Engineering, 11(3):58–63.

Feldman, Y., Dershowitz, N., and Hanna, Z. (2005). Parallel multithreaded satisfiability solver: Design and implementation. In Electronic Notes in Theoretical Computer Science (ENTCS), volume 128 of Proceedings of the 3rd International Workshop on Parallel and Distributed Methods in Verification (PDMC 2004), pages 75–90.

Fisher, M. (1997). An open approach to concurrent theorem-proving. In Geller, K. and Suttner, editors, Parallel Processing for Artificial Intelligence, pages 121–164. Elsevier/North Holland.

Fisher, M. (2004). Multi-agent programming based on distributed deduction. In Zhang, W. and Sorge, V., editors, Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems, volume 112 of Frontiers in Artificial Intelligence and Applications. IOS Press.

Fisher, M. and Ghidini, C. (2002). The ABC of Rational Agent programming. In First International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS), pages 849–856. ACM Press.

Forman, S. L. and Segre, A. M. (2002). Nagsat: A randomized, complete, parallel solver for 3-sat. sat2002. In In Proceedings of Theory and Applications of Satisfiability Testing, SAT02, pages 236–243.

Fuchs, D. and Denzinger, J. (1997). Knowledge-based cooperation between theorem provers by techs. Technical Report SEKI-Report SR-97-11, University of Kaiserslautern.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (2002). Design patterns: abstraction and reuse of object-oriented design. In Software pioneers:contributions to software engineering, pages 701–717, New York, NY, USA. Springer-Verlag New York, Inc.

Gelder, A. V. (1999). Autarky pruning in propositional model elimination reduces failure redundancy. J. Autom. Reasoning, 23(2):137–193.

Gent, I. P. and Walsh, T. (1993). Towards an understanding of hill-climbing procedures for sat. In Proceedings of the eleventh national conference on Artificial intelligence, AAAI'93, pages 28–33. AAAI Press.

Gent, I. P. and Walsh, T. (1994a). The SAT phase transition. In Proceedings of the Eleventh European Conference on Artificial Intelligence (ECAI'94), pages 105–109.

Gent, I. P. and Walsh, T. (1994b). The sat phase transition. In European Conference on Artificial Intelligence (ECAI), pages 105–109.

Gil, L., Flores, P., and Silveira, L. M. (2008). PMSat: a parallel version of MiniSAT. Journal on Satisfiability, Boolean Modeling and Computation, 6:71–98.

Giles, C. L., Bollacker, K. D., and Lawrence, S. (1998). Citeseer: an automatic citation indexing system. In DL '98: Proceedings of the third ACM conference on Digital libraries, pages 89–98, New York, NY, USA. ACM.

Gomes, C., Selman, B., Crato, N., and Kautz, H. (2000). Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. Journal of automated reasoning, 24:2000.

Gomes, C. P., Selman, B., and Kautz, H. (1998). Boosting combinatorial search through randomization. In Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence, pages 431–437, Menlo Park, CA, USA. American Association for Artificial Intelligence.

Gow, J. (2004). The Dynamic Creation of Induction Rules Using Proof Planning. PhD thesis, University of Edinburgh.

Groote, J., van Vlijmen, S., and Koorn, J. (1995). The safety guaranteeing system at station hoorn-kersenboogerd. In Computer Assurance, 1995. COMPASS '95. 'Systems Integrity, Software Safety and Process Security'. Proceedings of the Tenth Annual Conference on, pages 57 –68.

Haken, A. (1985). The intractability of resolution. Theoretical Computer Science, 39(0):297 – 308. ¡ce:title¿Third Conference on Foundations of Software Technology and Theoretical Computer Science¡/ce:title¿.

Halstead, Jr., R. H. (1985). Multilisp: a language for concurrent symbolic computation. ACM Trans. Program. Lang. Syst., 7(4):501–538.

Hamadi, Y., Jabbour, S., and Sais, L. (2009). Control-based clause sharing in parallel sat solving. In Proceedings of the 21st international jont conference on Artifical intelligence, pages 499–504, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Hamadi, Y. and Sais, L. (2009). ManySAT: a parallel SAT solver. Journal On Satisfiability, Boolean Modeling And Computation (JSAT), 6.

Harrison, J. (1996). Stalmarck's algorithm as a HOL derived rule. In LNCS 1125, pages 221–234. Springer-Verlag.

Harrison, J. (2009). Handbook of Practical Logic and Automated Reasoning. Cambridge University Press, New York, NY, USA, 1st edition.

Heule, M. and van Maaren, H. (2008). Parallel SAT solving using bit-level operations. JSAT, 4(2-4):99–116.

Heyman, T., Geist, D., Grumberg, O., and Schuster, A. (2002). A scalable parallel algorithm for reachability analysis of very large circuits. In Formal Methods in System Design, pages 317–338.

Hickey, J. (1999). Fault-tolerant distributed theorem proving. In Proceedings of the 16th International Conference on Automated Deduction: Automated Deduction, CADE-16, pages 227–231, London, UK, UK. Springer-Verlag.

Hooker, J. and Vinay, V. (1995). Branching rules for satisfiability. Journal of Automated Reasoning, 15(3):359–383.

Hoos, H. H. and Stützle, T. (2000). SATLIB: An Online Resource for Research on SAT.

Huth, M. and Ryan, M. (2004). Logic in Computer Science: Modelling and Reasoning about Systems. Cambridge University Press, New York, NY, USA.

Hyvärinen, A. E., Junttila, T., and Niemelä, I. (2008a). Incorporating learning in grid-based randomized sat solving. In Proceedings of the 13th international conference on Artificial Intelligence: Methodology, Systems, and Applications, AIMSA '08, pages 247–261, Berlin, Heidelberg. Springer-Verlag.

Hyvärinen, A. E., Junttila, T., and Niemelä, I. (2008b). Strategies for solving SAT in grids by randomized search. In Proceedings of the 9th AISC international conference, the 15th Calculemas symposium, and the 7th international MKM conference on Intelligent Computer Mathematics, pages 125–140, Berlin, Heidelberg. Springer-Verlag.

Intosh, D. J. M., Conry, S. E., and Meyer, R. A. (1991). Distributed automated reasoning: Issues in coordination, cooperation, and performance. IEEE Transactions on systems, man and cybernetics, 21(6).

Jones, S. P. and Singh, S. (2008). A tutorial on parallel and concurrent programming in Haskell. In Lecture Notes in Computer Science. Springer Verlag.

Karp, R. M. and Ramachandran, V. (1990). Parallel algorithms for shared-memory machines. MIT Press, Cambridge, MA, USA.

Kaufmann, M. and Moore, J. S. (2009). Some key research problems in automated theorem proving for hardware and software verification.

Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. (1983). Optimization by simulated annealing. Science, 220(4598):671–680.

Kornfeld, W. A. and Hewitt, C. E. (1981). The scientific community metaphor. IEEE Trans on Systems, Man, and Cybernetics, 11(1).

Kraan, I. (1994). Proof Planning for Logic Program Synthesis. PhD thesis, University of Edinburgh.

Kunz, W. and Pradhan, D. (1994). Recursive learning: a new implication technique for efficient solutions to CAD problems-test, verification, and optimization. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 13(9):1143–1158.

Leroy, X. (1996). Website for the Objective Caml (OCaml) programming language. http://caml.inria.fr/.

Leroy, X. (2003). Website for work on MPI bindings for Objective Caml (OCaml) programming language. http://forge.ocamlcore.org/projects/ocamlmpi/.

Marques-Silva, J. P. and Sakallah, K. A. (1996). GRASP - A New Search Algorithm for Satisfiability. In Proceedings of IEEE/ACM International Conference on Computer-Aided Design, pages 220–227.

Marques-Silva, J. P., Sakallah, K. A., Marques, J. P., Karem, S., and Sakallah, A. (1996). Conflict analysis in search algorithms for propositional satisfiability. In Proceedings of the IEEE International Conference on Tools with Artificial Intelligence.

Matthews, D. (2010). Website for the PolyML programming language. http://www.polyml.org/index.html.

Matthews, D. C. and Wenzel, M. (2010). Efficient parallel programming in Poly/ML and Isabelle/ML. In DAMP '10: Proceedings of the 5th ACM SIGPLAN workshop

on Declarative aspects of multicore programming, pages 53–62, New York, NY, USA. ACM.

McCune, W. (1994). Otter 3.0 reference manual and guide. Technical report, Argonne National Laboratory, Argonne, IL.

Melis, E. and Meier, A. (2000). Proof planning with multiple strategies. In Lloyd, J., Dahl, V., Furbach, U., Kerber, M., Lau, K.-K., Palamidessi, C., Pereira, L., Sagiv, Y., and Stuckey, P., editors, Computational Logic CL 2000, volume 1861 of Lecture Notes in Computer Science, pages 644–659. Springer Berlin Heidelberg.

Melis, E. and Siekmann, J. (1999). Knowledge-based proof planning. Artif. Intell., 115(1):65–105.

Milner, R., Tofte, M., and Macqueen, D. (1997). The Definition of Standard ML. MIT Press, Cambridge, MA, USA.

Monien, B. and Speckenmeyer, E. (1985). Solving satisfiability in less than 2n steps. Discrete Applied Mathematics, 10:287–295.

Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. (2001). Chaff: Engineering an Efficient SAT Solver. In Proceedings of the 38th Design Automation Conference (DAC'01).

Nipkow, T., Paulson, L. C., and Wenzel, M. (2002). Isabelle/HOL — A Proof Assistant for Higher-Order Logic, volume 2283 of LNCS. Springer.

Odersky, M. (2004). An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland.

Okushi, F. (1999). Parallel cooperative propositional theorem proving. Ann. Math. Artif. Intell., 26(1-4):59–85.

Paulson, L. C. (1989). The foundations of a generic theorem prover. Journal of Automated Reasoning, 5:363.

Paulson, L. C. (1996). ML for the Working Programmer. Cambridge University Press, 2nd edition edition.

Robinson, A. and Voronkov, A., editors (2001). Handbook of automated reasoning. Elsevier Science Publishers B. V., Amsterdam, The Netherlands.

Rossberg, A. (2007). Typed Open Programming - A higher-order, typed approach to dynamic modularity and distribution. PhD thesis, Saarland University, Germany.

Rossberg, A., Botlan, D. L., Tack, G., Brunklaus, T., and Smolka, G. (2006). Alice Through the Looking Glass, volume 5 of Trends in Functional Programming, pages 79–96. Intellect Books, Bristol, UK, ISBN 1-84150144-1.

Roy, P. V. and Haridi, S. (2004). Concepts, Techniques, and Models of Computer Programming. MIT Press, Cambridge, MA, USA.

Sabharwal, A., Beame, P., and Kautz, H. (2003). Using problem structure for efficient

clause learning. In In Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing, pages 242–256. Springer-Verlag.

Schubert, T., Lewis, M., and Becker, B. (2005). PaMira - a parallel SAT solver with knowledge sharing. In MTV '05: Proceedings of the Sixth International Workshop on Microprocessor Test and Verification, pages 29–36, Washington, DC, USA. IEEE Computer Society.

Segre, A. M., Forman, S., Resta, G., and Wildenberg, A. (2002). Nagging: A scalable fault-tolerant paradigm for distributed search. Artificial Intelligence, 140(1-2):71 – 106.

Selman, B., Kautz, H. A., and Cohen, B. (1996). Local search strategies for satisfiability testing. In Johnson, D. and Trick, M., editors, Second DIMACS implementation challenge : cliques, coloring and satisfiability, volume 26 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 521–532. American Mathematical Society.

Sheeran, M. and Stalmarck, G. (1998). A tutorial on Stålmarck's proof procedure for propositional logic. In Gopalakrishnan, G. and Windley, P., editors, Proceedings 2nd Intl. Conf. on Formal Methods in Computer-Aided Design, FMCAD'98, Palo Alto, CA, USA, 4–6 Nov 1998, volume 1522, pages 82–99. Springer-Verlag, Berlin.

Sheeran, M. and Stalmarck, G. (2000). A tutorial on Stålmarck's proof procedure for propositionallogic. Form. Methods Syst. Des., 16:23–58.

Singer, D. (2006). Parallel resolution of the satisfiability problem: A survey. In Talbi, E.-G., editor, Parallel Combinatorial Optimization, pages 123–147. JohnWiley & Sons, Inc.

Sinz, C., Blochinger, W., and Kuchlin, W. (2001). PaSAT - parallel SAT-checking with lemma exchange:implementation and applications. In Kautz, H. and Selman, B., editors, LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001), volume 9 of Electronic Notes in Discrete Mathematics, Boston, MA. Elsevier Science Publishers.

Sinz, C., Küchlin, W., Feichtinger, D., and Görtler, G. (2006). Checking consistency and completeness of on-line product manuals. J. Autom. Reason., 37(1-2):45–66.

Spears, W. M. (1993). Simulated annealing for hard satisfiability problems. In In, Workshop, pages 533–558. American Mathematical Society.

Speckenmeyer, E., Böhm, M., and Heusch, P. (1997). On the imbalance of distributions of solutions of CNF-formulas and its impact on satisfiability solvers. In Du, D., Gu, J., and Pardalos, P. M., editors, Satisfiability Problem: Theory and Applications, volume 35 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 669–676. American Mathematical Society.

Speckenmeyer, E., Monien, B., and Vornberger, O. (1988). Superlinear speedup for parallel backtracking. In Proceedings of the 1st International Conference on Supercomputing, pages 985–993, New York, NY, USA. Springer-Verlag New York, Inc.

Sripriya, G., Bundy, A., and Smaill, A. (2007). Concurrent/distributed theorem proving in Isabelle/IsaPlanner. In The Isabelle Workshop 2007, CADE 07.

Stallman, R. M. and Sussman, G. J. (1977). Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. Artificial Intelligence, 9(2):135 – 196.

Stalmarck, G. (1992). A system for determining propositional logic theorem by applying values and rules to triplets that are generated from a formula. Swedish Patent No. 467 076 (approved 1992), U.S. Patent No. 5 276 907 (1994), European Patent No. 0403 454 (1995.

Stalmarck, G. (1994). A proof theoretic concept of tautological hardness. unpublished manuscript. In Unpublished manuscript, Logikkonsult NP AB.

Stalmarck, G. and Saflund, M. (1990a). Modelling and verifying systems and software in propositional logic. Technical report, University of Cambridge Computer Laboratory.

Stalmarck, G. and Saflund, M. (1990b). Modelling and verifying systems and software in propositional logic. Technical report, University of Cambridge Computer Laboratory.

Steele, Jr., G. L. (2009). Organizing functional code for parallel execution or, foldl and foldr considered slightly harmful. SIGPLAN Not., 44:1–2.

Sturgill, D. and Segre, A. M. (1997). Nagging: A distributed, adversarial search-pruning technique applied to first-order inference. J. Autom. Reason., 19(3):347–376.

Sutter, H. (2005). The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobb's Journal, 30(3).

Syme, D., Granicz, A., and Cisternino, A. (2007). Expert F#. Apress.

Thiffault, C., Bacchus, F., and Walsh, T. (2004). Solving non-clausal formulas with DPLL search. In Proc. 10th International Conference on Principles and Practice of Constraint Programming (CP 2004), in: Lecture Notes in Comput. Sci, pages 663–678. Springer.

Tseitin, G. S. (1968). On the complexity of derivation in the propositional calculus. Zapiski nauchnykh seminarov LOMI, 8:234–259. English translation of this volume: Consultants Bureau, N.Y., 1970, pp. 115–125.

Urquhart, A. (1987). Hard examples for resolution. Journal of the Association for Computing Machinery, 34(1):209–219.

von Behren, R., Condit, J., and Brewer, E. (2003). Why events are a bad idea (for high-concurrency servers). In HOTOS'03: Proceedings of the 9th conference on Hot Topics in Operating Systems, pages 4–4, Berkeley, CA, USA. USENIX Association.

Waerden, B. L. v. d. (1971). How the proof of Baudet's conjecture was found. In

Mirsky, L., editor, Studies in Pure Mathematics (papers presented to R. Rado on the occassion of his 65th birthday, pages 252–260. Academic Press.

Walsh, W. E., Yokoo, M., and Hirayama, K. (2001). Market based on market-inspired approaches to propositional satisfiability. IJCAI 2001, 0(0).

Walsh, W. E., Yokoo, M., Hirayama, K., and Wellman (2003). On market-inspired approaches to propositional satisfiability. Artificial Intelligence, 144:125–126.

Weber, T. (2006). Integrating a SAT solver with an LCF-style theorem prover. Electron. Notes Theor. Comput. Sci., 144:67–78.

Wenzel, M. (2009). Parallel proof checking in isabelle/isar. In ACM SIGSAM Workshop on Programming Languages for Mechanized Mathematics Systems (PLMMS 2009). ACM Digital Library, 2009. Parallel Poly/ML and Isabelle 10 2009/9/28.

Woolridge, M. (2001). Introduction to Multiagent Systems. John Wiley & Sons, Inc., New York, NY, USA.

Yelick, K. A. (1992). A parallel completion procedure for term rewriting systems. In CADE-11: Proceedings of the 11th International Conference on Automated Deduction, pages 109–123, London, UK. Springer-Verlag.

Yogesh Mahajan, Zhaohui Fu, S. M. (2004). Zchaff2004: An Efficient SAT Solver, volume 3542, pages 360–375. Springer.

Zhang, H. (1997). Sato: An efficient propositional prover. In McCune, W., editor, Proceedings of the 14th Conference on Automated Deduction, number 1249 in LNAI, pages 272–275.

Zhang, H., Bonacina, M. P., and Hsiang, J. (1996). PSATO: a distributed propositional prover and its application to quasigroup problems. Journal of Symbolic Computation, 21(4):543–560.

Zhang, H. and Stickel, M. E. (1994). Implementing the Davis-Putnam algorithm by tries. Technical report, Artificial Intelligence Center, SRI International, Menlo.

Zimmer, J. and Dennis, L. A. (2002). Inductive theorem proving and computer algebra in the mathweb software bus. In Proceedings of the Joint International Conferences on Artificial Intelligence, Automated Reasoning, and Symbolic Computation, AISC '02/Calculemus '02, pages 319–331, London, UK, UK. Springer-Verlag.

Glossary

Cloud computing is focussed on the *virtualisation* of applications, thus allowing for software to be provided as services running on huge commodity clusters [2]..

Cluster A group of workstations that are interconnected by general purpose communication networks such as fast ethernet or other advanced forms of high-speed connections.The terms GRIDS and clusters are used interchangeably and are treated essentially as distributed systems in the context of the material discussed in this work.

Concurrent program is characterised by more than one instruction sequence executing at the same time.

Concurrent programming In this thesis, this is used to refer to programming that allows for asynchronous (concurrent) modes of execution, irrespective of the architecture of implementation or the computational models used..

Data parallelism is characterised by the parallel execution of the same operation on different data or different parts of a large data set. In this thesis, it is used to refer to forms of computation where the same operation(s) is being performed on multiple datum by different processes in parallel. Typically, the size of the data set is huge and this computational model suits scenarios where bulk parallel processing resources are available..

Dataflow variable declarative variables that cause the thread of execution to wait until they are bound. Use of these allows for the order of execution to become inconsequential..

Grid is a distributed network of often heterogeneous computing elements (CE)that communicate using the infrastructure of the Internet.The terms GRIDS and clusters are used interchangeably and are treated essentially as distributed systems in the context of the material discussed in this work.

Lag tolerance the rest of the computation can continue while the result is being computed..

[2]For more on cloud computing, the reader may want to read this url http://en.wikipedia.org/wiki/Cloud_computing

MIMD Multiple Instruction Multiple Data Stream, a category of classification of parallel architectures based on notions of instruction and data stream.

Open programming the development of programs that support dynamic exchange of higher-order values with other processes.

Partial termination A thread of execution is said to have partially terminated if it has not terminated completely yet. Further binding of inputs would cause it to execute further,up to the next partial termination, and will execute no further if no binding happens..

Partial value A dataflow variable that has not yet been bound.

Referential transparency It usually means that an expression always evaluates to the same result in any context. This is the case in pure functional programming languages, but need not be always the case in other functional programming languages. Side effects like (uncontrolled) imperative update break this desirable property..

Serialisation/Marshalling conversion of a data structure into a format such that it can be stored in memory and/or can be transmitted over a network, to be reassembled into the original data structure in a similar or different environment..

Stream is used in concurrent programming to refer to a list with an unbounded tail. The term *port* refers to an abstraction used to manage a stream..

Thread An independently executing instruction sequence is called a *thread*.

Work stealing Common concurrent programming technique: when a process becomes idle, it tries to take over part of the work of another busy process..