INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Practical Executable Specifications for Distributed Systems

Nuno Claudino Pereira Lopes

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática e de Computadores

Júri

Presidente:	Doutor Nuno Mamede
Orientador:	Doutor José Carlos Monteiro
Co-Orientador Externo:	Doutor Andrey Rybalchenko
Vogal:	Doutor David Matos

Outubro 2009

Let us change our traditional attitude to
the construction of programs: Instead of
imagining that our main task is to instruct
a computer what to do, let us concentrate
rather on explaining to human beings what
we want a computer to do.

Donald E. Knuth,
Literate Programming, CSLI, 1992

Resumo

A presente tese descreve o DAHL, um ambiente de programação para o desenvolvimento de sistemas distribuídos. O DAHL oferece uma linguagem de alto nível especializada para a implementação de sistemas distribuídos, bem como um ambiente de execução estável e de alto desempenho, que consegue executar programas escritos em DAHL num ambiente de produção de forma eficiente.

A linguagem DAHL é declarativa, baseada em regras lógicas e tem execução desencadeada por eventos, o que permite implementações concisas, embora legíveis, de aplicações distribuídas. A linguagem DAHL oferece as instruções típicas de controlo de fluxo, tipos de dados expressivos, bem como um conjunto de predicados de rede e segurança embutidos.

O estado da arte das linguagens para implementar sistemas distribuídos inclui o Mace, que é baseado em C++ e portanto não é indicado para verficação formal, e o P2, que é baseado em Datalog, uma linguagem orientada para perguntas a bases de dados, que não é suficientemente expressiva para implementar aplicações interessantes. Para além disso, a implementação actual do P2 não é suficientemente eficiente para aplicações que não sejam puramente limitadas pela rede.

O DAHL combina o desempenho e a expressividade do Mace com a concisão do P2 para criar um melhor ambiente de desenvolvimento de sistemas distribuídos.

Para demonstrar as potencialidades do DAHL, implementámos diversas aplicações. Esta tese apresenta duas dessas implementações: Chord, uma tabela de dispersão distribuída, que é um protocolo de rede, e o D'ARMC, um verificador de modelos de *software* distribuído, que é um uma aplicação limitada pelo *CPU*.

Abstract

This thesis presents DAHL, a programming environment for the development of distributed systems. DAHL provides a high-level language that is specialized for the implementation of distributed systems, and a stable and high-performance runtime that can efficiently execute DAHL programs in a production environment.

The DAHL language is declarative, event-triggered and rule based, which enables concise, albeit clear, implementations of distributed applications. The DAHL language features the usual control-flow statements, expressive data types, as well as a set of network and security related built-in predicates.

State-of-the-art languages to implement distributed systems include Mace, which is based on C++ and thus lacks aptitude for formal verification, and P2, which is based on Datalog, a query language that is not expressive enough to implement interesting applications. Moreover, the current P2 implementation is not efficient enough for non-purely network bound applications.

DAHL combines the performance and expressiveness of Mace with the succinctness of P2 to create a better and practical programming environment for the development of distributed applications.

To demonstrate the power of DAHL, we implemented several applications of different representative categories. This thesis presents two of them: Chord, a distributed hash table, which is a network driven protocol, and D'ARMC, a distributed software model checker that is CPU-bound.

Palavras Chave
Keywords

Palavras Chave

Sistemas Distribuídos
Linguagens de Programação
Programação em Lógica
Verificação de Modelos

Keywords

Distributed Systems
Programming Languages
Logic Programming
Model Checking

Contents

List of Figures

List of Tables

List of Abbreviations

API	Application Programming Interface
BFS	Breadth-First Search
BFT	Byzantine Fault Tolerant
CEGAR	Counter-Example Guided Abstraction Refinement
CLP	Constraint Logic Programming
DAG	Directed Acyclic Graph
DFS	Depth-First Search
DHT	Distributed Hash Table
DSL	Domain Specific Language
FSM	Finite State Machine
GC	Garbage Collector
IDL	Interface Description Language
LoC	Lines of Code
P2P	Peer-to-Peer
RPC	Remote Procedure Call
RTT	Round-Trip Time
TCP	Transmission Control Protocol
URI	Universal Resource Identifier

Chapter 1

Introduction

Building a high-performance, fault-tolerant distributed application is a difficult and error-prone task (see, e.g., [46, 66]). The programmer has a lot of implementation details to consider, such as memory management, socket programming, message serialization, and operating systems subtleties, which are unrelated to the protocol specification itself. Moreover, a distributed application has to deal with ever-changing network conditions, node and link failures, asynchrony, and possible malicious behavior from other nodes in the system.

Chandra et al. [11] recently published their experience in building a real-world high-performance database using the Paxos protocol [31]. They note that, even with the authors' experience in building complex distributed systems and that the Paxos protocol was proposed and proven correct more than 15 years ago, implementing such a high-performance and robust database turned out to be a non-trivial multi-year effort. They also note that the main reason behind the complexity was the lack of appropriate tools that can help practitioners correctly implement these protocols.

Software tools can make the development of distributed systems an easier, less error-prone, and more productive activity, while improving the quality of the produced code. High-level programming languages supported by analysis and verification tools hold vast potential towards achieving these goals (e.g., [28,29,34,40,65,70]). Unfortunately, the existing approaches do not provide a comprehensive solution to the challenge because they sacrifice some of the necessary attributes, like language expressiveness, succinctness of implementation, efficiency, or aptitude to correctness checking.

In this thesis, I present DAHL, a programming system specialized for the development of distributed applications. DAHL is particularly targeted for the development of complex distributed applications and protocols, such as fault tolerant protocols, or distributed applications with complex decision logic (such as dynamic P2P overlays), although DAHL can be used to implement any general-purpose networked application. DAHL pursues three design objectives: make distributed systems easy to write, easy to understand and analyze, as well as easy to execute efficiently. To achieve these goals, DAHL offers a high-level programming language providing constructs for writing distributed applications, and a runtime environment for efficient program execution.

The DAHL language is based on Prolog [59], a declarative logic language. Declarative languages often result in simpler, clearer and more compact programs [32]. DAHL extends Prolog with specific predicates and other language constructs to enable efficient construction of networked programs. Moreover, DAHL extends Prolog with network-driven queries, opposed to the traditional user-driven queries. The programming model of DAHL is event-driven, i.e., a program can send and receive events trough event handlers. We believe this is a good model for the development of distributed programs, as it is similar to how protocols are usually specified (diagrams of messages that are exchanged between nodes), and because it holds potential for code reuse between programs, which is usually not possible or not efficient

with other programming models (e.g., finite state machines, where the code is tightly coupled with the set of states and transitions of a particular system).

By layering on top of an existing language, we can reuse its existing tools and runtime. In particular, the DAHL runtime is based on SICStus Prolog [62], which is an industry standard compiler. We use its compiler as well as its Prolog runtime environment (the blackboard, the facts database, the garbage collector, etc...). The networking back-end is based on libevent [38], which natively delivers application events when a network message arrives or when a timer expires. By being event-based and by reusing industrial strength tools, DAHL can achieve high throughput, scalability, and reliability, as demonstrated by the tests and benchmarks that we run.

In order to show that faithful implementations of complex distributed applications can be quickly built with DAHL, and that DAHL is expressive enough for a wide range of applications, we have implemented several distributed applications. In this thesis I present two of those: Chord [60], a peer-to-peer (P2P) distributed hash table (DHT) that is a network-driven protocol, and D'ARMC, a distributed software model checker that is CPU-bound. These applications were developed in a few days and their code size is at least an order of magnitude smaller than other implementations in more traditional languages, such as C. Moreover, these applications perform well when compared with the reference implementations. In case of Chord, the latency of lookups and the consistency levels are similar with the reference implementation, while D'ARMC achieves good speedups and efficiency levels.

D'ARMC (which stands for Distributed Abstraction Refinement Model Checking) is a distributed software model checker that was designed and implemented in DAHL, and it is a prominent approach to software verification that is based on predicate abstraction and its counterexample-guided refinement. The new algorithm implements a robust and deterministic model checker in a distributed environment, despite all the inherent non-determinism. Moreover, this distributed algorithm also achieves good speedups. In fact, to the best of my knowledge, D'ARMC is the world's first scalable distributed software model checker.

The main technical contributions of this thesis are: the specification of a language (DAHL) that is specialized for the development of distributed systems, a high-performance implementation of the runtime environment for the DAHL language, and an extensive evaluation of the DAHL language and runtime, including the implementation of well known protocols (such as Chord), as well as a newly proposed protocol for distributed software model checking (D'ARMC).

This thesis is organized as follows. The related work is presented in Chapter 2. In Chapter 3, the DAHL language is described. In Chapter 4, an overview of the implementation of the DAHL runtime is given, as well as an evaluation of its raw performance. Chapters 5 and 6 describe two applications that were implemented in DAHL and show their results. Finally, Chapter 7 presents some limitations and future work for DAHL, and Chapter 8 concludes.

Chapter 2

Related Work

Over the last few years, several specialized languages for the implementation of distributed systems have been proposed. Often called domain specific languages (DSL), the list includes, for example and in no special order, Linda [7, 8], nesC [20], MACEDON [51] and Mace [28], Acute [55, 56], and P2 [12, 33–37].

These languages are very different both in syntax and semantics. So, we identified several key aspects to better compare them, which are presented in this section. We also give a short summary of each language, as well as a description of how well they score in each aspect.

2.1 Evaluation Criteria

We have identified seven attributes that we believe are the most important to compare DAHL with other similar specialized languages. The attributes are the following: expressiveness, succinctness, bug finding, property proving, efficiency, and concurrency and programming models.

We say a language is expressive if it is possible to implement any distributed program in it without resorting to "hacks", like implementing extensions in another language. A language is succinct if programs written in such a language are considerably smaller than reference programs written in a more traditional language (e.g., C or C++). On the static analysis side, we propose two attributes: bug finding, and property proving. A language has these attributes if the accompanying programming system provides tools to perform those tasks. We say a language is efficient if its main implementation is at most one order of magnitude slower than a low-level language (e.g., C) in a set of benchmarks. Concrete results are given in Section 4.4. Finally we list the programming and concurrency models. In particular we are interested to note if a language forces any particular programming model, like finite state machines (FSM), and if there is support for concurrency.

We believe the attributes just described are the most important to compare the state-of-the-art languages. In particular, succinctness and expressiveness help us determining if a language is good enough to easily and quickly implement distributed systems. The bug finding and property proving attributes inform us if the language can be used for the implementation and verification of trustworthy systems. Finally, we are also interested in the performance, to evaluate whether the language can be used in production systems or not.

Table 2.1 summarizes the six languages that were mentioned previously: Linda, nesC, MACEDON, Mace, Acute, and P2 (pure[1]). These are further described in the following sections.

[1]Note that one can use C++ extensions within P2, which improves expressiveness at the expense of losing aptitude for program analysis.

Attribute	Linda	nesC	MACEDON	Mace	Acute	P2 (pure)	DAHL
Base Language	Prolog	C	C++	C++	OCaml	Datalog	Prolog
Expressiveness	✓	✓	✓	✓	✓	×	✓
Succinctness	✓	×	×	×	×	✓	✓
Bug finding	×	✓	×	✓	×	✓	✓
Property proving	×	×	×	×	×	✓	✓
Efficiency	×	✓	✓	✓	×	×	✓
Prog. Model	any	events	FSM	FSM	any	events	events
Concurrency	no	pseudo	lock based	between layers	no	no	no

Table 2.1: Feature comparison between Linda, nesC, MACEDON, Mace, Acute, P2, and DAHL.

2.2 Linda

Linda [7,8] augments Prolog with a set of predicates that extends the local view of the Prolog's database to a distributed setting, i.e., each node can add arbitrary Prolog terms to another node's Linda database. Linda does not impose any programming model, although it does not deliver events to the application when a new term is added to the database. It is the responsibility of the application to check whether there are new terms or not.

The two main predicates that Linda provides are `in/1` and `out/1`, which, respectively, receive and send an arbitrary term. The `in/1` predicate unifies its argument with an appropriate term from the Linda database (e.g., `in(event(lookup(Key, Client)))`), and blocks until some term matches. Newer versions of Linda include more predicates, such as a non-blocking `in` [62].

Linda borrows the expressiveness and succinctness of the language from Prolog. However, it does not provide any static analysis or verification tool. Moreover, the performance and stability of the current implementation – which often crashes – are not good, making the system unusable for production.

2.3 nesC

nesC [20] is a specialized language based on C (and heavily inspired by C++'s classes) that is designed for building distributed systems. It is particularly targeted to networked embedded systems, such as sensor networks (e.g., "motes" and "smart dust" [67]). nesC's compiler includes a simple optimizer (dead-code elimination and function inlining only) and a compile-time data-race detector.

2.3.1 The Language

The nesC language retains most of the features of C. Notable omissions are the lack of dynamic memory allocation and the lack of function pointers. This means that all resources are allocated statically and that the full call graph is know at compile-time. This reduction in expressiveness has the purpose of simplifying static analysis for optimizations (which is a very sensitive topic for embedded systems) and for the race-condition checker. However, an alias analysis pass is still required (and included) in the checker.

2.3.2 Concurrency Model

nesC's programming paradigm is event-driven and includes a somewhat simple concurrency model. nesC applications are built using components, which have an interface specifying which services they provide and use (similar to C++ classes). Each component may implement event handlers, commands, and tasks. Event handlers are called asynchronously when an event occurs, while commands (the equivalent

of C++'s class methods) are called synchronously by other commands, event handlers, or tasks. Tasks are a deferred computation mechanism: they run asynchronously and return immediately when called.

Tasks run to completion and do not preempt each other, but events may preempt the execution of a task or another event, while also running until completion. This raises consistency problems with regard to concurrency (e.g., when accessing shared data). To simplify this problem, nesC introduces an `atomic{...}` block in the language, meaning that the statements included inside such a block are executed atomically. However, this still imposes an additional burden on the programmer in order to make the code reentrant. Inside an atomic block it is forbidden to call commands, so that the effects of an atomic block are confined to a single module.

In nesC and in its accompanying Operating System (TinyOS [23]) there are no blocking operations. All long-latency operations are split into several phases (split-phase operations in the authors' terminology). This imposes some extra work on the programmer side, as one has to perform some additional housekeeping. For example, the network data send operation is split into two phases: in the first phase, the program asks the system to send a message by calling the `send` command, and in the second phase, the system notifies the program of the result of the operation (either success or failure in this case) by delivering a `sendDone` event.

2.3.3 Race-Condition Checker

nesC includes a race-condition checker that is used to validate individual variable accesses. It conservatively checks if all possible accesses to shared data are safe, i.e., either they occur inside an atomic block or all accesses to a given variable occur in synchronous code. However, this checker cannot detect a read-modify-write access pattern through a temporary variable, where the read and write operations occur in different atomic blocks.

To work around some limitations in the checker, nesC offers a `norace` qualifier that can be used in the variables' declarations in order to selectively disable the race condition checker.

2.4 MACEDON

MACEDON (Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks [51]) is a framework that provides a DSL for specifying distributed protocols, but focuses on overlay networks (distributed hash tables and application-layer multicast). It consists of a specification language, a source-to-source compiler, and a toolset for experimental evaluation of distributed protocols.

2.4.1 The Language

The specification language is a superset of C++. The event handlers (actions in MACEDON terminology) are coded in plain C++, but the code is augmented with some directives to describe, for example, the state transitions and the type of actions (i.e., read-only or not). Additionally, protocols can be composed. For example, one can specify that Scribe [9] uses Pastry [52] as its underlying distributed hash table protocol (DHT), as in the original paper, but it is also possible to make it use Chord [60] instead. This can be accomplished by just changing the name of the base protocol in the specification file (assuming that both DHT protocols implement the same API [2]).

[2]In MACEDON, overlay protocols must implement a specific API, which is similar with the one proposed in [16].

2.4.2 Concurrency Model

In MACEDON, protocols are described by event-driven finite state machines (FSM), which have some drawbacks. Using a FSM model may compromise code reuse, as the code for the actions is tightly coupled with the set of states and transitions of a particular system. Moreover one has to specify all the possible transitions for each state, even if some are not relevant (or an error).

The events are asynchronous and include things like timer expiration, message reception, and API function calls. In order to exploit parallelism and improve performance of reads, MACEDON can process multiple events in parallel, by using a thread pool. However, this entails some effort on the programmer side, as he has to specify which event handlers are read-only and which may modify the state. Given this information, MACEDON is able to generate code that implements an exclusive or a reader locking mechanism to maintain the consistency. Handlers not marked as read-only (control operations in their terminology) are thus serialized within a protocol instance, while the others may run in parallel.

2.4.3 Implementation

The source-to-source compiler generates plain C++ from the specification language that can then be compiled with a regular C++ compiler (along with the runtime library). The compiler also creates the protocol messages definitions, data packet receiver and demultiplexer, state variables and transitions, and so on.

2.5 Mace

Mace [28] is a framework for building distributed systems that succeeds MACEDON (described in the previous section). As with MACEDON, Mace provides a set of extensions to C++ in order to allow more succinct specifications of distributed systems. Mace includes a source-to-source compiler, and a model checker, and supports automated profiling and debugging. Mace's authors claim reductions[3] in terms of lines of code from 1.7x to 13.6x (approximately) when comparing the implementation of several protocols in Mace and the reference implementations in traditional languages (C++ and Java).

2.5.1 Aspect Orientation

Similar to MACEDON, Mace's programs are described as reactive state transition systems. The major difference is the addition of support for aspects (from the Aspect Oriented Programming – AOP – paradigm [27]), which describe computations that cut across event or layer boundaries. With aspects, the programmer can implement cross-cutting concerns in a single place. These include, for example, logging and failure detection, which otherwise would have to be implemented across the whole system. Aspects are only evaluated at state transition boundaries to avoid checking possibly inconsistent states in the middle of a transition.

2.5.2 Concurrency Model

In Mace, protocols can be layered and may execute concurrently. However, event handlers within a given layer execute without preemption and thus state transitions happen atomically. This restriction frees the programmer from worrying about concurrency, but may result in worse performance than MACEDON.

[3]However the paper does not mention how feature complete the implementations in Mace were versus the reference implementations.

2.5.3 Debugging

Another important feature of Mace is its debugging support. Mace supports logging at a semantic-level and includes tools to automatically aggregate and analyze logs from several nodes (see [47]). Mace supports event-level logging, which captures the order and timings of events at each node, state-level logging, which records the state transitions, and message-level logging, which logs the content and transmission time for each message sent and received by each node.

2.5.4 Model Checker

Mace also provides a model checker, named MaceMC [29], that checks for liveness violations. The work of the checker is simplified due to the restrictions in the Mace language, like structured layering and the state-event (atomic) processing semantics. This allows MaceMC to easily replay executions and to perform deep random walks.

More recently, a new model checker for Mace has been proposed – CrystalBall [70] – that is capable of exploring a larger portion of the state space. However, these model checkers cannot consider the whole state space, since the employed exploration techniques perform enumeration search that cannot be exhaustive if there are infinitely many protocol states.

2.6 Acute

Acute [56] is a general-purpose programming language designed for distributed computation. It is built on top of Objective Caml (OCaml), which is a popular variant of the Caml language. Acute is formally defined, including its syntax, typing compilation, and operational semantics [55]. Acute's main features include: type-safe marshalling, universal type names, and controlled dynamic rebinding of marshalled values. The accompanying interpreter is reportedly 300 times slower than the OCaml bytecode interpreter and it interprets the abstract syntax tree (extended with closures) directly.

2.6.1 Type-Safe Marshalling

Acute provides type-safe marshalling in the language. This is particularly useful when sending and receiving data to/from the network. Marshalled values thus retain type information, which is used and verified when unmarshalling them. However, marshalling and unmarshalling must be done explicitly. For example, to send the number 42 over the wire, one must write the following:

```
IO.send( marshal "StdLib" 42 :  int )
```

2.6.2 Global Naming

To support heterogeneous systems, where several versions of a program may coexist, Acute offers several features to help handling this burden. One of them is a global naming of types. The idea is that different versions of a given type (e.g., a structure that was augmented with an additional field) have different identifiers. This feature allows different versions of a given program to notice if they exchange marshalled data whose types have the same name but are not equal (one can think of this as a structural type system spanning multiple nodes). These *globally-meaningful* type names are generated automatically at compile-time by hashing the abstract syntax of the module up to alpha equivalence, although they can also be overridden by the programmer at run-time.

2.6.3 Dynamic Linking

Acute also supports marshalling of code and thus implements dynamic linking and rebinding. Security concerns aside, transferring code between nodes poses again the problem of heterogeneity in the deployed code. Unmarshalled code is dynamically linked and may depend on other modules. Such modules may be of a different version than the ones found in the node that sent the code or they may not even exist at all. Acute provides two mechanisms to help mitigate this problem: it allows the developer to provide a set of URIs (using an ad hoc language) where the dependencies can be automatically fetched by the interpreter, and it supports module versioning. The responsibility for sane versioning is left to the programmer. When importing a module, a constraint on the version can also be specified using wildcards (e.g., `2.3.*`).

2.7 P2

The P2 project[4] started in 2003 with the recognition [21] that structured peer-to-peer networks could reuse some of the distributed databases' technology already available, such as distributed aggregation and recursive queries. Since then, the project has produced several similar declarative languages specialized for distributed systems. Those languages differ mostly in minor syntactic details and they are all heavily based on Datalog [10], a recursive query language with bottom-up evaluation semantics. The new languages created by the project are: declarative routing [36, 37], OverLog [35], Network Datalog (NDlog) [34], and Snlog [12]. OverLog was designed with overlay networks in mind, while Snlog targets sensor network systems. Snlog also features a lighter runtime, due to the resource restrictions in the targeted (embedded) devices. These languages are further described in Boon Thau Loo's PhD thesis [33].

2.7.1 The Language

Datalog is the base of P2's languages. It arose from the database community as a recursive query language for deductive databases [45] and it has been adapted by the P2 project to describe distributed protocols. Although it is syntactically a subset of the Prolog logic programming language, Datalog's query evaluation is different from Prolog's. Datalog's query evaluation is done in a bottom-up fashion, while Prolog uses a top-down evaluation strategy.

In P2, protocols are specified using a set of logic rules that derive new tuples when they are triggered. There are two type of tuples: the event tuples and the materialized tuples. Event tuples are used to trigger other rules, while materialized tuples are stored for a certain (configurable) period. Materialized tuples, which constitute the network state, are stored distributively across the network in relational tables.

The order of predicates in the body of a rule is semantically irrelevant, although they are evaluated from left to right in practice. This means that different orders may result in very different query execution plans, and so the programmer has to implicitly choose between left and right recursion. A discussion of the benefits of each recursion method in this domain can be found in, e.g., [34, 36, 37].

The P2 languages have several drawbacks. First, there are many variations of the main language, which create a lot of confusion around the system. Second, none of the languages has clearly specified semantics, despite the recent efforts [41]. Third, none of the languages is expressive enough. For example, a recent effort to implement several Byzantine fault tolerant (BFT) protocols failed to implement the protocols exclusively in P2, and ended having to create several extensions in C++ [57]. Moreover, the P2 languages lack widely used constructs, such as the if-then-else statement and non-primitive data types, which makes P2 programming difficult and counter-intuitive.

[4] http://p2.cs.berkeley.edu

2.7.2 Implementation

The current P2 compiler takes as input one of its languages and compiles the program down to a distributed dataflow execution plan. Thus a program is in fact a distributed query.

For execution, it uses a set of well-studied evaluation algorithms from the deductive database literature (e.g., [45, 69]) for distributing and executing the program. It uses a semi-naive fixed point evaluation [3], together with aggregate selections [44, 61] and magic-sets [5] optimizations. The optimizations performed by the compiler are expressed themselves in OverLog [14].

2.7.3 Debugging

Debugging in P2 can be viewed as a mater of making a distributed query over the network, by executing a set of rules. Logging can be done in the same way. The support for debugging, logging and monitoring functionalities in P2 – and Datalog based languages in general – is described in [1, 58].

There also exists a performance simulator for P2 [57] that has been used, for example, to benchmark Byzantine fault tolerant (BFT) protocols.

2.7.4 Static Analysis

On the verification side, there exists two tools to verify programs exclusively written in P2 (i.e. without extensions in C++). DNV [65] is a tool that converts P2 programs into axioms. Then the user can encode the properties he wants to prove as theorems, which get automatically discharged by the PVS theorem prover [42]. Cardan [40] abstracts a P2 program into a set of cardinality constraints, and then verifies the program using the ARMC model checker [43].

2.8 Summary

In summary, we have that nesC, P2, and DAHL have event-driven specification and execution models, while MACEDON and Mace are finite state machine based. Moreover, only nesC, MACEDON, and Mace support some sort of concurrency. On the verification side, many languages already provide bug finding tools, although only P2 and DAHL provide tools for property proving (not described in this thesis).

Chapter 3

The DAHL Language

In this chapter, a comprehensive description of the DAHL programming language is given. The DAHL language is based on Prolog [59, 62].

The presentation is divided in four parts. First we introduce the language with an example. We then describe how data is represented and manipulated by programs, and then we present the syntax and semantics of DAHL. Finally, we describe the built-ins and language constructs provided by DAHL to support the development of distributed systems.

3.1 DAHL by Example

As an introductory example to the DAHL language, we present an implementation of the Bully protocol [19]. This classic protocol implements a distributed leader election algorithm which is suitable for synchronous systems.

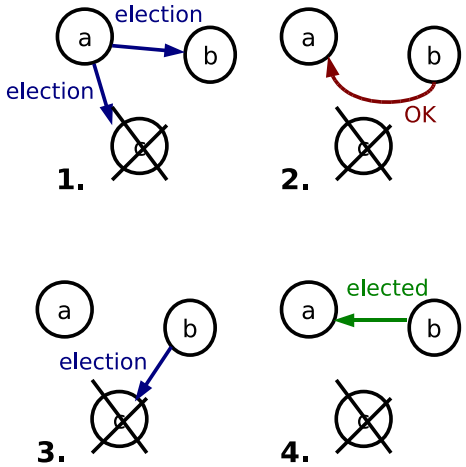

Figure 3.1: Sample execution of the Bully protocol.

A quick introduction to the protocol is given following Figure 3.1. In step 1, node 'a' detects that node 'c' (the current leader) failed. Therefore it broadcasts an 'election' message to all its neighbors with an identifier greater than its own (in this case, nodes 'b' and 'c'). Node 'b' receives the message from node 'a' and replies with an 'OK' message (step 2). At the same time, node 'b' starts the election protocol in the same way that node 'a' did in step 1 (step 3). Finally, as node 'b' does not receive any reply during the timeout period, it becomes the leader, and broadcasts itself as such (step 4).

```
1  :- table(node(Id, Address), [key(Id), key(Address)]).
2
3  election :-
4          this_node(MyAddress),
5          sendall(Addr,
6                  ( node(Id, Addr), Id > MyId ),
7                  election(MyAddress)),
8          expect(ok, timeout, 3000).
9
10
11 event(election(Addr)) :-
12         send(Addr, ok),
13         election.
14
15
16 event(elected(Id)) :-
17         log('Elected: ~p', [Id]).
18
19
20 timeout :-
21         this_node(MyAddress),
22         node(MyId, MyAddress),
23         log('I am the leader: ~p', MyId),
24         sendall(Addr, node(_, Addr), elected(MyId)).
```

Figure 3.2: Implementation of the Bully protocol in DAHL.

Figure 3.2 shows the full source code of the implementation of the protocol in DAHL. Line 1 declares the node table, which consists of pairs in the form $(Id, Address)$. To simplify the example, we assume that each node has its own table populated with information about all the nodes in the system.

Lines 3–8 implement the election protocol: lines 5–7 broadcast an 'election' message to all nodes with an identifier greater than its own, and line 8 asks the system to execute the `timeout` predicate if no 'ok' message arrives within 3 seconds. The `election` predicate is the one that should be called whenever a node detects that the leader crashed (not shown in this example).

Lines 11–13 define the event handler for the 'election' event. It sends an 'ok' message back to the sender of the event (line 12), and then runs the election protocol (line 13).

Lines 16–17 define the event handler for the 'elected' event. It simply logs that there is a new leader. In a real system, this handler would probably also need to save the new leader to some variable.

Finally, lines 20–24 define the predicate that is called when no 'ok' message was received within the timeout period. Line 21 queries the system for the node's address, and line 22 retrieves the node's identifier from the table out of the address. Line 23 logs the event, while line 24 broadcasts the 'elected' event to all nodes.

3.2 DAHL data structures and storage

The central data structure of DAHL is that of *terms*. Terms are used to represent a wide variety of data types ranging from primitive ones (integers, floats, and strings) to arbitrary complex compound structures, e.g., tuples, lists, trees, queues, sets, and bags.

Data manipulation in DAHL is performed in a declarative fashion, e.g., data structures are traversed by first decomposing them into components and then iteratively descending into the appropriate component. Data structures are immutable, i.e., they cannot be modified in-place, and a program variable can be

bound to a data value only once. The flow of data in DAHL programs can be organized in three ways: assigning a value to a variable, as parameter passing between procedures, and by storing/retrieving data tuples from the DAHL database.

The database is a storage facility that, since it persists during the program execution, can be used by developers for maintaining the application state. DAHL extends the storage facilities provided by standard Prolog implementations with automatic indexing for fast retrieval, akin to relational database management systems, as well as user defined key constraints that enforce uniqueness of data lookup by keys. This allows updates to the database to be done in a predictable, fast, and declarative way.

We included this feature in DAHL as it proved to be useful for the development of distributed systems in the P2 language, which was later confirmed by us while developing applications in DAHL.

The database organizes the stored data into tables. In order to use DAHL's extensions, each table has to be declared. Note that standard Prolog tables can still be used without declaration, although without the benefits provided by DAHL (Prolog only provides limited indexing capabilities). The syntax for declaring tables is the following:

`:- table(`*Template*`, [`*Options*`]).`

where *Template* is a term describing the name and number of fields in the table, and *Options* is a list containing terms of the form:

`unique`, `key(`*Vars*`)`, or `index(`*Vars*`)`.

The `key` options, as well as its special case `unique`, impose constraints on the combinations of tuples that can be stored on the table. Specifically for each assignment to the variables in the `key`, at most one tuple can be stored in the database. For example,

```
:- table(f(X, Y), [key(X)]).
:- table(g(X, Y), [key(X), key(Y)]).
:- table(h(X, Y), [unique]).
```

declares `f` to behave as a function so that, for each value of the key `X`, there is at most one corresponding value `Y`. Similarly, `g` is defined to satisfy the constraints of a one-to-one mapping. Finally, the table `h` can store at most one tuple. Note that this is the special case of a `key` with no variables. The `index` options specify fields to be automatically indexed for fast retrieval. Fields declared with `key` are also indexed, but `index` does not impose any constraints on the number of tuples that can be stored for each assignment to the index variables.

3.3 DAHL clauses and operational semantics

DAHL borrows its syntax from Prolog and other similar languages in the context of constraint logic programming (CLP). Code is organized into *clauses* of the form:

Head `:-` *Body*`.`

Such a clause defines that the predicate in the *Head* should be evaluated according to the query-like expression defined by its *Body*. The *Body* is an arbitrary expression built from operators such as conjunction, negation, and conditionals, as well as calls to built-in or user defined predicates. The actual semantics used to evaluate the bodies of clauses is inherited from Prolog's procedural semantics which allow the execution of operations with complex control flows, including iteration, recursion, backtracking and side-effects.

Broadly speaking, the semantics of evaluation can be described as follows. To evaluate a predicate, also known as a *goal*, the system searches for a clause declaration whose head *unifies* with the goal. If a match is found, the matching clause is activated and each of the goals on its body are themselves evaluated from

left to right. If, at any time, the system fails to find a match for a goal, it backtracks undoing previous unifications, and reconsiders alternative matches for the activated clause that got rejected. For a more precise and detailed description, we refer the reader to the rich body of literature on Prolog's books and manuals [59, 62].

We devote the rest of this section to describe the additional features and extensions – and their semantics – particular to DAHL.

The control structure of a DAHL program, suitable for the development of distributed and reactive systems, is driven by an event manager. The event manager triggers the evaluation of clauses for events received from the network, or caused by the expiration of alarms. This is achieved by declaring special clauses of the form:

`event(`*Head*`) :-` *Body*`.`
`alarm(`*Head*`) :-` *Body*`.`

These are respectively called event and alarm handlers. An event handler is called automatically when a corresponding event arrives from the network. Alarm handlers are called when an associated timer expires. Alarms (that can be used, for example, for periodic message retransmission) are further described in the following section.

By selectively marking which clauses can be triggered remotely, we ensure that no random local clause can be triggered by a possibly malicious remote host. This scheme is similar to the interface description languages (IDL) commonly used by remote procedure calls (RPC) systems.

3.4 DAHL built-in predicates

In order to support the development of distributed applications, and to be able to describe them at a high level of abstraction, DAHL provides a comprehensive set of built-in predicates. Such predicates include communication and security primitives, predicates for storing and retrieving data from DAHL's database, as well as facilities for logging and diagnosis.

A brief overview of each of these predicates is given in the rest of the section.

3.4.1 Networking

DAHL provides a rich set of primitives that enable communication of nodes over the Internet in either unicast, broadcast, or multicast mode.

In the following predicates, *Address* is in the format IP:Port or Hostname:Port. All timing related parameters are specified in milliseconds.

`this_node(`*Address*`)`

Unifies *Address* with the address of the node executing the program.

`send(`*Address*`,` *Message*`)`

Sends *Message* to the node at the corresponding *Address*. This predicate fails if an error occurs when trying to send the *Message*, e.g., because the remote host is down. Note that due to the semantics of the underlying transport protocol (TCP), it is not guaranteed that a message is delivered even if the predicate succeeds.

Messages are delivered in-order to the remote node in respect to a single sender, i.e., if a given node sends more than one message to another node, they are delivered in the same order as they were sent.

When *Message* is received by the node at *Address*, the corresponding event handler is evaluated. The *Message* is discarded if none exists. Nodes cannot use the `send` facility to trigger the evaluation of alarm handlers or private predicates in other nodes.

Note that the delivered *Message* does not include, unless explicitly added as part of its content, the address of sender node (compare with `send_signed`).

Example code that sends the event `ping` to the host 'node3' listening on port 42:

```
send(node3:42, ping)
```

`sendall(`*Address*`,` *Generator*`,` *Message*`)`

`sendall(`*Address*`,` *Generator*`,` *Message*`,` *Failed*`)`

Sends *Message* to each *Address* for which the goal *Generator* succeeds.

This predicate never fails, but executes the *Failed* predicate (if one is provided) for each node that could not be contacted.

If a `neighbor` table is kept with the addresses of all the neighbors of a node, a broadcast operation can be implemented as:

```
sendall(Node, neighbor(Node), hello)
```

`expect(`*Template*`,` *TimeOut*`,` *Expired*`)`

Instructs DAHL to remember that a message matching *Template* is expected to be received from the network. If a matching message is received within the specified *TimeOut* period, the corresponding event handler is evaluated as usual. Otherwise, if no such message arrives on time, then the goal *Expired* is evaluated instead.

An exception is thrown if the system is already expecting a message with a more general template than *Template*.

The wait for the incoming message is non-blocking, and thus the system continues receiving and processing other events while awaiting for the expected message.

This built-in allows a simple implementation of request/retry communication idioms. Consider the following example where a client retransmits a request every 10 seconds to a server, until it receives the appropriate response.

```
make_request(Server, Request) :-
  this_node(Client),
  send(Server, request(Client, Request)),
  expect(response(Server, Request, Response),
         10000,
         make_request(Server, Request)).

event(response(Server, Request, Response)) :-
  % do something with the Response
```

`alarm(`*Event*`,` *Time*`)`

`alarm(`*Event*`,` *Time*`,` *Periodic*`)`

Tells the DAHL system to later evaluate *Event* in the specified amount of *Time*. The predicate fails if there is already a timer declared for *Event*.

If *Periodic* is `true`, then the event is periodically triggered at intervals specified by *Time*.

The corresponding handler for *Event* must have been declared as an alarm handler.

As an example, consider the following code to send an heartbeat to all neighbors every 5 seconds:

```
init :-
  alarm(heartbeat, 5000, true).

alarm(heartbeat) :-
  this_node(MyAddress),
  sendall(N, neighbor(N), heartbeat(MyAddress)).
```

`has_alarm(`*Event*`)`

The goal succeeds if there is currently an alarm set for *Event*, and fails otherwise.

`stop_alarm(`*Event*`)`

If any, an alarm previously set for *Event* is canceled.

`reset_alarm(`*Event*`, `*Time*`)`

`reset_alarm(`*Event*`, `*Time*`, `*Periodic*`)`

These are semantically equivalent to calling `stop_alarm` followed by `alarm`.

3.4.2 Security

DAHL extends the communication primitives with authentication capabilities. For example, the predicate `send_signed(`*Address*`, `*Message*`)` sends an envelope to *Address* containing *Message* together with a secure signature identifying the sender as its legitimate source. Similarly, the `sendall_signed` primitive extends `sendall`. On the receiving end, one can use the predicate `signed_by(`*Node*`)` to validate the incoming event, and retrieve the address of the sender *Node*.

For the most part, signing and validation of signatures is automatically done behind the scenes, so that the DAHL programmer does not have to worry about signatures. It is still possible, however, to get hold of the actual data of a signature, if there is a need to store it or send it for re-validation by third parties.

`send_signed(`*Address*`, `*Message*`)`

Sends to *Address* an envelope containing *Message* together with a secure signature identifying the sender as its legitimate source. The predicate fails if there is an error trying to send the message (see `send` for the exact fail semantics).

It is not necessary to include the address of the sender in *Message*, as this information is already contained in the envelope. For the details on the actual signing procedure, see `sign` below.

Envelopes are automatically opened on the receiving node, and the corresponding event handler for *Message* is evaluated as usual. Verification of the signature is done on demand by the recipient (see `signed` below).

`sendall_signed(`*Address*`, `*Generator*`, `*Message*`)`

`sendall_signed(`*Address*`, `*Generator*`, `*Message*`, `*Failed*`)`

These have the same semantics as `sendall`, but all the messages sent are signed.

`signed`

`signed_by(`*Principal*`)`

`signed_by(`*Principal*`, `*Signature*`)`

Verifies that the message that triggered the current event handler was properly signed. These predicates fail if the message was not signed, or if the signature is invalid.

If requested, *Principal* is unified with the address of the sender node and *Signature* with the provided signature.

As an example, consider a secure version of a token passing protocol, where nodes only accept `pass` messages that have been successfully authenticated.

```
event(pass) :-
  signed,
  add(token),
  alarm(release, 3000).
```

```
alarm(release) :-
  del(token),
  neighbor(Node),
  send_signed(Node, pass).
```

`sign(`*Message*`,` *Signature*`)`

Computes a signature to authenticate the current node as the source of *Message*, and unifies *Signature* with the result.

The signing algorithm used can be selected as a global option. It is read from a predefined table declared as

```
table(sign_algorithm(Algorithm), [unique]).
```

The only algorithm supported at this time is HMAC+MD5 [30, 50].

The private key used to sign the messages is read from a predefined table declared as

```
table(private_key(PrivateKey), [unique]).
```

`verisign(`*Principal*`,` *Message*`,` *Signature*`)`

The predicate succeeds only if *Message* is properly signed by *Principal* with the correct *Signature*. Public keys for signature validation are read from a predefined table declared as

```
table(public_key(Node, PublicKey), [key(Node)]).
```

The `signed` family of predicates are implemented using this predicate, so in most cases there is no needed to call `verisign` directly.

3.4.3 Local state

DAHL provides several primitives to retrieve and manipulate the local database, that is an extension to Prolog's own database. The usage of these built-in predicates is not mandatory, although they provide several benefits over Prolog, such as more powerful indexing capabilities.

The built-ins include, for example, `add(`*Tuple*`)` and `update(`*Tuple*`)` that are used to add tuples to the database, while the `del(`*Term*`)` primitive is used to deletes all matching tuples from the database.

`add(`*Tuple*`)`

Adds the given *Tuple* to the database. The predicate fails if the addition of *Tuple* violates a key constraint.

All the values in *Tuple* corresponding to keyed columns must be ground.

`update(`*Tuple*`)`

Adds *Tuple* to the database after removing, if necessary, all the conflicting tuples.

`del(`*Term*`)`

Deletes all tuples from the database that are instances of *Term*.

`get(`*Term*`)`

Finds an instance of *Term* currently stored in the database, and unifies *Term* with it. If no such instance exists, the predicate fails. Upon backtracking, `get` generates all instances of *Term* stored in the database.

Defined `key` and `index` options in the corresponding table declaration are used, whenever possible, for fast retrieval of tuples.

For illustration, consider the following example that shows the interaction between `add`, `update`, and `get` predicates.

```
:- table(g(X, Y), [key(X), key(Y)]).

:- add(g(1, c)).    % ok
:- add(g(2, a)).    % ok
:- get(g(1, Y)).    % Y = c
:- get(g(X, a)).    % X = 2
:- add(g(1, d)).    % fail, conflicts with g(1,c).
:- update(g(1, d)). % ok
:- get(g(1, X)).    % X = d
:- update(g(1, a)). % ok
:- get(g(2, Y)).    % fail, g(2, a) no longer in store
```

3.4.4 Additional control and development support

DAHL also provides several general primitives for simplifying the control flow of program, as well as some debugging facilities.

`all(`*Goal*`)`

Computes, without storing, all solutions of *Goal*. This predicate is useful, for example, to iterate over a set of nodes and execute some action on them. It is equivalent to `findall(_, `*Goal*`, _)`.

As an example, consider the following code that prints all the neighbors that have a latency smaller than 10 ms (provided the `neighbor` table exists and has such information):

```
p :-
  all((
    neighbor(N, Latency),
    Latency < 10,
    format('good neighbor: ~p (latency: ~d ms)\n', [N, Latency]).
  )).
```

`ensure(`*Goal*`)`

Throws an exception if the evaluation of *Goal* fails.

`optional(`*Goal*`)`

Evaluates *Goal* and succeeds, regardless of whether *Goal* succeeded or not.

`bound(`*Goal*`, `*Time*`)`

Evaluates *Goal* and raises an exception if the execution does not stop before the specified amount of *Time* in milliseconds. The predicate succeeds whenever *Goal* succeeds in the allowed time.

`log(`*Control*`)`

`log(`*Control*`, `*Arguments*`)`

Appends to the private log of the node a line with the current time (formatted as an ISO 8601 [24] string) and a string representation of the *Control* term. The variant with two arguments accepts the same formatting options as the standard Prolog `format/2` predicate.

`vprint(`*Control*`)`

`vprint(`*Control*`, `*Arguments*`)`

If the runtime is executed in *verbose* mode these commands print a line to the standard output with a visual representation of *Control*. Otherwise, they silently succeed. The variant with two arguments accepts the same formatting options as the standard Prolog `format/2` predicate.

3.4.5 Important Prolog built-in predicates

The following are a few predicates already built-in in the Prolog language, but that are particularly useful for describing the complex control flows required by distributed applications. We refer the reader to a

Prolog User's Manual [62] for further details and information.

`(`*If* `->` *Then* `;` *Else*`)`

Evaluates the *If* goal and, if it succeeds, evaluates the *Then* goal. Otherwise, the *Else* goal is evaluated. It also 'cuts' any choices left open by the *If* goal, committing to the first solution found.

This construct can also be used to implement a if-else-if idiom, e.g.:

```
rel(X, Y) :-
  ( X > Y ->
    print('x > y\n')
  ; X < Y ->
    print('x < y\n')
  ;
    print('x = y\n')
  ).
```

`\+` *Goal*

Negates *Goal*, i.e., fails if *Goal* has a solution, and succeeds otherwise.

`throw(`*Exception*`)`

`catch(`*Goal*`,` *Exception*`,` *Handler*`)`

Exception throwing and catching mechanism. Whenever an exception is thrown, evaluation of *Goal* is abandoned. If *Exception* can be unified with the value of the exception thrown, *Handler* is evaluated. Otherwise, the exception is thrown again, looking for an ancestor exception handler.

If an event handler throws an exception that is not caught, the runtime catches the exception, logs this information, and continues the evaluation of further events (but immediately stops the evaluation of the current event). Optionally, the programmer can decide to kill the entire application if an exception is not caught.

`findall(`*Template*`,` *Generator*`,` *List*`)`

Unifies *List* with all the instances of *Template* for which the goal *Generator* succeeds.

For example, the following code constructs and prints a list with all the numbers smaller than 42 that are in the `number` table.

```
list :-
  findall(Val,
          (
            number(Val),
            Val < 42
          ),
         List),
  print(List).
```

`once(`*Goal*`)`

Computes only the first solution of *Goal*, i.e., even if *Goal* has multiple solutions, the evaluation will not backtrack to find other solutions of *Goal*. This is often used as an optimization, when one knows that indeed *Goal* can have at most one solution, to prevent the Prolog interpreter from storing unnecessary backtracking information.

Chapter 4

DAHL Runtime Implementation

This chapter gives an overview of the current DAHL runtime. It starts with a presentation of its implementation, which is then followed by a description of how to execute a program written in DAHL. Finally, the results of a synthetic benchmark to assess the performance of our implementation are presented.

4.1 Implementation

DAHL is implemented using the SICStus Prolog compiler [62]. It consists of a runtime library written in Prolog (with around 460 lines), and an optimized networking back-end written in C (around 450 lines of code). The networking back-end uses libevent [38] to efficiently dispatch the incoming events from the network. The back-end also interfaces with the OpenSSL library [63] in order to implement the cryptographic primitives in the language. In order to interface with Prolog, the back-end uses stubs that are generated automatically by the SICStus Prolog compiler. The software architecture of DAHL is shown in Figure 4.1.

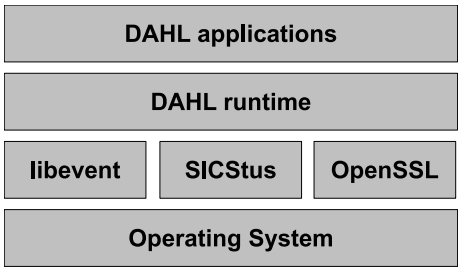

Figure 4.1: The DAHL software stack.

4.1.1 Runtime

DAHL programs are interpreted directly by the SICStus Prolog compiler, but under the DAHL runtime control. The main program in execution is part of the runtime, and a DAHL program's code is only called when an appropriate event arrives from the network, or when a timer is triggered.

The runtime works by running a loop to receive an event and dispatch it to the right handler. The loop is implemented in Prolog, and the process to wait for events is done by libevent. This library performs a non-active wait for network and timer events (if the underlying operating system supports that).

Figure 4.2 shows the execution flow for processing a message that arrives from the network (steps 1–4), and for a message that is sent from an application (steps 5–7) in more detail. When a message arrives from the network, the operating system dispatches it to libevent (step 1), which queues the message.

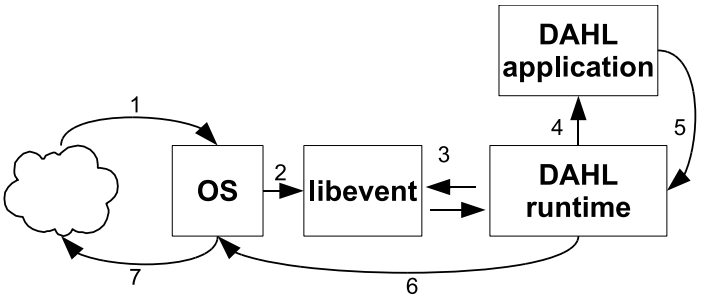

Figure 4.2: Execution flow for message processing.

Then, when the DAHL runtime asks for the next message, libevent picks one arbitrarily and delivers it to the DAHL network back-end (step 2). The DAHL network back-end then deserializes the message and calls the runtime dispatcher (in Prolog) through a stub (step 3). Finally, the dispatcher triggers an event to the corresponding event handler of the application (step 4). When a DAHL application sends a message, the message is first handed over to the DAHL runtime through a stub (step 5). The runtime then serializes the message and delegates the network transmission to the operating system (step 6).

4.1.2 Optimizations

We implemented several optimizations in the DAHL runtime to improve its performance. Here, we present these optimizations in detail. The deserialization of network messages was a CPU-intensive operation since the SICStus Prolog compiler implements this operation in Prolog through a complex process chain. Since each message sent was serialized to a single atom, it led to an explosion in memory usage because the SICStus Prolog compiler aggressively caches all atoms. We therefore implemented our own custom deserialization in C to improve performance. This resulted in a performance improvement of the deserialization function of about 70%.

As described before, the main loop is implemented in Prolog, and it calls a function in C that "produces" events through libevent, which are then dispatched from within the Prolog environment. After an event is dispatched and processed, the loop backtracks until the beginning of it. This provides an important advantage, which is that every event/alarm handler is executed in a "clean" environment, as all the garbage possibly left by a previous handler is discarded. Moreover, it improves the performance of the garbage collector (GC), as the SICStus Prolog compiler will delete most of such garbage when backtracking as an optimization, reducing the overhead of the GC. Our tests show that without this environment cleanup, the overhead of the GC would be noticeable (from 8% to 45%).

4.1.3 Timers

The implementation of timers is based on libevent. We maintain an ordered list of timers that have not expired yet in Prolog, so that libevent only knows about the next timer to expire. When a timer expires, libevent calls our runtime handler, which then triggers an event to the DAHL application, and reschedules the next timer on the list.

4.1.4 Network support

Currently all the network events are sent using the TCP protocol, which requires establishing a connection before the first contact. The DAHL runtime automatically establishes these connections when needed, and caches them indefinitely for future contacts. Other caching policies can be easily implemented in the runtime if required by the application. Moreover, although the TCP protocol establishes a bidirectional

channel, TCP connections opened by the DAHL runtime are used unidirectionally. This is because it would require extra communication in order to reliably detect that a given input channel corresponds to the address of an outgoing message (including the port number), which would not be desirable for some applications (such as sensor networks, or any bandwidth restricted application).

4.2 Running a DAHL Program

The easiest way to run a DAHL program is to type the following in the console:

```
$ dahl -start filename.pl
```

This command will trigger the execution of the DAHL interpreter, which will load the `filename.pl` file and execute it (the 'init/0' predicate is called by default). There are many configuration options that can be changed through command-line arguments. Those are listed in the following section.

4.3 Command-Line Options

Table 4.1 lists the command-line options supported by the DAHL interpreter. DAHL Programs can specify their own options. The notation <*foo*> means that *foo* is a non-optional argument to the option.

Argument	Description
-start <*filename*>	Load the <*filename*> file for posterior execution.
-term	Start a terminal where it is possible to give commands in DAHL in an interactive fashion.
-init <*goal*>	Set the initial goal to <*goal*> (which can be an arbitrary term). By default, the initial goal is 'init', i.e., by default a program starts with the execution of the 'init' predicate.
-host <*hostname*>	Set the host name seen by the program to <*hostname*>. By default, the DAHL interpreter will try to get this information automatically. However, as some systems do not provide the correct information, this option can be used to manually set it.
-port <*port*>	Set the port number in which the DAHL interpreter will listen for messages to <*port*>. The default is 9999.
-noalarms	Disable the alarm events, i.e., no event is delivered when an alarm is triggered.
-v	Enable the verbose mode. The verbose mode makes, for example, the DAHL interpreter print a message to the terminal every time an event is either sent or received. It also activates the warnings for non-fatal errors (e.g., a message could not be sent, or an event handler failed to process an event). Enabling this switch slows down the interpreter.

Table 4.1: Command-line options supported by the DAHL interpreter.

4.4 Basic Performance Evaluation

To evaluate the performance of the DAHL runtime, we performed a test to compare the performance of P2, Mace, C, and DAHL. We performed a simple network ping-pong experiment. One of the machines (called a client) sends a small 20 byte 'ping' message to the other machine (called a server) which immediately responds with a small 20 byte 'pong' message. We use as many client machines as needed to saturate the server in order to measure its raw throughput. The measurement of the number of requests served per second is done at the server. The machines were connected by a gigabit switch with a round trip latency of 0.09 ms, and both the network and the machines were unloaded.

The results are presented in Table 4.2.

P2	DAHL	DAHL w/ -Oh	Mace	Mace w/ -O2	C
230	14,000	19,220	14,221	21,937	142,800

Table 4.2: Basic performance comparison between P2, DAHL, Mace and C (in requests per second).

First, we note that the DAHL runtime outperforms P2's performance. We believe that the reason behind P2's poor performance is that the runtime of P2 is not yet optimized while DAHL uses SICStus Prolog 4 [62] compiler that has been extensively optimized. Second, DAHL is as fast as Mace. However, given that Mace is a restricted form of C++, it can exploit powerful C++ compiler optimizations. For example, with the '-O2' set of optimizations of gcc 4.1, Mace's performance improves by 60%. As an upper bound on the performance, we also present the performance of a C implementation (using the high-performance networking library libevent [38]) and note that all systems that strive to improve the analysis capability are an order of magnitude slower than it.

Finally, we also note that the SICStus Prolog compiler does not perform any significant code optimizations. As an experiment, we modified the part of the code of the server that updates the number of served requests so that it did not use dynamic tuples nor DAHL's `add` built-in predicate. The first version used a single Prolog blackboard fact (i.e. updated using `bb_get` and `bb_put` predicates), and the performance increased by 27%. The second version used a function in C with a native integer to store and increment the value. This solution increased the performance by 37% over the original solution to 19,220 requests per second (denoted as -Oh – human optimized – in the table above). This experiment shows that the performance of DAHL could be improved with some simple code optimizations, which were not implemented due to time constraints.

Chapter 5

Chord

In this chapter, an implementation of Chord [60] in DAHL is evaluated. Chord is a popular peer-to-peer (P2P) distributed hashed table (DHT), also called a P2P overlay. Chord maintains its member nodes in a logical ring, and allows members to lookup the node that is responsible for a given key in the ring. The maximum number of hops for a lookup is $\lceil \log N \rceil$, where N is the number of nodes in the ring (as each time a lookup query is forwarded, the search space is reduced at least in half). This bound allows Chord to scale to millions of nodes.

In Chord, each node maintains a so-called finger table of size $\log m$, where m is the number of bits of the node identifiers. This table holds the address of nodes with certain identifiers that are exponentially spaced over the ring, i.e., it has more entries for close nodes (in the ring) than for long-distanced ones. Each node also maintains the address of the successor and the predecessor nodes in the ring. These nodes, as well as the entries of the finger table are periodically refreshed.

The purpose of this evaluation is twofold. First, we wanted to measure a networking driven application performance by using application specific benchmarks. For this purpose, we ran the standard benchmarks for P2P overlays. Second, we wanted to compare the performance and succinctness of our implementation with the reference implementation (in C) and with other implementations in state-of-the-art languages for distributed systems. Our implementation of Chord implements all the features contained in the original paper [60], but using recursive queries and 160-bit identifiers.

To the best of our knowledge, P2 is the only state-of-the-art language for the implementation of distributed systems to offer an implementation of Chord. Therefore we compare our results only with P2. To compare with the P2 Chord implementation, we obtained the latest release of P2 at time of writing. [1] Unfortunately, we were unable to get P2's Chord implementation running in our local setup. We therefore cite results from their paper [35].

5.1 Implementation in DAHL

As with most networking protocols, Chord does not require many specific network built-ins. In particular it only needs a unicast and a multicast operations. However, DAHL's features allow a simple implementation of Chord. For example, consider the following operation from Chord's stabilization procedure. Request the successors of all the successors of a given node and, at the same time, discard those successors to whom the request delivery failed can be expressed in DAHL in a one-liner:
`sendall(`*Succ*`, succ(_,` *Succ*`), reqSuccs(`*MyAddr*`), del(succ(_,` *Succ*`)))`

The signature of the `sendall/4` predicate is: `sendall(`*Address*`,` *Generator*`,` *Message*`,` *Failed*`)`. So,

[1] Version 3570 in https://svn.declarativity.net/p2/trunk.

the line above can be read as follows. For each successor in the `succ/2` table, try to send a `reqSuccs/1` event. If the event delivery fails, delete the successor from the `succ/2` table.

Also important is the asynchronous and event-based nature of DAHL. Because of this, our implementation of Chord can naturally pipeline multiple lookups without much effort on the programmer's side. The only concern for the programmer is to tag each lookup, so that when a lookup response is received, it is possible to identify to which lookup it refers to. As a nice side-effect, all the periodic lookups performed by the finger fixing procedure are done in parallel – and without interference – with the normal application lookups.

Other important built-in predicate is `findall/3`. We use it in Chord to iterate over the finger table and select the best finger (i.e., the closest finger to the key being looked up):

```
bestFinger(Id, FingerAddr) :-
  findall(D-FingerAddr,
          (
            finger(_, FingerId, FingerAddr),
            in_range(FingerId, MyId, Id),
            distance(Id, FingerId, D)
          ),
          List),
  min_member(_-FingerAddr, List).
```

One minor negative aspect of DAHL, and of relative importance to overlay protocols, is that DAHL does not provide modular arithmetic in the language, as all numbers are arbitrary long. Overlay protocols, such as Chord, usually need to perform computations of node identifiers and so on that have to be made modulo the ring size. Languages specialized in the implementation of overlays often provide modular arithmetic out-of-the-box. However, we implemented these operations in user-space for Chord, and those routines are small and simple, and do not constitute a major burden.

5.2 Evaluation

5.2.1 Setup

We used Modelnet [64] to emulate a GT-ITM transit-stub topology [17] consisting of 100 to 500 stubs and ten transit nodes. The stub-transit links had a latency of 2 ms and 10 Mbps of bandwidth while transit-transit links had a latency of 50 ms and 100 Mbps of bandwidth. We used 10 physical hosts, each with dual core AMD Opteron 2.6 Ghz processor with 8 GB of RAM, running Linux kernel version 2.6.24. We ran 10 to 50 virtual nodes on each physical node, producing a population of 100 to 500 nodes. In each experiment neither the CPU nor the RAM were the bottleneck. This setup reproduces the topology used by the P2 experiments in [35], although they used Emulab [68].

We have also ran the same experiments in Emulab, but with a smaller set of nodes (only ten), because it was not possible to reserve more machines in a timely fashion. Therefore we do not present the results from this experiment, although they were similar with the the ones obtained with Modelnet.

5.2.2 Static Membership

Our first goal was to see if the DAHL implementation of Chord met the high-level properties of the protocol. We have first evaluated our implementation by performing 10,000 DHT lookup requests generated from a randomly chosen node in the network for a random set of keys. The lookups were generated after waiting for five minutes after the last node joined in order to let the network stabilize. The nodes joined the overlay sequentially using a random landmark.

In Figure 5.1, we present the cumulative distribution of latency incurred to receive the response to the lookup requests with 100 and 500 nodes. The results are comparable or better than the published results for P2's Chord [35].

In Figure 5.2 we present the frequency distribution of the number of hops taken to complete the lookups. As expected, the average number of hops taken by each query is around $\frac{\log N}{2}$ and the maximum number of hops is under the theoretical limit of $\lceil \log N \rceil$.

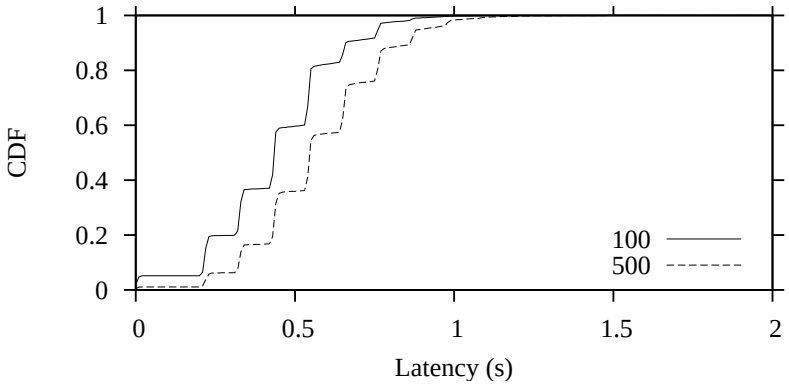

Figure 5.1: Chord: Lookup latency distribution.

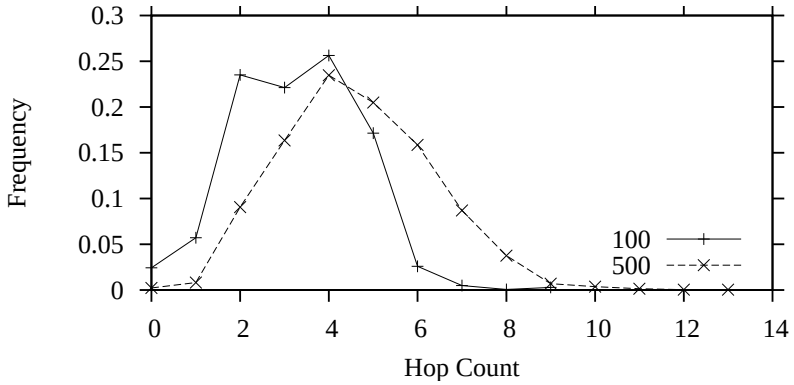

Figure 5.2: Chord: Hop count distribution.

5.2.3 Dynamic Membership

Our implementation of Chord in DAHL also handles churn. In this experiment we used 500 nodes, each one maintaining four successors and performing finger fixing every 10 seconds and successor stabilization every 5 seconds. This configuration is similar to the setup of P2's Chord. We generated artificial churn in our experiment by killing and joining nodes at random with different session times by following the methodology presented in [48]. Figure 5.3 presents our results for three churn rates. We found that with 30 minutes of session time, our implementation delivers approximately 66% of lookup consistency while with 60 minutes of session time, we obtain lookup consistency of 96%. While these results are lower than the consistency results for P2's Chord (they achieve 84% of lookup consistency for 32 minutes and 97% of consistency for 64 minutes of session time), our implementation can be further tuned to improve the lookup consistency by, for example, implementing retransmission of queries on timeout.

As our implementation of Chord does not perform retransmission of timed-out queries, it is normal that our consistency will be lower than MIT's original implementation. Moreover, as their implementation is iterative and ours is recursive, the results should not be compared directly.

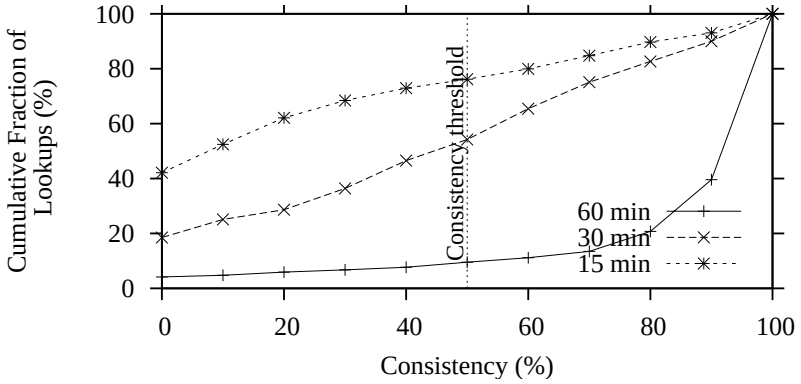

Figure 5.3: Chord: Lookup consistency under churn with different session times.

5.2.4 Code Size and Effort

Our implementation of Chord is comparable in size to the P2 implementation in terms of lines of code (LoC). DAHL's Chord is implemented in 215 LoC while P2's is implemented in 211 LoC. These sizes are an order of magnitude more succinct than MIT's reference implementation in C. [2]

Chord was implemented in DAHL in one week by one person, who had implemented the protocol before in C#. During that period, we also did some simple tests in our cluster (like testing the result of lookups, and graphically displaying the membership graph). The deployment of Modelnet and the churn experiments took another week.

5.3 Summary

Our results show that our implementation of Chord in DAHL covers the major algorithmic aspects of the Chord protocol and that its performance is competitive with P2's Chord. In particular, our implementation shows an average hop count of $\frac{\log N}{2}$ for a lookup, and maintains a lookup consistency of 96% for 60 minutes of session time, which is similar to other implementations. Moreover, in terms of code size, our implementation is an order of magnitude smaller than the reference implementation, and about the same size as P2's implementation.

In summary, DAHL delivers good performance in the benchmarks of a network protocol. The implementation of this protocol is also simple and succinct.

[2] Available from http://pdos.csail.mit.edu/chord/snapshots/.

Chapter 6

D'ARMC: Distributed Abstraction Refinement Model Checking

Model checking is a prominent way to verify the correctness of software and hardware designs (e.g., [4, 22, 25, 26]). However, model checking techniques suffer from the fundamental problem of state space explosion during exploration. Moreover, the number of states in software model checking is potentially infinite. Scalability of model checking for large, real systems thus remains a challenge.

Several techniques to overcome the problem of state explosion exist, such as abstract interpretation [15], which tries to abstract only the relevant properties of a program to prove its correctness. These abstractions can be automatically computed using, e.g., counterexample guided abstraction refinement (CEGAR [13]).

However, even the state-of-the-art techniques can take several weeks to verify large systems. This chapter is focused on distributing the computation of abstraction refinement (software) model checkers in order to improve their performance. The ARMC model checker [43] is used as the base for the distributed version, named D'ARMC, which stands for Distributed Abstraction Refinement Model Checking.

Distributing the current model checking algorithms is not a difficult task. However, if done in a naive way (e.g., reusing the algorithms found in sequential model checkers), the performance of the system will not match the expectations. Due to the inherent non-determinism of a distributed setting, the states of a reachability tree will most likely be discovered in different orders in multiple runs. As different exploration orders may lead to different counterexamples, which in turn may lead to a different set of predicates, the run time of naive algorithms is not predictable. With D'ARMC, we propose a new algorithm that improves the robustness of the system without sacrificing the performance.

6.1 Architecture

The architecture of the solution consists of a standard master-slave system. The master is responsible for assigning work to slaves, as well as keeping the global view of the reachability DAG. The slaves can be (dynamically) assigned two different tasks: compute the one step (abstract) reachability operation, or check the feasibility of a given counterexample and refine the abstraction if the counterexample is spurious (i.e., not feasible in the concrete model).

It is the responsibility of the master to store the global view of the reachability DAG. To achieve this, slaves communicate with the master each time they discover a new reachable state. To avoid excessive communication with the master, each slave keeps a partial view of the reachability DAG as a cache. This way, some abstract state entailment checks can be done locally without contacting the master. This cache

reduces the entailment checks in the master by about 50% (and in some cases by up to 85%) in our tests, but obviously it is highly dependent on the input program.

To argue for the correctness of the addition of the cache, we note that a slave's cache is a (possibly non-strict) subset of the global DAG. So, any state that is discarded using local information would also be discarded if using the global DAG because the abstract DAG computation is monotonic. Any missing information in the cache only implies a performance penalty (as the slave might send a redundant state to the master), but it does not affect the correctness of the algorithm.

Table 6.1 and Table 6.2 present the events that are handled, respectively, by the master and by the slave nodes.

Event	Description
register_slave/1	Called by a slave when it wants to join a (possibly on-going) computation. This allows slaves to dynamically join a computation and obviates the need for the master to know *a priori* how many slaves there will be.
get_work/1	Called by a slave when it has finished the previously assigned piece of work. Returns a piece of work that can be either an abstract state to apply the one-step reachability operator or a counterexample to check for feasibility and possibly refine.
assert_state/5	Add a new abstract state to the DAG and, if it was previously unknown, add it to the work queue.
new_preds/1	Add a new set of predicates to the global predicate set. Called by a slave after refining a counterexample. It is the responsibility of the master to clean duplicates and "non-interesting" predicates when merging the new set with the current one (although slaves also try to perform this action as best as they can with the information they have locally).
feasible_ce/1	Called when a feasible counterexample was found, meaning that the program is not correct. It makes the master (distributively) abort the computation and print the offending counterexample to the user.

Table 6.1: D'ARMC: Events handled by the master.

Event	Description
process/4	Process an abstract state (i.e. apply the one-step reachability operator), and send back to the master its immediately reachable children.
check_ce/1	Check if a counterexample is feasible. If not, refine it and send the new predicates back to the master.
new_preds/1	New set of predicates to be used in following operations. This event is received at the beginning of every iteration.
new_depth/1	Assert the depth of the smaller counterexample found so far. This is sent by the master when the value decreases, so that the slaves can locally discard abstract states and counterexamples that have a greater depth.
the_end/0	The algorithm has terminated and thus the process should end.

Table 6.2: D'ARMC: Events handled by the slaves.

6.2 Algorithm

In this section, the D'ARMC algorithm is presented. The algorithm is split in several pieces for ease of explanation.

We use the following conventions. A transition $\tau \in \mathsf{Trans}$ is defined as $m \xrightarrow{\tau} n$ (that is, m is the parent of n). The depth of abstract states is given by the function $\mathsf{D} : \mathsf{Node} \to \mathbb{N}_0$, where $\mathbb{N}_0$ is the set

of natural numbers starting in 0. The set of reachable nodes is defined as $\mathsf{R} \subseteq \mathsf{Node}$. Finally, the parent relation is given by the function $\mathsf{P} : \mathsf{Node} \to 2^{\mathsf{Node} \times \mathsf{Trans}}$, which is initialized as $\mathsf{P}(n) = \emptyset, n \in \mathsf{Node}$. The notation $F[X \mapsto Y]$ means update the function F to map X to Y thereafter.

In order to develop a deterministic algorithm for a distributed setting, we needed a conservative definition of counterexample. So, for a given k, we define a *non-subsumed* counterexample as a set of states $\{s_1, \ldots, s_k\}$ such that $s_1 \xrightarrow{\tau_1} \ldots \xrightarrow{\tau_{k-1}} s_k$, and that the following property holds:
$\forall_{1 \leq i \leq k} : \neg(\exists s' \in R \setminus \{s_i\} : \ s_i \subseteq s' \wedge \mathsf{D}(s') \leq \mathsf{D}(s_i))$.

The global variables used across the program are the following: R, D, and P (corresponding to the functions described previously), Pruned (the set of pruned nodes), Slave (the set of possibly on-line slaves), Idle (the set of idle slaves), and $\mathsf{depth\_smaller\_ce}$ (holds the depth of the smallest counterexample found so far, or ∞ if none was found).

The main loop cycle in the master assigns work to slaves. Pieces of work consist either of states that should be explored, or counterexamples that need to be checked for feasibility. Pieces of work are kept in a priority queue, where states with smaller depth have higher priority, in order to implement a BFS search. In each iteration, all shortest non-subsumed counterexamples are computed (although not all are necessarily refined).

We use the BFS exploration order, as it was shown to produce good results, and is often better than other strategies, like DFS. Moreover, ARMC also uses BFS as its default exploration order.

Termination of an iteration and of the program happens when all the slaves become idle. If there is no counterexample in the list, then the program was proved correct, otherwise the master assigns counterexample checking tasks to slaves. We note that the detection of termination of the algorithm can be done locally by the master because slaves only ask for a new piece of work after completing the previous one. If the slaves were prefetching work pieces (to, e.g., reduce the penalty of latency between master and slaves), a more complex (and possibly distributed) protocol would be needed.

6.2.1 Example

Before presenting further details on the algorithm, we present a sample execution of the algorithm on a simple example, with two slaves. We run the algorithm on the program in Figure 6.1 to show that the assertion is never triggered, i.e., that states e_1 and e_2 are not reachable from the root (s_1).

```
main() {
    int x = 0;
    x++;
    assert(x >= 1 && x <= 2);
}
```

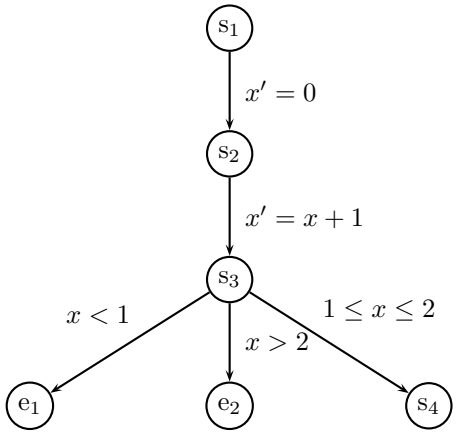

Figure 6.1: Sample program in C and its corresponding control flow graph. Update expressions of the form $x' = x$ were omitted for brevity.

The algorithm starts by adding the (abstracted) start node ($s_1^{\#}$) to the queue. Once the first slave joins the computation, $s_1^{\#}$ is assigned to it. We note that the set of predicates is empty and thus $s_1^{\#}$ is also represented by an empty set of predicates (i.e., it is restricted by the true condition). The slave then

applies the one-step reachability operator to $s_1^\#$ and obtains $s_2^\#$, which is then sent back to the master, which directly assigns the state to the other slave. Using the same procedure, $s_3^\#$ is discovered. As $s_3^\#$ is represented by the true condition, all its children are reachable (for example, $\mathsf{true} \rightarrow x < 1$ is true, and so $e_1^\#$ is reachable).

At this point, we have $e_1^\#$, $e_2^\#$, and $s_4^\#$ in the queue. Exploring the error states ($e_1^\#$ and $e_2^\#$) leads to the discovery of two counterexamples, and the exploration of $s_4^\#$ does not add anything to the queue (as s_4 does not have any children), and so the iteration ends after these three states are explored. The important thing to note is that the error states are explored in a non-deterministic order. So, an algorithm that stops an iteration when the first counterexample is found would not be deterministic in this case. The reason is that the refinement of a counterexample may eliminate another counterexample, but the opposite might not be true. So, different orders of exploration greatly influence the set of predicates, as well as the total number of iterations. The resulting DAG of the first iteration is show in Figure 6.2 (a).

D'ARMC in turn gathers all the shortest non-subsumed counterexamples and sorts them deterministically. As we have two slaves, we can refine both counterexamples in the same iteration (one each slave). We could, however, refine just the first counterexample, according to some total order function. The new set of predicates will be, for example, $\{x \geq 0, x \geq 1, x \leq 0, x \leq 2\}$. There are several other valid predicate sets. We use an algorithm that performs linear interpolation [54].

Rerunning the procedure just described with the new set of predicates leads to the DAG in Figure 6.2 (b). This time, states $e_1^\#$ and $e_2^\#$ are not reachable anymore (for example, for $e_1^\#$ we have that $x \geq 0 \wedge x \geq 1 \wedge x \leq 2 \rightarrow x < 1$ is false). As the whole tree was explored and no error states were found, the program was proven correct.

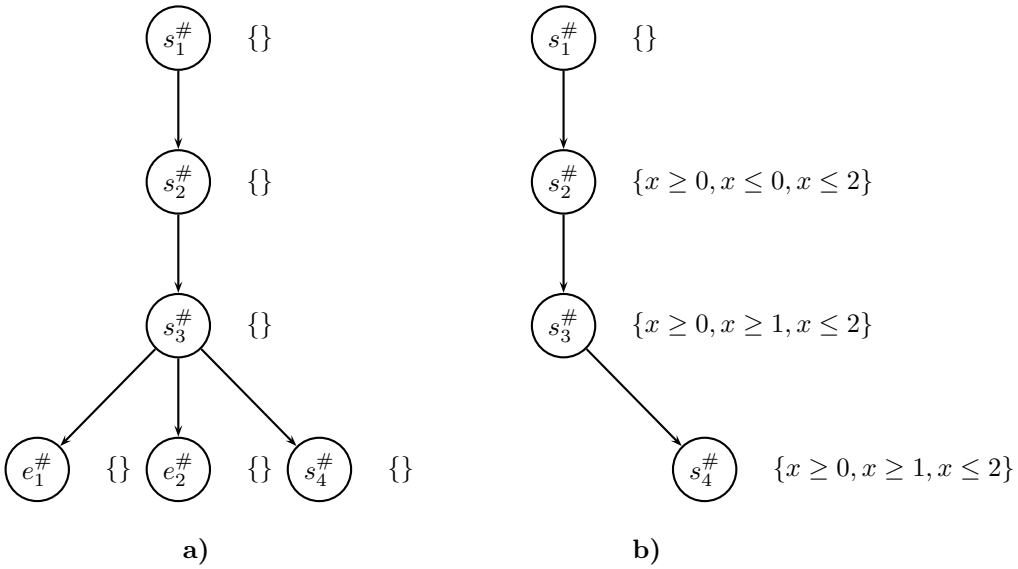

Figure 6.2: DAGs of the first (a) and second (b) iterations generated by D'ARMC when run on the program of Figure 6.1.

6.2.2 Add new abstract states

The algorithm to add newly discovered abstract states to the reachability DAG is shown in Figure 6.3. It can be called by both the master and the slaves when they encounter an abstract state that they believe is relevant. This algorithm guarantees deterministic exploration of the DAG between refinement iterations regardless of all the non-determinism present in a distributed environment. This effectively means that although abstract states may (and most likely will) be discovered in different orders when run multiple times, the algorithm ensures that the resulting reachability DAG at the end of an iteration will be the

```
     algorithm assert_state
     input
        mode ∈ {"master","slave"}
        m: Node
        n: Node
        τ: Trans, with m --τ--> n
        d_n: ℕ
     begin
1       prune({n' ∈ R | n = n' ∧ d_n < D(n') ∨ n' ⊂ n ∧ d_n ≤ D(n')})
2       if n ∈ R ∧ d_n = D(n) then
3           P := P[n ↦ P(n) ∪ {(m, τ)}]
4       if ¬(∃n' ∈ R : n ⊆ n' ∧ d_n ≥ D(n')) then
5           R := R ∪ {n}
6           D := D[n ↦ d_n]
7           P := P[n ↦ {(m, τ)}]
8           if mode ="master" then
9               if n ∩ Error ≠ ∅ then
10                  return "new counterexample"
11              else
12                  add_work(n, d_n)
13          else
14              send assert_state(m, n, τ, d_n) to master
     end.
```

Figure 6.3: Algorithm to add an abstract state to the reachability DAG.

same. This invariant is not preserved by a naive algorithm (e.g., similar to the ones found in sequential model checkers), as the resulting DAG is dependent on the processing order of the results from the slaves (which is non-deterministic).

This invariant is in fact crucial to achieve consistent performance results. Although it does not influence the correctness of the algorithm, it guarantees that the time needed to prove a program correct will be consistent across multiple runs. This happens because different exploration orders may lead to different sets of counterexamples and thus to a different set of predicates, which greatly influences the total number of iterations and the time taken by each iteration. Our tests indicate that the run time of a naive algorithm can vary by 150–3300%, sometimes leading to run times that are worst than sequential algorithms.

The algorithm (in Figure 6.3) works as follows. It starts by pruning non-interesting states that can be so deterministically, i.e., states that are subsumed by the node being added and whose depth is smaller than the new node (line 1). Then, if the new node is not subsumed, the node is added to the DAG (lines 4–14). If it is a slave executing the code, the node is sent back to the master (lines 13–14), otherwise the node is added to the work queue (line 12). As an optimization, if a node with the exact abstraction was already present in the DAG and its depth is equal to the new one, a new parent is added to the old node instead of adding the new node to the DAG, to avoid unnecessary re-exploration of the same subtree (lines 2–3).

As an example, consider the two DAGs in Figure 6.4. On side (a), we are asked to add the node s_4 to the DAG. As it subsumes another node with a higher depth (s_3), it is added to the DAG and the state s_3 and its children are pruned (see Section 6.2.3). On side (b), we want to add the node s_4 to the DAG. As the node s_4 subsumes the node s_2, it is added to the DAG, although node s_2 is not pruned as it has a lower depth. This happens because it is possible that a shorter (or of equal length) counterexample may be found from s_2 than from s_4.

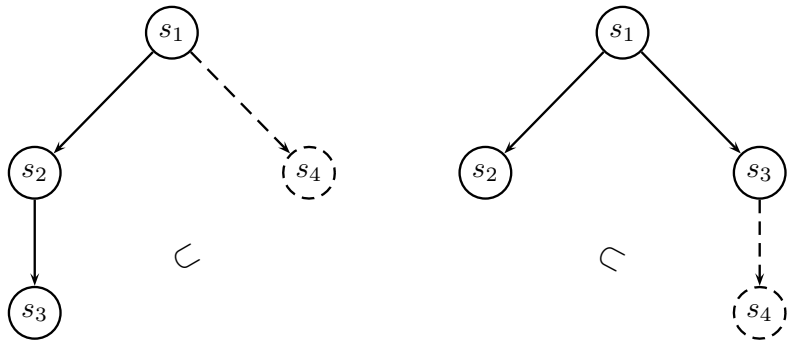

Figure 6.4: Two example cases to illustrate the execution of the assert_state algorithm.

6.2.3 Prune abstract states

The abstract state pruning algorithm is presented in Figure 6.5. Its objective is to prune the entire subtrees rooted at the nodes given as input (that are subsumed by other nodes in the DAG). Nodes with multiple parents are only pruned if they become orphan, i.e., if all the parents were already pruned.

To illustrate how it works, we give an example based on Figure 6.6. Assuming that $\mathsf{Pruned} = \{s_4\}$ and that the prune procedure is called with the argument $\{s_3\}$, the final prune set after executing the algorithm is $\mathsf{Pruned} = \{s_3, s_4, s_6, s_7, s_8, s_9\}$. State s_3 is included in the prune set, as it belongs to the input set. State s_6 is also included as it is a direct child of s_3 and s_3 is its single parent. State s_7 is pruned because its two parents are in the prune set, and so it can be safely pruned. On the other hand, state s_5 cannot be pruned, as it has one remaining parent that is not pruned.

The algorithm works as follows. It starts by adding to the prune set all of its input nodes (line 2) and by removing those nodes from the reachable set (line 3). Then it calls itself recursively to remove the input node's children that have all its parents in the Pruned set (line 4). In particular, if a child only has one parent (an input node), it means that it will be pruned as its parent was just pruned (in line 2).

```
      algorithm prune
      input
        ns: 2^Node
      begin
1       if ns ≠ ∅ then
2         Pruned := Pruned ∪ ns
3         R := R \ ns
4         prune({n ∈ R | ∃(n', τ) ∈ P(n) : n' ∈ ns ∧ ∀(n'', τ') ∈ P(n) : n'' ∈ Pruned})
      end.
```

Figure 6.5: Abstract subtree pruning algorithm.

6.2.4 Retrieve a work piece

Work pieces are organized in a priority queue, where states with lower depth have higher priority (in order to implement a BFS search). So, if the queue is not empty, retrieving a work piece is as simple as dequeuing an item with the highest priority (lines 1–2 of Figure 6.7). As an optimization, only states whose depth is smaller than the smallest counterexample found so far are considered, since others (with equal or greater depth) will never generate counterexamples with equal or smaller depth than the smallest found so far.

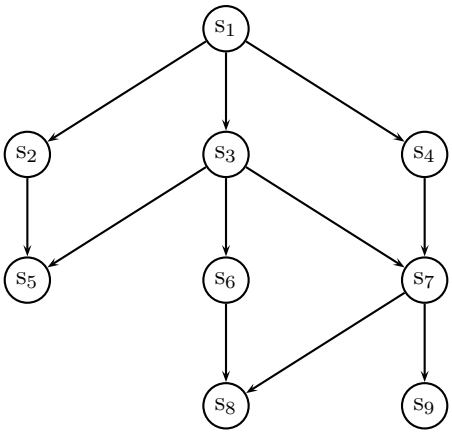

Figure 6.6: Example abstract state reachability DAG.

On the other hand, if the queue is empty, the slave that requested the work piece is added to the list of idles slaves (line 4). If all the slaves become idle (line 5) then it means that the algorithm has finished (line 7), or that at least one counterexample was found (i.e., depth_smaller_ce is not infinite), which needs to be processed (line 9). The argument for termination of the algorithm was given in Section 6.2.

```
      algorithm get_work
      input
        s : Slave
      begin
1       if (w, d) = dequeue() ∧ d < depth_smaller_ce then
2         send w to s
3       else
4         Idle := Idle ∪ {s}
5         if |Idle| = |Slave| then
6           if depth_smaller_ce = ∞ then
7             return "program is correct"
8           else
9             return "found counterexample"
      end.
```

Figure 6.7: Algorithm to retrieve a piece of work.

6.2.5 Add a work piece

The algorithm to add a new work piece to the queue is listed in Figure 6.8. First, states whose depth is greater or equal than the smallest counterexample found so far are rejected (lines 1–2), for the same reason given for the previous algorithm. Then, if there is any idle slave, the work piece is sent immediately to an arbitrary slave picked from the idle list (lines 3–5). Otherwise, the work piece is added to the work queue (line 7).

6.2.6 Pick n counterexamples

Finally, the last algorithm is used in the refinement phase. In order to make the whole algorithm deterministic, we needed a way to always choose the same counterexamples to refine across multiple runs. And that is precisely what this algorithm accomplishes. It picks the first n (configurable) non-subsumed counterexamples in the DAG, according to some (parametrized) total order function, represented by the

```
      algorithm add_work
      input
        w : Work
        d : ℕ
      begin
1       if d ≥ depth_smaller_ce then
2         return
3       if |Idle| > 0 then
4         s := take one from Idle
5         send w to s
6       else
7         enqueue(w, d)
      end.
```

Figure 6.8: Algorithm to add a new piece of work to the queue.

symbol $\prec$. We use the lexicographic order of the transition identifiers for this purpose, and we note that the abstract node identifiers cannot be used for ordering, as they are specific to a run. Transition identifiers are part of the input program, and thus they do not change across multiple runs over the same program.

The ordering function must handle comparison between paths with different lengths. In particular, it must respect the following conventions (assuming that $s_1 \prec s_2 \prec s_3 \prec s_4$): $s_1 \cdot s_2 \prec s_1 \cdot s_3 \cdot s_4$, $s_1 \cdot s_2 \prec s_1 \cdot s_2 \cdot s_3$, and $s_1 \cdot s_2 \cdot s_3 \prec s_1 \cdot s_3$.

The algorithm appears in Figure 6.10. As input, it receives the set of reachable error nodes and the maximum number of counterexamples to select. The algorithm uses a work list to generate the minimal reversed counterexamples (i.e. paths from the error node to the initial node) in a mixed DFS/BFS way. The algorithm also tries to generate the minimum number of counterexamples (as opposed to generate all of them and then pick the minimal), as there can be many of them.

The algorithm starts by initializing the work list with pairs of transition identifiers and nodes, that correspond to the parents of the input nodes and the respective transition (line 2). Then, it iterates over the work list until it is either empty (i.e., all paths have been explored) or the requested number of counterexamples has been generated already (line 3).

The loop works as follows. First it picks the first element from the work list, according to the $\prec$ order function (line 4). Then, if the start node is reached, the path is added to the list of counterexamples (lines 5–6). Otherwise, we still have a partial path, and so we increment the path with the parents of the node, and add the resulting paths to the work list (line 8).

As an example, consider the execution of the algorithm in the DAG of Figure 6.9. The DAG has three counterexamples: $s_1 \cdot s_2 \cdot s_4 \cdot e_1$, $s_1 \cdot s_2 \cdot s_5 \cdot e_2$, and $s_1 \cdot s_3 \cdot s_5 \cdot e_2$. Assuming that $\mathsf{Pruned} = \emptyset$ and that $\mathsf{NumCEs} = 1$, we have that the work list is initialized to $\{(\tau_6, s_4), (\tau_5, s_5)\}$. Then the algorithm picks the first element from the list (according to our $\prec$ function), which is (τ_5, s_5). The first iteration thus changes the work list to $\{(\tau_6, s_4), (\tau_5 \cdot \tau_4, s_2), (\tau_5 \cdot \tau_4, s_3)\}$ (note that the paths are reversed). After the second and third iterations, the work list becomes: $\{(\tau_6, s_4), (\tau_5 \cdot \tau_4 \cdot \tau_1, s_1), (\tau_5 \cdot \tau_4 \cdot \tau_2, s_1)\}$. In the fourth iteration, the element $(\tau_5 \cdot \tau_4 \cdot \tau_1, s_1)$ is picked from the work list. As s_1 is the initial node, this counterexample is added to the list of counterexamples. As NumCEs was defined as 1, the loop terminates, and we return the minimal counterexample path (albeit in reverse order), which is $\tau_1 \cdot \tau_4 \cdot \tau_5$ or, in term of nodes, $s_1 \cdot s_2 \cdot s_5 \cdot e_2$.

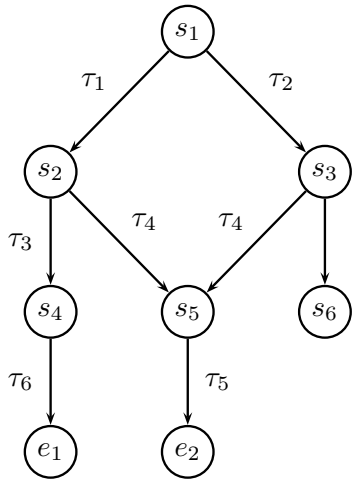

Figure 6.9: Example DAG with two reachable error states (e_1 and e_2).

```
        algorithm pick
        input
           E ⊆ R : reachable error nodes
           NumCEs : maximum number of counterexamples to select
        begin
1          CEs := ∅
2          WL := {(τ, s') | s ∈ E \ Pruned ∧ (s', τ) ∈ P(s) ∧ s' ∉ Pruned}

3          while WL ≠ ∅ ∧ |CEs| < NumCEs do
4              (π, s) := take first from WL according to ≺
5              if s ∈ Init then
6                  CEs := CEs ∪ {π}
7              else
8                  WL := WL ∪ {(π · τ', s') | (s', τ') ∈ P(s) ∧ s' ∉ Pruned}

9          return CEs
        end.
```

Figure 6.10: Algorithm to deterministically pick n counterexamples.

6.2.7 Main master loop

The main loop of the D'ARMC master that glues the several pieces of the algorithm presented in the previous sections is shown in Figure 6.11. The loop iterates over the events received from the slaves, which are queued in no special order. Note that, because of the event-driven nature of DAHL, this loop is implicit in the implementation.

The loop starts by adding the initial state, say s_1, to the work queue, as well as other trivial initializations (lines 3–5). Then, the master waits for incoming events from the slaves. The master handles five events (described in Table 6.1): register_slave, get_work, assert_state, new_preds, and feasible_ce.

The register_slave handler (lines 7–10) simply registers the sender (a slave) in the slaves list. It also sends the predicate list, as slaves may join after the first refinement iteration (and thus the predicate list will be non-empty). As an optimization, the get_work procedure is invoked on behalf of the slave, to avoid an extra round-trip.

The get_work handler (lines 11–18) has the purpose of sending work pieces to slaves. However it may also detect several exceptional conditions. The first is that if the work queue is empty then the program is correct (lines 13–14). Second, if the refinement iteration is over and if there are counterexamples to

refine, then it picks the first NumCEs counterexamples and refines them or asks the slaves to do so (lines 15–16). A new iteration is started after the refinement process is completed (line 18).

The assert_state handler (lines 19–26) adds the input node to the work queue (line 20). If it is an error node, then it is recorded if needed (lines 21–26).

The new_preds and feasible_ce handlers, lines 27–28 and 29–30, respectively, are trivial.

```
        algorithm main_loop
        input
            Program
        begin
1           Preds := ∅
2           while true do
3               D := P := R := queue := ∅
4               depth_smaller_ce := ∞
5               assert_state(NIL, s_1, NIL, 0)

6               foreach incoming event e do
7                   if e = register_slave(slave) then
8                       Slave := Slave ∪ {slave}
9                       send Preds to slave
10                      get_work(slave)

11                  if e = get_work(slave) then
12                      Res := get_work(slave)
13                      if Res is correct then
14                          return "program is correct"
15                      if Res is found_counterexample then
16                          refine(pick(CEs, NumCEs))
17                          broadcast new predicates to slaves
18                          exit foreach loop

19                  if e = assert_state(m, n, τ, d_n) then
20                      S := assert_state("master"m, n, τ, d_n)
21                      if S is a counterexample then
22                          if D(S) ≤ depth_smaller_ce then
23                              if D(S) < depth_smaller_ce then
24                                  depth_smaller_ce := D(S)
25                                  CEs := ∅
26                              CEs := CEs ∪ {S}

27                  if e = new_preds(Ps) then
28                      Preds := Preds ∪ {Ps}

29                  if e = feasible_ce(trace) then
30                      return "program is not correct"
        end.
```

Figure 6.11: Main loop of the D'ARMC master.

6.2.8 Main slave loop

The main slave loop is shown in Figure 6.12. When a slave is started, it first registers itself in the master (line 2), and thereafter loops to wait and process master's messages (lines 3–17). The new_preds event is sent when a new refinement iteration begins, and thus the slave needs to reset the relevant data structures, like caches, and the predicate set (lines 4–6). The process event handler starts by computing

all the successors of the input node (lines 8–9), and then sends them back to the master (line 10). When this process is completed, the slave asks for another piece of work (line 11).

The master can run the refinement in two ways: it either refines the counterexamples itself, or it delegates the task to slaves. In the later case, the master sends a check_ce event. The event handler for this event starts by checking if the counterexample is feasible in the concrete program (line 13). If it is not, then the counterexample is refined, and the resulting set of predicates is sent back to the master (lines 14–15). If the counterexample is indeed feasible, then a feasible_ce event is sent to the master (lines 16–17).

```
        algorithm main_slave
        input
            Program
            self - this node
            master - master node
        begin
1           Preds := ∅
2           send register_slave(self) to master

3           foreach incoming event e do
4               if e = new_preds(Ps) then
5                   Preds := Ps
6                   P := R := D := ∅

7               if e = process(m, d) then
8                   foreach τ ∈ T do
9                       n := α^Preds(post(τ, m))
10                      assert_state("slave", m, n, τ, d + 1)
11                  send get_work(self) to master

12              if e = check_ce(π) then
13                  if ρ_π = ∅ then
14                      Ps := refine(π)
15                      send new_preds(Ps) to master
16                  else
17                      send feasible_ce(π) to master
        end.
```

Figure 6.12: Main loop of the D'ARMC slaves.

6.3 Evaluation

In this section the results of running D'ARMC on a set of benchmarks are presented. The benchmark suite is composed by difficult examples from the transportation domain[1] and a standard hybrid system example, gasburner. Our benchmarks are automata-theoretic models compiled from complex specifications that describe communication, timing, and data manipulation aspects. Table 6.3 provides some details about the size of the benchmarks.

The tests were run on a cluster of AMD Opteron 252 (2.6 Ghz) machines, with 3 GB of RAM, 64 KB of L1 caches, and 1 MB of L2 cache each, running Linux kernel version 2.6.24. The computers were connected through a gigabit LAN with an average RTT of 0.14 ms.

[1] Automatic Verification and Analysis of Complex Systems (AVACS), `http://www.avacs.org`.

Test	Program size		# Predicates	Max. length of counterexamples
	# Variables	# Transitions		
larger_scale1_lb	16	6127	154	15
larger_scale1_ub	16	6127	217	25
scale1_lb	15	3337	98	15
scale1_ub	15	3337	182	25
timing	47	99093	17	17
gasburner	19	3124	209	41
rtall_tcs	20	18757	50	15

Table 6.3: Size of benchmark programs and details of their verification in sequential setting.

Table 6.4 presents a comparison between the number of iterations that D'ARMC performs with different number of counterexamples refined in each iteration and also with the sequential version (ARMC). Table 6.4 also presents the speedups obtained with 40 nodes (i.e., one master and 39 slaves) relative to ARMC run with standard options (the same as D'ARMC). Table 6.5 presents the absolute running times for the same experiment. Figures 6.13 and 6.14 present graphically the speedups obtained in each test, as well as the median, with refinement of 1 and 20 counterexamples per iteration (respectively). Figures 6.15 and 6.16 show the efficiency of D'ARMC for the same tests.

Test	Sequential	Distributed - 1 counterexample		Distributed - 20 counterexamples	
	Iterations	Iterations	Speedup	Iterations	Speedup
larger_scale1_lb	32	43	2.66	14	5.93
larger_scale1_ub	46	57	22.13	37	13.81
scale1_lb	20	28	4.24	16	4.20
scale1_ub	37	43	9.24	31	5.54
timing	14	15	16.73	12	10.97
gasburner	64	83	18.35	37	6.54
rtall_tcs	30	29	28.76	27	9.94
Median	–	–	16.73	–	6.54

Table 6.4: D'ARMC: Speedups and number of iterations with 40 nodes with refinement of 1 and 20 counterexamples per iteration.

Test	Sequential	Distributed - 1 counterexample	Distributed - 20 counterexamples
larger_scale1_lb	1.8	0.7	0.3
larger_scale1_ub	50.9	2.3	4
scale1_lb	0.3	0.1	0.1
scale1_ub	5.9	0.6	1.1
timing	0.7	0.04	0.1
gasburner	5.5	0.3	0.8
rtall_tcs	1.0	0.04	0.1

Table 6.5: D'ARMC: Absolute running times (in hours) with 40 nodes with refinement of 1 and 20 counterexamples per iteration.

6.3.1 Discussion

The idea of refining multiple counterexamples per iteration appeared as a way to use the wasted power of the slaves during the refinement phase. By using the traditional approach of refining just one counterexample per iteration, all the slaves have to wait until the master (or a slave) performs the refinement

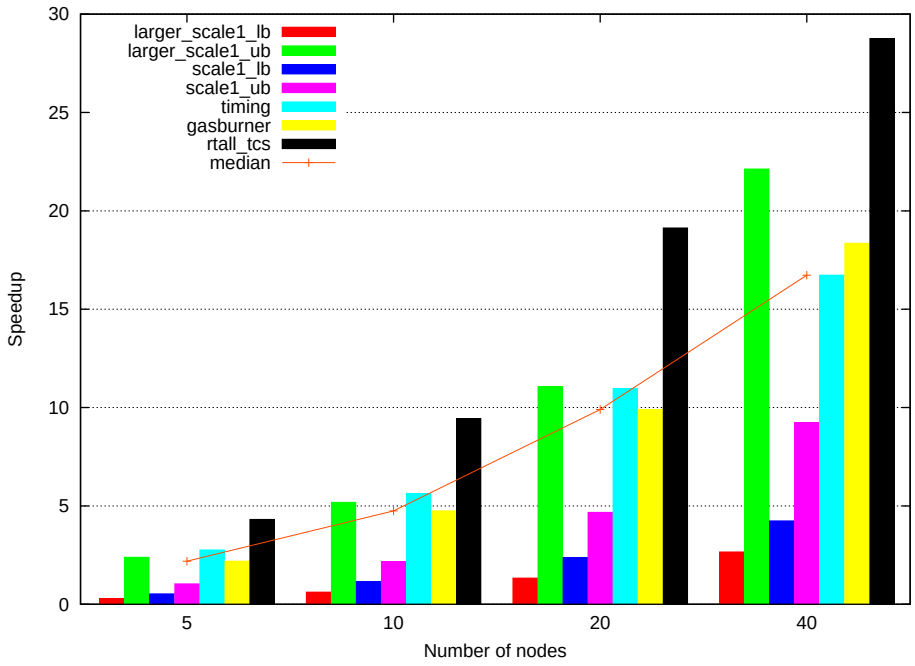

Figure 6.13: D'ARMC: Speedups with refinement of one counterexample per iteration.

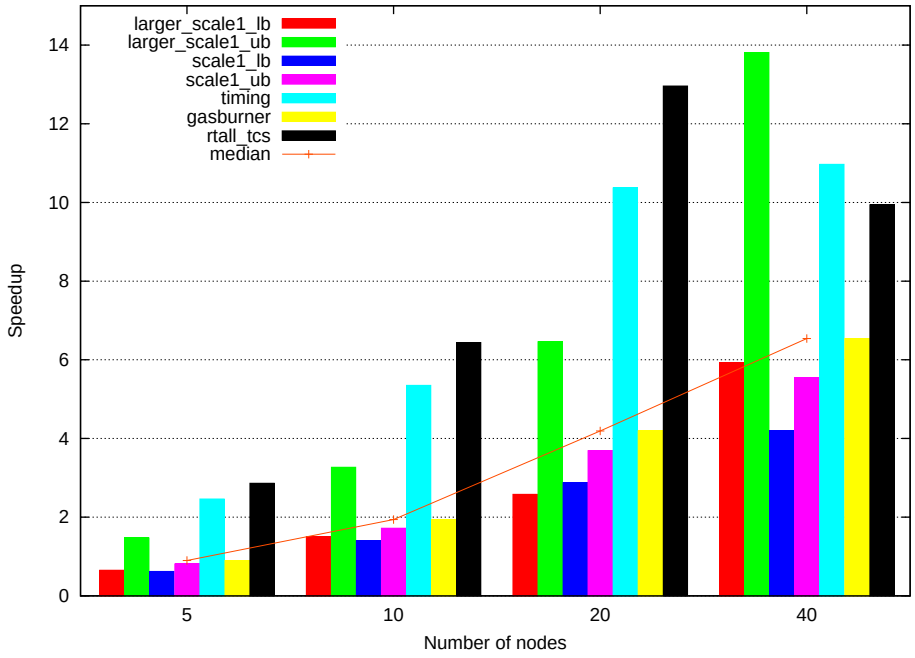

Figure 6.14: D'ARMC: Speedups with refinement of 20 counterexamples per iteration.

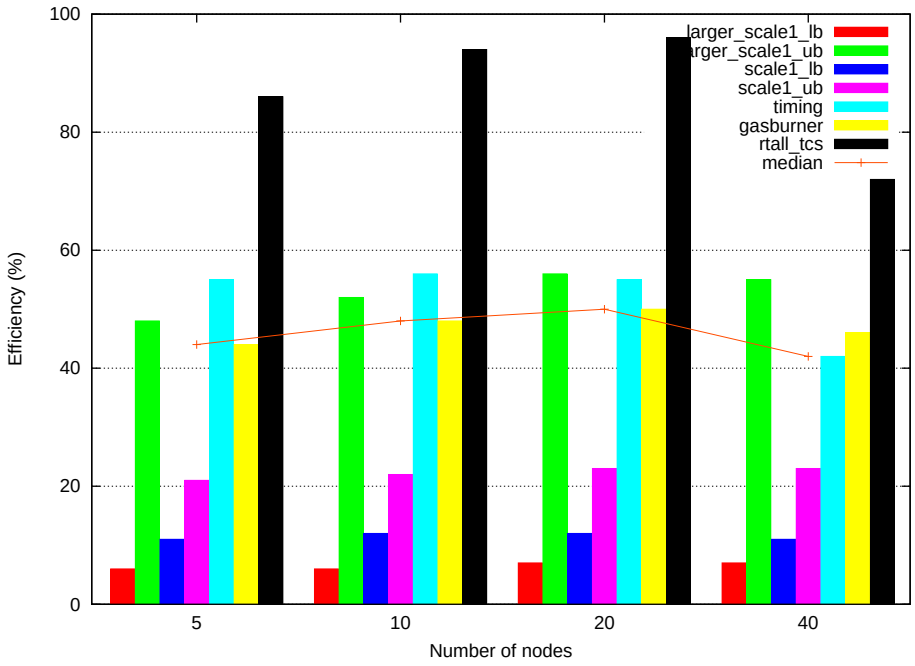

Figure 6.15: D'ARMC: Efficiency with refinement of one counterexample per iteration.

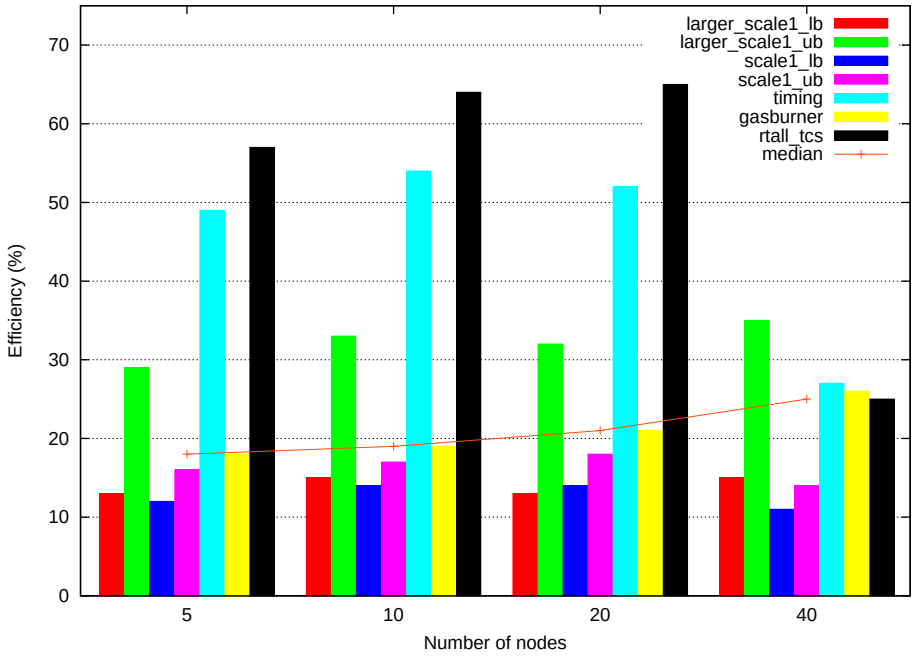

Figure 6.16: D'ARMC: Efficiency with refinement of 20 counterexamples per iteration.

step. Moreover, by refining multiple counterexamples, the total number of iterations can be potentially decreased and so as the total computation time. However, this comes at the expense of increasing the number of predicates, which considerably slow downs the entailment checks in the one-step reachability computation. Table 6.4 shows that in fact the number of iterations are reduced by using 20 counterexamples instead of just one, but it does not pay off the increased cost of the one-step reachability phase. Methods to reduce the cost of the increased number of predicates are certainly necessary in order to make this technique useful. For now, our results suggest that it is better to refine just one counterexample per iteration.

It is also worth noting that even with efficiency rates that are not great (the median is around 50% when refining just one counterexample), the running time is still dramatically decreased. For example, running the larger_scale1_ub test sequentially takes more than 2 days, while D'ARMC takes only a little more than 2 hours. This reduction is certainly a big win.

6.4 Limitations and Future Work

There are several aspects in D'ARMC that can be further improved. For example, currently we use a single one-step reachability operation as a work piece. This can be seen as computing a subtree with depth one. Extending the algorithm to allow the computation of subtrees with arbitrary depth is helpful in several ways. First, it improves scalability, as the communication with the master is reduced (because the work piece size is increased). Second, it allows deployment of slaves across multiple sites, which is currently not advisable, as the increased network latency would become a bottleneck. However, by allowing a slave to independently compute a subtree with depth greater than one, we may end up computing redundant subtrees. The depth of the subtrees should then be (automatically) tuned according to the latency of the network and to the required scalability of the system.

Currently we use only one master server. We believe that our algorithm can be easily extended to support at least two masters: one to keep the reachability DAG and another to manage the work queue. As there will be some communication involved between the master servers, it is not clear if having more than one is actually beneficial.

Another area of improvement lies in the pruning algorithm. Reuse of subtrees instead of pruning whole subtrees altogether was shown to deliver good performance improvements [53]. However this approach may imply more work and increased memory usage in the master, and thus it may reduce the scalability of the system.

Closely related is our definition of non-subsumed counterexample given in the beginning of the chapter, which is a bit loose. We believe that a stronger definition (but that does not void the deterministic property of the system) should both reduce the number of counterexamples per iteration, as well as improve their "quality" (because the remaining counterexamples will potentially include more concrete states that will be refined at once).

Finally, the current system uses a centralized server to store the whole reachability DAG. For programs with a large number of abstract states, this limitation may become a bottleneck due to memory constraints. Distributing the DAG uniformly across the slaves seems a reasonable solution for this case. Note that it is also possible to distribute the work queue (by, e.g., using a work stealing protocol [6,18]), and thus the system becomes fully distributed and potentially more scalable, with each node playing a similar role.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this thesis I have presented DAHL, a new programming system to simplify the construction of distributed programs. By providing a high-level language and an efficient runtime environment, DAHL enables the development of more succinct, albeit clear, programs, while enabling efficient execution and static analysis. DAHL thus provides the succinctness and verification aptitude of the high-level languages, combined with the expressiveness and efficiency of the low-level languages.

The benefits of DAHL are exemplified by building two representative distributed applications, such as Chord, a network-driven protocol, and D'ARMC, a CPU-bound application. To the best of my knowledge, D'ARMC is the world's first scalable distributed software model checker, and is implemented in DAHL.

The applications implemented in DAHL were tested on a local cluster, with and without Modelnet, as well as in Emulab, which shows the high reliability and flexibility of DAHL.

7.2 Limitations and Future Work

DAHL uses its own serialization format to send events over the wire. However, this is a problem if one needs to interact with existing systems or if one needs to conform with some standard packet format. Several techniques exist to address this problem, such as using attributive grammars [2] or DSLs (e.g., [39, 49]). By adopting one of these languages, one can extend DAHL to automatically serialize and unserialize event tuples according to the given specification of the protocol. We leave this as future work as it is not a major limitation in the DAHL system and it can be easily plugged in.

In terms of performance, our DAHL interpreter is one order of magnitude slower than C in terms of raw performance (see Section 4.4). This fact can be a concern for CPU-bound applications, although it should not be for network protocols. We believe the performance of DAHL could be substantially improved with a newer compiler. The one we use (SICStus Prolog) fails to perform simple optimizations (as it is demonstrated in Section 4.4), which are of apparent importance. The raw performance is, however, comparable with the other state-of-the-art systems.

In terms of language design, DAHL can still be further extended with some language features presented in other systems, such as check-pointing, distributed logging, and aspect-oriented extensions.

Bibliography

[1] S. Abiteboul, Z. Abrams, S. Haar, and T. Milo. Diagnosis of Asynchronous Discrete Event Systems: Datalog to the Rescue! In *Proc. of the 24th ACM Symposium on Principles of Database Systems (PODS)*, June 2005.

[2] D. P. Anderson and L. H. Landweber. A Grammar-Based Methodology for Protocol Specification and Implementation. In *Proc. of the 9th SIGCOMM Symposium on Data Communications*, Sept. 1985.

[3] I. Balbin and K. Ramamohanarao. A Generalization of the Differential Approach to Recursive Query Evaluation. *The Journal of Logic Programming*, 4(3):259–262, Sept. 1987.

[4] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic Predicate Abstraction of C Programs. In *Proc. of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2001.

[5] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proc. of the 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS)*, Mar. 1986.

[6] R. D. Blumofe and C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *Journal of the ACM (JACM)*, 46(5):720–748, Sept. 1999.

[7] N. Carriero and D. Gelernter. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys (CSUR)*, 21(3):323–357, Sept. 1989.

[8] N. Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, Apr. 1989.

[9] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications (J-SAC)*, 20(8):1489–1499, Oct. 2002.

[10] S. Ceri, G. Gottlob, and L. Tanca. What You Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, Mar. 1989.

[11] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos Made Live: An Engineering Perspective. In *Proc. of the 26th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, Aug. 2007.

[12] D. Chu, L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica. The Design and Implementation of a Declarative Sensor Network System. In *Proc. of the 5th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, Nov. 2007.

[13] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Proc. of the 12th International Conference on Computer Aided Verification (CAV)*, July 2000.

[14] T. Condie, D. Chu, J. M. Hellerstein, and P. Maniatis. Evita Raced: Metacompilation for Declarative Networks. In *Proc. of the 34th International Conference on Very Large Data Bases (VLDB)*, Aug. 2008.

[15] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, Jan. 1977.

[16] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a Common API for Structured Peer-to-Peer Overlays. In *Proc. of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS)*, Feb. 2003.

[17] K. C. Ellen W. Zegura and S. Bhattacharjee. How to Model an Internetwork. In *Proc. of the 15th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, Mar. 1996.

[18] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Proc. of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, June 1998.

[19] H. Garcia-Molina. Elections in a Distributed Computing System. *IEEE Transactions on Computers*, 31(1):48–59, Jan. 1981.

[20] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proc. of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2003.

[21] J. M. Hellerstein. Toward Network Data Independence. *ACM SIGMOD Record*, 32(3):34–40, Sept. 2003.

[22] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *Proc. of the 29th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, Jan. 2002.

[23] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System Architecture Directions for Networked Sensors. *ACM SIGOPS Operating Systems Review*, 34(5):93–104, Dec. 2000.

[24] International Organization for Standardization. *ISO 8601:2004: Data elements and interchange formats – Information interchange – Representation of dates and times.* 2004.

[25] H. Jain, F. Ivančić, A. Gupta, and M. K. Ganai. Localization and Register Sharing for Predicate Abstraction. In *Proc. of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Apr. 2005.

[26] R. Jhala and R. Majumdar. Software Model Checking. *ACM Computing Surveys (CSUR)*, 41(4), Oct. 2009.

[27] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proc. of the 11th European Conference on Object-Oriented Programming (ECOOP)*, June 1997.

[28] C. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. Vahdat. Mace: Language Support for Building Distributed Systems. In *Proc. of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2007.

[29] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *Proc. of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2007.

[30] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, Feb. 1997.

[31] L. Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, May 1998.

[32] J. W. Lloyd. Declarative Programming in Escher. Technical Report CSTR-95-013, University of Bristol, Aug. 1995.

[33] B. T. Loo. *The Design and Implementation of Declarative Networks.* PhD thesis, University of California, Berkeley, 2006.

[34] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative Networking: Language, Execution and Optimization. In *Proc. of the 2006 ACM SIGMOD International Conference on Management of Data*, June 2006.

[35] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In *Proc. of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2005.

[36] B. T. Loo, J. M. Hellerstein, and I. Stoica. Customizable Routing with Declarative Queries. In *Proc. of the 3rd Workshop on Hot Topics in Networks (HotNets-III)*, Nov. 2004.

[37] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In *Proc. of the ACM SIGCOMM 2005 Conference*, Aug. 2005.

[38] N. Mathewson and N. Provos. *libevent Documentation*, 2009. Release 1.4.9.

[39] P. J. McCann and S. Chandra. Packet Types: Abstract Specification of Network Protocol Messages. In *Proc. of the SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, Aug. 2000.

[40] J. Navarro and A. Rybalchenko. Cardinality Analysis for Declarative Networking. In *Proc. of the 21st International Conference on Computer Aided Verification (CAV)*, June 2009.

[41] J. Navarro and A. Rybalchenko. Operational Semantics for Declarative Networking. In *Proc. of the 11th International Symposium on Practical Aspects of Declarative Languages (PADL)*, Jan. 2009.

[42] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining Specification, Proof Checking, and Model Checking. In *Proc. of the 8th International Conference on Computer Aided Verification (CAV)*, Aug. 1996.

[43] A. Podelski and A. Rybalchenko. ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement. In *Proc. of the 9th International Symposium on Practical Aspects of Declarative Languages (PADL)*, Jan. 2007.

[44] R. Ramakrishnan, K. A. Ross, D. Srivastava, and S. Sudarshan. Efficient Incremental Evaluation of Queries with Aggregation. In *Proc. of the 1994 International Symposium on Logic Programming (ILPS)*, Nov. 1994.

[45] R. Ramakrishnan and J. D. Ullman. A Survey of Deductive Database Systems. *The Journal of Logic Programming*, 23(2):125–149, May 1995.

[46] A. Ranganathan and R. H. Campbell. What is the Complexity of a Distributed System? *Complexity*, 12(6):37–45, July 2007.

[47] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *Proc. of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, May 2006.

[48] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling Churn in a DHT. In *Proc. of the USENIX'04 Annual Technical Conference (ATEC)*, June 2004.

[49] F. Risso and M. Baldi. NetPDL: An Extensible XML-Based Language for Packet Header Description. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 50(5):688–706, Apr. 2006.

[50] R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321, Apr. 1992.

[51] A. Rodriguez, C. Killian, S. Bhat, D. Kostić, and A. Vahdat. MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks. In *Proc. of the 1st USENIX Symposium on Networked Systems Design & Implementation (NSDI)*, Mar. 2004.

[52] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proc. of Middleware 2001: 18th IFIP/ACM International Conference on Distributed Systems Platforms*, Nov. 2001.

[53] A. Rybalchenko and R. Singh. Subsumer-First: Steering Symbolic Reachability Analysis. In *Proc. of the 16th International SPIN Workshop on Model Checking of Software*, June 2009.

[54] A. Rybalchenko and V. Sofronie-Stokkermans. Constraint Solving for Interpolation. In *Proc. of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, Jan. 2007.

[55] P. Sewell, J. J. Leifer, K. Wansbrough, M. Allen-Williams, F. Z. Nardelli, P. Habouzit, and V. Vafeiadis. Acute: High-level programming language design for distributed computation. Technical Report UCAM-CL-TR-605, University of Cambridge, Oct. 2004.

[56] P. Sewell, J. J. Leifer, K. Wansbrough, F. Z. Nardelli, M. Allen-Williams, P. Habouzit, and V. Vafeiadis. Acute: High-Level Programming Language Design for Distributed Computation. In *Proc. of the 10th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Sept. 2005.

[57] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe. BFT Protocols Under Fire. In *Proc. of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2008.

[58] A. Singh, P. Maniatis, T. Roscoe, and P. Druschel. Using Queries for Distributed Monitoring and Forensics. In *Proc. of the 1st EuroSys Conference*, Apr. 2006.

[59] L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques*. The MIT Press, 1994.

[60] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. of the ACM SIGCOMM 2001 Conference*, Aug. 2001.

[61] S. Sudarshan and R. Ramakrishnan. Aggregation and Relevance in Deductive Databases. In *Proc. of the 17th International Conference on Very Large Data Bases (VLDB)*, Sept. 1991.

[62] The Intelligent Systems Laboratory. *SICStus Prolog User's Manual*. Swedish Institute of Computer Science, 2009. Release 4.0.5.

[63] The OpenSSL Project. *crypto(3) - OpenSSL cryptographic library*, 2007. Release 0.9.8g.

[64] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proc. of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.

[65] A. Wang, P. Basu, B. T. Loo, and O. Sokolsky. Declarative Network Verification. In *Proc. of the 11th International Symposium on Practical Aspects of Declarative Languages (PADL)*, Jan. 2009.

[66] R. Y. Wang, T. E. Anderson, and M. D. Dahlin. Experience with a Distributed File System Implementation. Technical Report CSD-98-986, University of California, Berkeley, 1998.

[67] B. Warneke, M. Last, B. Liebowitz, and K. S. J. Pister. Smart Dust: communicating with a cubic-millimeter computer. *Computer*, 34(1):44–51, Jan. 2001.

[68] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.

[69] O. Wolfson. Sharing the Load of Logic-program Evaluation. In *Proc. of the 1st International Symposium on Databases in Parallel and Distributed Systems (DPDS)*, Dec. 1988.

[70] M. Yabandeh, N. Knežević, D. Kostić, and V. Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *Proc. of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2009.