diligent reading of many versions of the present text, and the numerous comments that led to its improvement.

I also thank the other members of the Concurrent Constraint Programming Group—Khayri M. Ali, Kent Boortz, Per Brand, Björn Danielsson, Dan Sahlin, Thomas Sjöland, Galal Atlam, Björn Carlson, Ralph Clarke Haygood, Torbjörn Keisu, and Khaled Shaalan—for taking part in the development of AKL and AGENTS. It has been, and will continue to be, team work.

Other colleagues at SICS, in particular my colleagues in the former Logic Programming and Parallel Systems Laboratory, have provided a highly stimulating working environment. Many thanks to them all.

Of my colleagues in Uppsala, I thank Johan Bevemyr for implementing the first prototype of the AGENTS programming system, Jonas Barklund and Håkan Millroth for advice on the process of submitting a dissertation, and not least Sten-Åke Tärnlund for his struggle for computer science at Uppsala University.

This work was in part funded by the ESPRIT projects PEPMA (EP 2471) and ACCLAIM (EP 7195). Sponsors of PEPMA were the Swedish National Board for Industrial and Technical Development (NUTEK), Ericsson Group, and Swedish Telecom. Sponsor of ACCLAIM was NUTEK. The partners of PEPMA and ACCLAIM provided valuable feedback to the research reported herein.

SICS is a non-profit research foundation, sponsored by the Swedish National Board for Industrial and Technical Development (NUTEK), Asea Brown Boveri AB, Ericsson Group, IBM Svenska AB, NobelTech Systems AB, the Swedish Defence Material Administration (FMV), and Swedish Telecom. I thank these companies and organisations for taking an active interest in Swedish computer science research.

Finally, I thank my wife Kia and sons Adam and Axel for their patience and encouragement, and my parents for raising me to believe that writing a dissertation is something that people typically do.

CHAPTER 1

INTRODUCTION

This chapter discusses the notion of programming language design. A number of design criteria are presented: general design principles as well as specific design goals for the present context.

1.1 WHY DESIGN PROGRAMMING LANGUAGES?

Why design yet another programming language? Do we not already have all the languages we need? Maybe not, languages evolve, even the languages that form the foundation of the software industry. Recently, a new paradigm, object-oriented programming, worked its way into respectability. It started out as an exotic creature, Simula, and even more exotic was Smalltalk, on its exotic host, a personal work station, with a bit-mapped graphic display. Research then, mainstream now. There is already an object-oriented COBOL. Surely, the next FORTRAN standard will include an object-oriented extension.

Some application areas, such as knowledge information processing, still do not enjoy strong support in the major programming languages. Writing such programs requires an undue amount of effort, just as writing object-oriented software is awkward without proper linguistic support.

A number of research languages provide very good support for such tasks, and some of them are commercially available. However, these languages are typically single paradigm languages. Even though other programming paradigms, such as object-oriented programming, would add expressiveness desirable for other aspects of application programming, the question of how to achieve such an augmentation of expressive power, *while staying within the computational framework at hand*, is and will continue to be an active topic for research, and was a strong motivation for the research presented in this dissertation.

Furthermore, a new generation of powerful and inexpensive multiprocessor computer systems has recently emerged, but the range of existing applications able to benefit directly is very limited. Ideally, it should be possible to exploit any parallelism inherent in programs, by automatically determining which parts may be run in parallel, and by doing so when appropriate. To determine this, today's programming languages in popular use require data-flow analysis, since they allow almost arbitrary interactions between statements. Not all parallelism can be discovered with today's analysis techniques, but even if they eventually are perfected, interactions between statements are frequent, and there is usually not much parallelism there to be discovered anyway. It is not likely that this approach will ever be useful as a general means of exploiting the potential of these machines. Too little parallel execution can be achieved.

There are two obvious roads to better utilisation of multiprocessor systems. One is to train highly qualified designers and programmers to perform the intricate task of producing good parallel software, something that is likely to be cost-effective only for standard software. Or, since new programs have to be written anyway, design new, even more structured, programming languages, amenable to automatic exploitation of potential parallelism. Both approaches carry an initial training cost, but the payoff of the latter is likely to be greater since the task of writing the actual parallel programs becomes so much easier.

Our interpretation of the current situation is that there is still a great need to do basic research in programming language design and implementation. Existing languages and programming systems have been found wanting.

1.2 BASIC DESIGN PRINCIPLES

It is easy to list a number of programming language design principles that are so general and so obviously appealing that any designer in his right mind would claim adherence to them. But, although such a list may therefore seem as simple-minded as the lyrics of a simple love song, firsthand experience can often give it both depth and meaning.

Any programming language should be simple, expressive, and efficient, at least from the programmer's point of view, but these criteria should also be applicable to the computation model, and its formal manipulation, as well as to the basic implementation technology.

- A *simple* language has few basic concepts, with simple interactions. A simple computation model and a simple basic implementation are often a natural consequence.
- An *expressive* language supports concise implementations of a wide range of algorithms, abstractions, and overall program structures. Anyone can design an expressive, but complex, language by “feature stacking”; e.g., PL/I, ADA, and CommonLisp. Anyone can design a simple Turing-complete language; e.g., (most) assembly languages and BASIC, which obviously lack expressiveness. Any language design is a compromise between expressiveness and simplicity.
- An *efficient* language allows algorithms to be implemented with their theoretical time and space requirements, with the “usual” constant factor for

> the computer in question. Efficiency is often best achieved by directing implementation efforts to a small number of fairly low-level basic constructs, with simple interactions. However, efficiency can also be achieved by providing very high-level constructs that have efficient implementations.

Where to strike the balance between simplicity, expressiveness, and efficiency, is determined by the intended application area. In this dissertation, some simplicity and efficiency of the programming language will be traded for the expressiveness required by knowledge information processing applications.

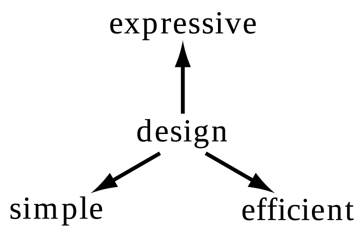

Figure 1.1. The balance between design principles

Another basic principle, still not as widely appreciated as one would expect considering that it has been well known since the early 1960s, is that the appearance (*syntax*) and the intended behaviour (*semantics*) of programs in a programming language should be *formally defined*. Some people in the business even seem to believe that formalism is an “egghead” notion, of little or no consequence in their daily life. Nevertheless, they can spend a large part of their life wrestling with the most rigid and formal of tools: computers and programming languages. Many of their problems, e.g., unexpected interactions between language features, stem from the fact that the definitions of these languages were not framed in a suitable formalism, which could have allowed an early pinpointing of difficulties in the design process. A suitable formalism allows the language designer to be precise, and thereby avoid inconsistency and ambiguity; it then allows the implementor to know exactly which behaviour to implement for any conceivable program; and it allows the programmer to know exactly what to expect in any situation.

Sometimes, simple formalisms can be used for the core of a language, e.g., SLD-resolution for Prolog, but do no longer suffice for the augmentations required to achieve a full-fledged programming language. The proper way to formalise Prolog behaviour has been debated for the better part of a decade. Indeed, there are many different approaches to defining programming languages.

A basic distinction is whether to have an *operational* or a *denotational* semantics. One tells us what programs do in a way that allows a corresponding implementation on a computer, the other what programs “mean” in terms of mathematical objects such as functions and relations. Having both is of course possible, but one should be regarded as the defining semantics.

Both have their advantages. A simple denotational semantics means that programs are simple from a mathematical point of view, which might simplify reasoning about a program. A simple operational semantics means that there is a simple basic implementation scheme, which might simplify predicting the actual behaviour of a program. In our experience, an operational semantics is to be preferred as the defining semantics, since it serves as a source of integrity for the language as a tool for programming. To also require properties such as soundness and completeness with respect to a particular semantics may be an additional constraint on the design.

Among operational semantics, we will make a distinction between *computation models* and *execution models*. A computation model defines computation states, transitions between computation states, and computations (as possibly nonterminating sequences of computation states derived by consecutive transitions). An execution model is a restriction of a computation model that should be capable of producing at least one, but not necessarily all, of the possible computations starting with a given state.

The *defining model* should be a computation model, which defines all legal computations for a given program. Associated with implementations are corresponding execution models, where particular choices may be made where the computation model offers a degree of freedom. For example, where the computation model may allow an arbitrary execution order (e.g., for the arguments to a function in a functional language), a particular order may be chosen in a sequential implementation, or that freedom may be exploited in a parallel implementation.

1.3 PARALLELISM

Programs should provide a potential for *parallel execution* that can be exploited automatically on multiprocessor computer systems, as argued in our initial discussion. Before we continue, here is some additional (informal) terminology that will be used throughout this dissertation:

- *Parallelism* will mean making use of several of the processors on a multiprocessor computer simultaneously, with the aim to make a program run faster, but without otherwise changing its observable behaviour.
- *Concurrency* will mean having components of a program that can proceed independently, with intermittent communication and synchronisation. Concurrency can often be exploited as parallelism.
- *Threading* will mean having concurrent components of a program that are given the opportunity to proceed at a certain well-defined pace, e.g., time-shared processes in an operating system. Threads can be used to satisfy real-time requirements.

Parallelism pertains to the implementation of a language, whereas concurrency and threading pertain to its definition.

The execution of a concurrent program becomes *nondeterministic* in that different components may make their move first, leading to a “random” choice of the next computation state. Such nondeterminism will usually not be visible in sequential implementations, but may be in parallel implementations due to the varying speed of the asynchronously working processors. Randomness is clearly in conflict with predictability. However, if the difference between alternative computation states is always such that they can “catch up” with one another in equivalent future computation states, then the choice doesn't matter. The same job will be done. Formally, this property is known as *confluence*.

Ideally, a concurrent language should have a confluent computation model, but there are reasons why this may not be possible. For example, in most many-to-one communication situations, messages have to be received from senders in an order that cannot be determined in advance. However, it is possible to localise such nondeterminism, which will be called *don't care nondeterminism*, to a few constructs in the language. If they are not used in a program, confluence should be guaranteed.

1.4 INTEROPERABILITY

A programming language and its implementation must be able to coexist and interact with their environment: the user, the hardware, the operating system, existing applications, and other computer systems. This property is often referred to as *interoperability*.

Interoperability can and should influence the language design process. Many new languages fail to reach a wider audience because of poor support of interoperability in the language and its programming system.

A program communicates with the user, other computers, and other programs through devices, secondary storage, shared memory, and process communication. A program communicates with subprograms written in other languages through procedure calls, potentially with updating of shared data, and message passing to objects. Such communication should fit into the computational framework of the new language, without ad hoc extensions, and without too roundabout modes of expression.

Many languages, typically found among the so called declarative languages in the functional and logical paradigms, lack proper models of state, thus crippling their interoperability with conventional languages, which rely on assignment to update data structures in a state.

1.5 PROGRAMMING PARADIGMS

Finally, let us attempt to characterise, very briefly, a number of basic language types that also stand for corresponding *programming paradigms*.

- An *imperative* language has the ability to mutate data objects. Given an arbitrary imperative language, its computation model may be understood as a transition relation, but no other reading is guaranteed to be possible.

- An *object-based* language associates operations with the types of data objects. In a *procedure-based* language, the definition of a generic operation has knowledge of all the data types to which it is applicable. In an object-based language, the operation may be defined for each data type separately.
- An *object-oriented* language is object-based. It also allows new data types to be defined based on existing types in such a way that all or selected parts of the operations on the existing type are *inherited* by the new type.
- A *functional* language allows programs to be read as function definitions, and the execution of a program as the evaluation of an applicative expression.
- A *logical* language allows programs to be read as defining predicates, and the execution of a program as finding a proof for a given statement.
- A *process-based* language provides notions of processes and message-passing between processes.

There is no conflict between these language types. Indeed, a language may be imperative, object-based, functional, logical, and process-based. Of course, a language representing a paradigm may sometimes be *pure* in that it will only admit constructs with the characteristic properties, e.g., a pure functional language will not admit operations that mutate data objects. However, single-paradigm languages are often crippled in their ability to interoperate with a multiparadigm environment.

Many (or most) languages provide elements of several paradigms, sometimes for efficiency, sometimes for expressiveness, and sometimes for all sorts of reasons. The problem is that the result may no longer be simple.

This dissertation describes a language that supports all the above paradigms, while still being surprisingly simple.

CHAPTER 2

LANGUAGE OVERVIEW

Although this is not an AKL textbook, an attempt has been made to provide a language introduction which is as readable as possible. Probably, the results in later chapters will be better appreciated given a firm basic understanding of AKL. Chapter 2, Language Overview, and Chapter 3, Programming Paradigms, form a self-contained informal introduction to AKL and its various aspects. A formal computation model is presented in Chapter 4.

2.1 THE DESIGN

The AGENTS Kernel Language[1], AKL, was conceived with *knowledge information processing* applications in mind, which motivates its particular compromise between simplicity, expressiveness, and efficiency. In addition, with a view to the next generation of multiprocessor computers, AKL provides a large potential for *parallel execution* that can be exploited automatically.

Among the most promising programming languages for knowledge information processing are the *logic programming* languages.

- Prolog and the related constraint logic programming languages have been commercially available for a number of years, and have been used, to great advantage, in a number of advanced industrial applications.
- The concurrent logic programming languages, e.g., Concurrent Prolog, PARLOG, GHC, and Strand, the last of which is commercially available, were designed to allow easy exploitation of multiprocessor computers.

However, both have shortcomings: Prolog lacks the expressiveness of the process-oriented framework, has no appropriate model of state and change, and has many properties that make parallelisation of programs more difficult than

[1] AKL was previously an acronym for the Andorra Kernel Language, but is now regarded as the kernel language of the AGENTS programming system.

necessary. The concurrent logic programming languages lack the expressiveness of don't know nondeterminism, which is useful for many knowledge information processing applications.

AKL integrates, in a coherent and uniform way, the don't know nondeterministic capabilities of Prolog with the process describing capabilities of languages such as GHC, thus remedying the shortcomings of both.

The rest of this chapter is an introduction to AKL.

First, the basic concepts of concurrent constraint programming are presented. Then, based on these concepts, the elements of AKL are introduced step by step, with examples illustrating the additional expressive power provided by each of the language constructs.

As a necessary complement to this fairly informal introduction, Chapter 4 provides a formal definition of the language and its computation model.

2.2 CONCURRENT CONSTRAINT PROGRAMMING

AKL is based on the concept of *concurrent constraint programming*, a paradigm distinguished by its elegant notions of communication and synchronisation based on constraints [Saraswat 1989].

In a concurrent constraint programming language, a computation state consists of a group of *agents* and a *store* that they share. Agents may add pieces of information to the store, an operation called *telling*, and may also wait for the presence in the store of pieces of information, an operation called *asking*.

The information in the store is expressed in terms of *constraints*, which are statements in some constraint language, usually based on first-order logic, e.g.,

$X < 1$, $Y = Z + X$, $W = [a, b, c]$, …

If telling makes a store inconsistent, the computation fails (more on this later). Asking a constraint means waiting until the asked constraint either is *entailed* by (follows logically from) the information accumulated in the store or is *disentailed* by (the negation follows logically from) the same information. In other words, no action is taken until it has been established that the asked constraint is true or false. For example, $X < 1$ is obviously entailed by $X = 0$ and disentailed by $X = 1$.

Constraints restrict the range of possible values of *variables* that are shared between agents. A variable may be thought of as a container. Whereas variables in conventional languages hold single values, variables in concurrent constraint programming languages may be thought of as holding the (possibly infinite) set of values consistent with the constraints currently in the store. This *extensional* view may be complemented by an *intensional* view, in which each variable is thought of as holding the constraints which restrict it. This latter view is often more useful as a mental model.