Software Development Environment Based on Object-Oriented and Logic Programming Paradigms

Khaled Fouad Sayed Shaalan

A Dissertation submitted in partial fulfillment of the
requirement for the Degree of Doctor of Philosophy at
Department of Computer Science
The Institute of Statistical Studies and Research (ISSR)
Cairo University, Egypt

1995

Supervised by

Prof. Salwa El Gamal
Dept. of Computer,
Science, ISSR,
Egypt

Prof. Ahmed Rafea
Dept. of Computer
Science, ISSR,
Egypt

Prof. Seif Haridi
Swedish Institute
of Computer Science,
Sweden

Abstract

Software development environments are essential in today's world of growing software systems. Software development environment increases the productivity of software by providing better languages and better tools.

Programming languages naturally play an essential role in the software development process. Finding more powerful and better suited language has been the aim of language designers ever since the dawn of computer programming. For instance, the most recent research in concurrent logic programming paradigm is directed towards concurrent constraints framework where the development of the AKL (Agents Kernel Language) language is significant. AKL+, the main contribution of this thesis, is a natural development which is derived from the fusion of concurrent constraint programming and object-oriented programming paradigms. The result is more than a sum of its parts since many of the inadequacies of one programming paradigm are compensated for by features of the other. The development of large concurrent object-oriented programs puts a new demand for tools that keep software components up to date. This has lead us to develop a general incremental compilation facility which is needed for comfortable processing of large programs.

This thesis has presented the background to, and motivation for, the design of AKL+, and described the language. The schemes for developing an efficient implementation have been discussed. Algorithms are developed for the general incremental compilation facility and multiple inheritance. The semantics of AKL+ has been described in terms of translation to AKL. A set of examples has been implemented to demonstrate the applicability of AKL+ both as a modeling language and as an implementation language. The AKL+ language and our incremental compilation facility have been implemented on Unix-based workstations and they are parts of the official release of the AKL system, AGENTS, developed at SICS (Swedish Institute of Computer Science).

⋆ ⋆ ★ ⋆ ⋆

Acknowledgments

I am greatly indebted to my supervisors: Prof. Dr. Ahmed Rafea, Prof. Dr. Salwa El-Gamal, and Prof. Dr. Seif Haridi for their help, encouragement, valuable discussions, and for their comments on this thesis.

I would like to thank the Central Lab for Agriculture Expert System (CLAES) leaded by Prof. Dr. Ahmed Rafea, the place in which my experience in object-oriented programming and logic for real life systems are initiated and has been grown up.

I would like to thank the Egyptian Government for offering the financial support that allwed my study in Sweden to undertake the research reported in this thesis.

Thanks are due to the entire members of Swedish Institute of Computer Science (SICS) for offering me the facilities, resources, and support that were needed for my research. In particular, I want to acknowledge the entire members of the PS group leaded by Dr. Sverker Janson. Also, a very special thanks to Prof. Khayri Ali who helped me and provided me with very valuable support during the critical time of the research. Finally, I would like to thank the entire technical support staff for keeping computer facilities alive and up-to-date.

This list of persons could be extended much further, therefore I would like to thank all those not mentioned above, who have helped and supported me.

Table of Contents

1 Introduction **1**
- 1.1 Software Development Environments . . . 1
- 1.2 Motivations . . . 2
- 1.3 Parts of Software Development Environments Treated in this Thesis . . . 4
- 1.4 Main Contributions . . . 5
- 1.5 Structure of the Thesis . . . 5

2 Previous and Related Work on Object-Oriented Programming Languages **9**
- 2.1 Conventional Languages . . . 9
- 2.2 Actor Based Languages . . . 14
- 2.3 Logic Based Languages . . . 17
- 2.4 AKL+ . . . 28
- 2.5 Summary . . . 32

3 AKL **35**
- 3.1 Concurrent Constraint Programming . . . 35
- 3.2 Language Design . . . 37
  - 3.2.1 Basic Concepts . . . 37
    - 3.2.1.1 Don't know nondeterminism . . . 41
    - 3.2.1.2 Don't care nondeterminism . . . 42
    - 3.2.1.3 Encapsulated computations . . . 43
  - 3.2.2 Definitions . . . 45
- 3.3 Basic Object-Oriented Style . . . 45
  - 3.3.1 Objects . . . 46
  - 3.3.2 Ports for Objects . . . 49
  - 3.3.3 Data Objects . . . 52
- 3.4 Syntactic Sugar . . . 53
- 3.5 Summary . . . 56

4 AKL+ **57**
- 4.1 Classes . . . 57
  - 4.1.1 Defining Classes . . . 58
  - 4.1.2 Attribute Methods . . . 64
  - 4.1.3 The Default Behavior . . . 66
  - 4.1.4 The Class Membership Method . . . 67
  - 4.1.5 Method dispatcher . . . 68
  - 4.1.6 Generic Classes . . . 69
  - 4.1.7 Metaclasses . . . 71
- 4.2 Objects . . . 72
  - 4.2.1 Message Sending vs Method Call . . . 73

4.2.2 Creating and Destroying Objects . . . 75
4.2.3 Sharing Objects . . . 76
4.2.4 Synchronization . . . 77
4.3 Standard Classes . . . 79
4.3.1 State Representation Classes . . . 79
4.3.2 Object type Classes . . . 82
4.4 Synchronization Constraints . . . 86
4.4.1 Accept Method Set . . . 87
4.4.2 Synchronizers . . . 88
4.4.3 Transitions . . . 90
4.5 Inheritance . . . 92
4.5.1 Class Inheritance . . . 92
4.5.2 Differential Inheritance . . . 94
4.5.3 Synchronization Constraints and Inheritance . . . 96
4.5.4 Multiple Inheritance . . . 100
4.6 Summary . . . 102

5 Definition of AKL+ 105
5.1 Definitions and Programs . . . 105
5.2 A Simple Example . . . 106
5.3 Expansion to AKL Code . . . 110
5.4 Summary . . . 118

6 Implementation 119
6.1 The Incremental Compilation Facility . . . 119
6.2 Applying the Inheritance Mechanism . . . 123
6.3 Expansion to AKL Code . . . 127
6.3.1 Examples of a Class Expansion . . . 127
6.3.2 The Class Expansion . . . 132
6.3.3 Efficiency . . . 141
6.4 Expansion to AKL Abstract Machine(PAM) . . . 142
6.5 The AKL+ Run-time . . . 143
6.6 Summary . . . 144

7 Programming Examples 145
7.1 A Car Washer Simulation . . . 145
7.2 A Moving Point Constraints . . . 155
7.3 A Car Maintenance Fault diagnosis . . . 160
7.4 A Bounded Buffer State Modification . . . 169
7.5 Summary . . . 176

8 Conclusion and Future Work 179
8.1 Conclusion . . . 179
8.2 Future Work . . . 182

A Interactive Incremental Compilation Commands 185

List of Figures

1.1 Chapter dependencies 7

3.1 Agents interacting with a constraint store 36
3.2 Abstract syntax of AKL program 46
3.3 An object consuming a list of messages 47

4.1 The AKL+ Object Oriented System 58
4.2 base-class reference 63
4.3 self-reference 64
4.4 The passing of parameters to a parameterized class 71
4.5 An object 73
4.6 The method execution 74
4.7 Method call vs self-reference 74
4.8 Creating an object 75
4.9 An example of object sharing 76
4.10 A conceptual view of an initial class 80
4.11 The structure of a synchronized object 84
4.12 Message acceptance of a bounded buffer 86
4.13 An example of method overriding 94
4.14 An example of multiple inheritance 101
4.15 An example of linearized multiple inheritance 102

5.1 Abstract syntax of AKL+ program 107
5.2 State threading through the body of a method 111

6.1 Compiler and run-time expansions of AKL 120
6.2 Finding the source files 121

7.1 Existing interaction in the Car Washer problem 146
7.2 The moving point class hierarchy 156
7.3 A car component hierarchy 161
7.4 Conceptual illustration of the state modification of a bounded buffer object 169

1

Introduction

Software development environments are essential in today's world of growing software systems. Software development environment increases the productivity of software by providing better languages and better tools. In this chapter some background on software development environments are first presented. The motivations and the subject of this thesis are then discussed. The parts of software development environments treated in this thesis follows. Finally, the main contributions and the structure of this thesis are summarized.

1.1 Software Development Environments

As ambitions show software applications get more and more complicated and thus harder to develop, maintain and understand. The need for increased efficiency in developing software has long been recognized and is an important aim in Software Development Environment research. The Software Development Environments include languages with expressive power desirable for application programming, the tools needed to develop programs (editors, compilers, debugger, browsers, etc.) and also include the tools needed to handle a large complex system (incremental compilation, interface design, etc.).

The wide acceptance of object-orientation has put new demands on the environments we use. The demands put on modern software environments are summarized by highlighting three important aspects [87]: interactiveness, integration, and incrementality.

Interactiveness. Aspects of interactiveness come up in a wide range of situations. Examples arise during editing, code generation and updating the executing environment; other examples are the availability to change a running program and the development and extension of the environment itself.

Integration. Aspects of integration apply to the user interface design, how functionality like editing, how programming in-the-large [116] and version control interact with the programming in the large support, and how language dependence and the environment itself are developed and customized.

Incrementality. Incremental algorithms are often recognized as essential in order to achieve quick enough turnaround between different activities. This is indeed important since programs will inevitably grow large enough for "batch" algorithms

to be too slow for high demands on response time. Incrementality is, however, not only a quantitative aspect, but also a qualitative aspect. Many of the demands on tight integration can only be achieved by incremental algorithms that keep program representation updated all the times.

Examples of existing software development environments are:

- Many Lisp environments fits into this category, and we point at Interlisp system as an example [140]. These environments provide integrated environments with quick turnaround between editing and execution. Often this is achieved by interpreting the program and having untyped language which puts little burden on the system in terms of checking. Also there is no need for time-consuming linking since the user's program is incrementally linked into the address space of the development environment.
- The Smalltalk environment [56] provides a graphical user interface, a browser, for navigating among classes and operations, but the program editor is purely textual. Quick turnaround time is achieved by translating a piece at a time (a procedure, a message) and again using linking in the same address space.

One important goal in these environments has been to achieve fast turnaround between editing and execution, which is an important aspect in order to support interactive development style of programming. However, the techniques used to develop these environments are not always generally applicable, being tailored to one specific language. Development of environments are very large tasks and more generally applicable techniques are needed to make integrated environments common for many languages.

1.2 Motivations

Object-Oriented programming is attracting a significant amount of attention from researchers and software developers alike. A major benefit from object-orientation is to reduce software complexity through its ability to increase the reuse of software components [25, 37]. In object-oriented programming, the program execution is viewed as a physical model that simulates the behavior of some real or imaginary part of the world. Programs/designs constructed in the object-oriented style are by most people considered easier to understand and change than traditional programs. In addition, the abstraction mechanisms of object-oriented languages are powerful means to encapsulate and isolate design and execution information [27].

A logic language provides mathematically based framework for symbolic evaluation and automatic deduction. Logic languages has been proved adequate to "declarative programming", deductive database implementation, expert systems, and more

generally as general-purpose tools for non-deterministic programming. Prolog is the most popular logic programming language. It is a sequential language designed to run efficiently on a von Neumann machine by exploiting its ability to perform efficient stack management. Nevertheless, Prolog cannot express concurrency [128]. For this purpose, languages of a new type have been defined such as *concurrent logic programming languages*.

Recently, some work on concurrent logic programming languages are extended with the notion of *constraints*. This evolution is clearly visible in AKL which is, first and foremost, a language incorporating ideas from constraints and concurrent logic programming. The view of computation in this setting is interesting in the context of concurrency because of the ability to represent and manipulate partial information about the domain of discourse, in the context of concurrency because of the use of constraints for communication and control, and in the context of AI because of the availability of simple yet powerful mechanisms for controlling inference [123].

There are many advantages in combining concurrent constraint logic programming and object-oriented programming paradigms, because the resulting language overcome most limitations of both approaches, taken singularly. For instance, concurrent constraint logic programming benefits because an object-oriented style provides good notations for expressing abstractions that helps in the comprehension of programs. The lack of structuring mechanisms for large applications can be removed using a class construct. Also, the problem of being able to reuse software components can be removed using inheritance. Conversely, a constraint programming provides an excellent computational support with a basic, but not enough, object-oriented style. The notion of self-contained independent object is very close to that of concurrent process (dynamically computing agent) or *actor*. Furthermore, the use of constraints for communication and control maps well onto communication by message passing between objects.

The development of large concurrent object-oriented programs puts a new demand for tools that keep software components up to date which is needed for comfortable processing of large programs.

The main motivations for investigating object-oriented programming in concurrent constraint logic programming are:

1. The design of a language that has the concurrent computation ability, and the knowledge representation and problem solving of concurrent constraint logic programming enriched with the program and knowledge structuring capabilities of object-oriented programming.

2. Supporting all features that are needed for any object-oriented application. These features will increase the generality and the reuse of software components such as generic classes, to support fine-grained objects, provide for higher-order and data-driven programming techniques, and synchronization,

as well as mechanisms to handle problems arise due to multiple inheritance and/or synchronization.

3. The design and implementation of a general incremental compilation facility. A primarily feature of this tool is that the work needed after a change is proportional to the size of the change rather than to the size of the program.

4. Our belief in the possibility of an efficient implementation.

1.3 Parts of Software Development Environments Treated in this Thesis

Software productivity is essential in today's world of growing software systems. Productivity can be increased by improving the situation in at least two different ways: better languages and better tools.

Better languages. Programming languages naturally play an essential role in the software development process. Finding more powerful and better suited language has been the aim of language designers ever since the dawn of computer programming. AKL+, the main contribution of this thesis, is evolved from the fusion of the most recent research in concurrent constraint programming and object-oriented programming paradigms supporting all features that are needed for any object-oriented application.

Better tools. Constructing programs involves the use of many different tools: editors, compilers, linkers, etc. By providing better environments much of these tools could be automated, freeing the developer's attention for important tasks. The *incremental compilation facility*, the supported tool in our environment, is an essential part of the software development environment with the aim of integrating support for interactive incremental compilation facility for object-oriented systems. The advantages of this tool are twofold. First, a quantitative effect is obtained, by decreasing the compilation time. This allows the programmer to switch rapidly between program editing and running the program. The second effect is qualitative. If the compilation is made with small increments after each single edit operation, and each such small increment is computed so fast that the programmer does not notice, the illusion can be maintained of having which is compiled at all times. The programmer does not have to issue different commands since the compilation and the incremental loading is always done automatically and all the dependent files made up to date.

The AKL+ language and our incremental compilation facility have been implemented on Unix-based workstations and they are parts of the official release of the AKL system developed at SICS (Swedish Institute of Computer Science). The AKL

system, AGENTS, is available from SICS for research and educational purposes (contact **agent-request@sics.se**).

1.4 Main Contributions

In short, our main original contributions are as follows:

1. A new language called AKL+ which can naturally express a large class of problems than most languages, simply because it combines two extremely powerful paradigms: concurrent constraint logic programming and object-oriented programming.
2. The ability to utilize AKL+ as a higher level AKL by virtue of the way that its features subsumes AKL programming style in a less verbose, proper linguistic support, and more intuitive manner.
3. The development of a class library for most common operations which fulfills the user's computational needs as it provides him with a simpler, and easier-to-use computing environment.
4. The investigation of application areas which benefit from the concurrent constraint logic programming and object-oriented features.
5. The definition of the language semantics in terms of translation to AKL which has been proved formally.
6. Developing techniques of an efficient implementation of the language.
7. An algorithm for a general incremental compilation facility.
8. An algorithm that computes multiple inheritance.

1.5 Structure of the Thesis

The thesis is structured as follows. The present chapter aims at presenting some background material and establish the problem area.

Chapter 2 gives a survey to the most popular object-oriented programming languages. In our presentation we try to group the related languages into separated sections. We present the main features of our language and compare it with the most related ones.

Although AKL+ is a language in its own right, many of the programming techniques used are those of AKL, the language on which it is based, and many AKL programs

can be incorporated into an AKL+ program without change. Chapter 3 gives a brief and informal summary of AKL and reviews the basic techniques that allows us to do object-oriented programming in AKL. We also give a short background of concurrent constraint programming.

Chapters 4 and 5 looks at the basic elements of the AKL+ language and its definition. Chapter 4 informally presents AKL+ and its computation model. We describe classes, including standard classes, objects, message passing, synchronization, and inheritance. Our approach to handle multiple inheritance, differential inheritance, and inheritance anomaly is discussed. Several examples show the main features of AKL+ as well as a number of programming techniques. Chapter 5 presents the syntax and semantics of AKL+. We describe the semantics in terms of translation to AKL. The translation is based on a source-to-source transformations. For each kind of transformation, we give a definition of a rewrite rule.

Chapter 6 presents the implementation aspects of the general incremental compilation facility and our language. The text in this chapter is kept on a higher level of abstraction in order to gain the applicability to a wide range of object-oriented languages. We give an algorithm for a general incremental compilation facility. The algorithm specifies dependencies as functions to bring a target file up to date. We give an algorithm for the dependency that computes multiple inheritance. We describe the dependency for expanding a class definition into AKL and show how we can implement AKL+ as an efficient programming language. The AKL+ run-time module that allows us to achieve a run-time uniform message sending is described.

Chapter 4 includes a number of small examples, but to bring out the advantages of combining the object-oriented and concurrent constraint logic programming paradigms it is necessary to look at some longer examples. This is done in Chapter 7. In this chapter we explore small but complete AKL+ examples, from its conception to realization. These problems, are general enough for the purpose of demonstrating the applicability of AKL+ both as a modeling language and as a programming language.

chapter 8 concludes the thesis and gives directions for future work.

Appendix A contains the description of the interactive incremental compilation commands.

See Fig. 1.1 for a graphical overview of the thesis.

⋆ ⋆ ★ ⋆ ⋆

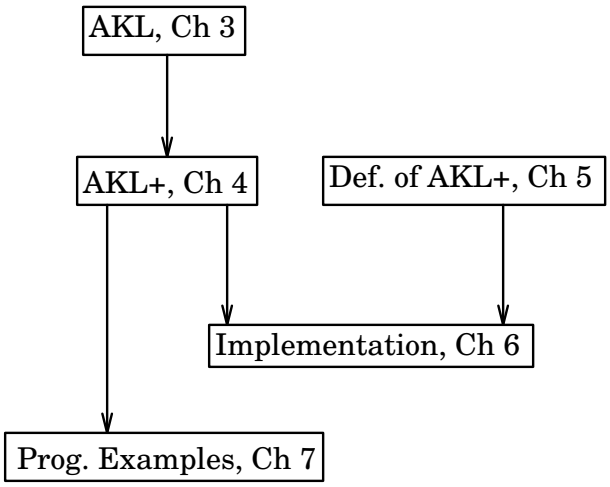

Figure 1.1: Chapter dependencies

2

Previous and Related Work on Object-Oriented Programming Languages

In this chapter, we will survey the most popular object-oriented programming (OOP) languages. In our presentation we try to group the related languages into separated sections. Section 2.2 presents actor based languages and Section 2.3 presents logic based languages. In Section 2.1, we present the conventional and other object-oriented programming languages which cannot be classified as either actor based or logic based languages. In section 2.4, we present the main features of our language and compare it with the most related ones.

2.1 Conventional Languages

Simula [38] is the first object-oriented language that took the block concept of Algol one step further and introduced the concept of an *Object*. It was mainly intended as a simulation language. Conceptually, an object contained both data and the operations, called *methods*, that manipulate its data. However, in allowing direct access to attributes, it failed to provide *encapsulation*. Simula also incorporated the notion of *classes*, which are used to describe the structure and behavior of a set of objects. Class inheritance where classes can be organized into hierarchies is also supported. Simula is a strongly typed language. Simula supports sequential execution as well as coroutining. Coroutining is different from sequential execution since a coroutine may temporary suspend execution ("detach") and "resume" later. This means that coroutines are useful to support program executions with multiple action sequences. Simula laid the foundation of object-oriented languages and some of the object-oriented terminology.

Smalltalk [56, 119, 91] was the first and popular object-oriented language, developed at Xerox PARC, and its success engendered many other object-oriented languages. Smalltalk is not only a language but also a development environment. All aspects of the Smalltalk language system are available through an on-line interpreter and class browser. The language is untyped and every thing is an object, including classes. The individual object described by a class is called *instance*. Smalltalk supports a standard class library–a predefined set of reusable software components, which can be subclassed and extended incrementally. Smalltalk allows inheritance

in the form of single inheritance, where each class can have at most one superclass. Smalltalk includes a *FORK* construct which allows processes to run concurrently. However, the processes are actually run sequentially, starting with the first process listed. This first process runs until completed or until it cannot continue, at which time the second process begins to execute. This second process also runs until completed or until it cannot continue. Then, if there is a third process it begins to run, otherwise the first process is again started. This continues until all processes have completed. Semaphores [129] are used for process synchronization. A *YIELD* statement is also provided which allows processes to yield the processor at any time. A process can be made to yield the processor between any pair of statements. Using a random number generator to calculate the probability of yielding the processor can lead to a nondeterministic system in which each Smalltalk statement is considered to be atomic (uninterruptible) [135]. However, this is probably not the best way to implement a concurrent system.

Lisp-Based Languages. The Common Lisp Object System (CLOS) [21, 82], CommonLoops [20], and Flavors [108] are the object-oriented extension of Lisp. The most notable OOP language of this family is CLOS, an object-oriented extension of Common Lisp [134]. In CLOS, every data object, including atoms and lists of Lisp, is a member of a class. Methods for Lisp primitives belong to an inheritance structure. CLOS does not have a standard class library. CLOS Contains facilities for multiple inheritance and applies "linearization", see [133], for resolving the ambiguity resulting from inherited features with the same name. New Classes can be defined, and methods can be added to classes dynamically. These features are a standard part of the language, documented as "meta-object protocol" [85]. Like Common Lisp, CLOS is a weakly-typed language. The concept of encapsulation is not enforced by the CLOS since there is nothing to prevent the code of one class from directly accessing the implementation details of another class. A Lisp-based OOP language does not support any constructs for handling concurrency.

However, as Lisp is a form of functional programming [60] (which is based on lambda calculus), some form of concurrency is possible. In pure functional programming, a program may be expressed as a single function call, with the arguments to the function themselves being function calls; the arguments to these functions can in turn be function calls, etc. Since the value returned by a pure function is determined solely by the arguments passed to it, implementations can be devised which allows for all of the arguments which are function calls to be executed in parallel [112]. The arguments of these function calls which are themselves functions can then be executed in parallel, and so on. This is sometimes referred to as divide and conquer in that a program is divided up into concurrent subprograms (the arguments which are function calls) which can then be conquered (solved) by again using the divide and conquer scheme [118].

Ada [144, 22, 17] is not generally considered to be object-oriented as it does not support inheritance; rather, it is more appropriately classified as object-based language. Ada does, however, support concurrency in the form of tasking and message passing with rendezvous for process interaction.

Eiffel is strongly typed object-oriented language [106]. Programs consist of class declarations that include methods. Multiple inheritance and parameterized classes (*generics*) are supported. A modest class library is provided. The focal point of Eiffel is the class declaration, which lists attributes and operations. Eiffel provides uniform access to both attributes and operations. Eiffel does not treat classes or operations as first-class objects. Eiffel does not contain any form of concurrency.

C-Based Languages. C++ [138, 137, 139, 154] and Objective-C [120] languages are the object-oriented extension of the C language [83]. The most notable OOP language of this family is C++. C++ is a hybrid language, in which some entities are objects and some are not. It is a strongly-typed language. It was originally implemented as a preprocessor. Unlike several other object-oriented languages, C++ does not contain a standard class library. Unfortunately, because C++ provides no guidelines for library organization, different libraries may be incompatible. C++ contains facilities for multiple inheritance and run-time method resolution, but a C++ data structure is not automatically object-oriented. Method resolution and the ability to override an operation in a subclass are only available if the operation is declared *virtual* in the superclass. C++ contains good facilities for specifying access to attributes and operations of a class. Access may be permitted by methods of any class (*public*), restricted to methods of subclasses of the class (*protected*), or restricted to direct methods of the class (*private*). In addition, "spot" access can be given to a particular class or function using the *friend* declaration.

LogiC++ [155] integrates logic and object-oriented programming. The language is designed primarily based on C++. A program in LogiC++ has a very strong resemblance to C++ programs. However, in a LogiC++ program, the keyword "methods" is used to indicate that the following methods would be defined by Horn clauses as in Prolog programs. The rest of the program structure is basically the same as any C++ program. The compiler takes a LogiC++ program as input and produces an equivalent C++ program as the output. In particular, the compiler takes the methods as input and produces C++ functions as output. The C++ program can then be compiled by a C++ compiler.

BETA [89, 95] is a language designed in the Simula tradition. It is untyped language as Smalltalk. BETA replaces classes, procedures, functions and types by a single abstraction mechanism called "pattern". It has a linguistic support for nesting and block structure. The language allows single inheritance. Object may execute their actions sequentially, "alternating" with other objects (coroutining), or concurrently with other objects. The alternation between coroutines may be deterministic in the sense that the sequencing is decided by the object itself. The shifts between corou-

tines may be triggered by events performed by other concurrent objects, leading to nondeterministic alternation. The basic mechanism in BETA for synchronizing concurrent objects is semaphore. Moreover, the library of the BETA languages includes also monitor [67] for guaranteeing exclusive access to an object and rendezvous mechanism for handling direct communication between objects [144].

ConcurrentSmalltalk [159, 160, 161, 39] is based on Smalltalk-80 [56]. The significant extensions were made: per-instance lock variables, asynchronous message passing, and distributed objects. ConcurrentSmalltalk allows several methods of an object to be active at one time. Actually, an object can respond to only one message at a time; however, a new context of the object can be allocated (activated) which uses the same set of variables as the original object. The lock variables are used in a fairly straight-forward way to control interference between methods which run concurrently. However, locking is only allowed on instance variables. A distributed object consists of one or more constituent objects. Each of these objects has the same behavior, i.e. the same set of methods available, but they may have different local data. An object which sends a message to a distributed object has no control over which constituent object handles the message. ConcurrentSmalltalk supports two distinct categories for objects: "atomic" objects and "non-atomic" objects which represent different levels of abstractions. The atomic objects act as serializers; messages to such objects are processed sequentially. The non-atomic objects are compatible with objects in Smalltalk; processing of invocation messages to such objects is concurrent.

$\mathcal{A}'\mathcal{UM}$ [162, 163] is a concurrent programming language which can be characterized by its stream-based computation model and by its relational representation. A stream is different from a thread in that a thread is a sequential set of actions (which may be performed concurrently with other threads), while a stream is a potentially concurrent set of actions (which may also be performed concurrently with other stream.) Stream computation may be explained using three terms: "inlet", "outlet", and "nil". These are all defined from the object's point of view. An inlet is where a stream may come into an object and an outlet is where a stream may leave an object. When an outlet is closed, it is referred to as nil. Messages from various streams may be *merged* together nondeterministically, and messages from one stream may be *appended* to messages from another stream. Messages become stream connector as the stream flows from one object to another. $\mathcal{A}'\mathcal{UM}$ consists only of objects which communicate with one another via streams. $\mathcal{A}'\mathcal{UM}$ supports multiple class inheritance.

Blaze2 [103] is designed to allow multiple threads within an object, i.e. both concurrency between objects and within objects is supported. Methods are "serial" by default, but may be explicitly declared to be "parallel". Both serial and parallel methods may be declared for each class. When an object is executing a serial method, no other methods may be active. Any number of parallel methods may

be executed concurrently as long as no serial method is currently active. Both read locks and exclusive (write) locks are supported. Blaze2 does not support any form of inheritance, and is therefore an object-based language, see [151, 152, 153] for the classification of languages.

CLIX [68] incorporates the notion of objects with communications as found in distributed systems. That is, the underlying communication system is modeled as a mail system. The basic elements in CLIX are objects and communications. An objects is similar to the notion of a process. Objects execute asynchronously and communicate via message passing, although synchronous communication are also supported. Additionally, messages may be forwarded from one object to another-if the message was originally a synchronous (blocking) one, then its reply address is provided to the object that the message is forwarded to. This forwarding of messages implements a form of delegation (inheritance is not supported). Messages are guaranteed to be delivered, and message ordering is preserved for all messages sent by one object to another.

Parallel Object-Oriented Language (POOL) [8, 9] is specifically designed to run on a specialized parallel architecture called Decentralized Object-Oriented Machine (DOOD) [14]. Concurrency is only supported between objects, i.e. concurrent methods within a single object are not allowed. An object in POOL cannot receive a message until it is ready to do so. It may indicate a readiness to receive only some subset of the messages which it is capable of responding to. Both synchronous and asynchronous message passing are supported. POOL is an object-based language.

PROtocol-constrained Concurrent Object Language (PROCOL) [23] is a C-Based object-oriented language. Concurrency is allowed between objects, but not within an object. Actions (methods) in PROCOL are atomic, i.e. they uninterruptible. Message passing is synchronous in that the sender must wait until the message has been received. The sending object does not wait, however, on the receiver to do any processing-as soon as the message is received a reply is sent back, so that both the sender and the receiver of the message are then executing concurrently. An object will accept messages only when it is ready to do so. It may use various guards to conditionally accept only certain kinds of messages from various objects. PROCOL is an object-based language.

SINA [142] is an OOP language for distributed and concurrent computing. It is based on the notion of prototypes [93]. SINA supports concurrency within objects. There are two kinds of objects in SINA, "passive" objects (instances of primitive types) and "active" objects (processes). Methods can process invocation requests either on a first-in-first-out basis or according to a priority scheme (as an integer-valued parameter). Every active object has a queue for buffering invocation request messages. In order to synchronize concurrent activities an object's interface can be in one of two states hold (messages are buffered in the queue) or accept (messages

can be processed). Moreover, an object operation can put in the accept or the hold state by executing the accept or hold operation, respectively.

Ellie [10, 11] is a fine-grained OOP language for parallel and concurrent computing. Ellie is a strongly typed language. Like Smalltalk, everything is treated as objects, e.g. classes, types, block, methods and regular objects. It is a prototypical language where objects can be organized into hierarchies. Multiple inheritance is supported. Synchronization is supported by means of interfaces that can change over time, i.e. an object is allowed to select between incoming requests as it pleases. An incoming request, that cannot be served because the accept interface is closed or the operation excluded, is delayed. When the interface is changed, the delayed requests are inspected in FIFO order to look for the first request that can be served, if any. If a request is found, it will be served immediately.

Orient84/K [69, 70] is a concurrent language which combines logic and OOP. Objects in Orient84/K are called "knowledge objects" (KO), which may execute concurrently. They consist of three parts: a behavior part, a knowledge part, and a monitor part. The behavior part is very similar to an atomic object in ConcurrentSmalltalk. It consists of a collection of instance variables and a set of methods. Both synchronous and asynchronous message passing, as well as future style messages of ABCL [156, 158], are supported. The knowledge part is essentially a Prolog-like collection of first-order predicate logic relations. The contents of the knowledge part can be examined and altered by the methods in the behavior part. The monitor part is used to achieve synchronization when required. Orient84/K supports multiple inheritance. Mutual execution are realized through changing the interface of the object.

2.2 Actor Based Languages

The actor model [65, 1, 2, 3, 66] is not object-oriented as it does not support inheritance or delegation; rather it is more appropriately considered to be object-based. However, it may serve as the foundation for an actor-based OOP language.

An actor is simply an object which responds to messages, but it can only respond to a single message at a time. A message queue is associated with each actor to hold incoming messages in the order of arrival. Two assumptions are inherent within the actor model [92]:(1) message are guaranteed to be received within a bounded time interval, and (2) an actor waiting to execute will eventually do so. An actor has one or more scripts which it can use in response to various messages (a script is essentially the equivalent of a method in more conventional OOP languages.)

Concurrency is supported in many ways. Messages can be sent to several actors, so that each is responding to its own message. An actor responds to exactly one

message then "dies". One of the things that it must do before it dies is to specify a replacement which will handle any additional messages sent to that actor (actually, the replacement will respond only to the next message, it too must specify a replacement which will then handle the next message, etc.) The message queue associated with the original actor is transferred to the replacement actor. This replacement may be another actor all together, or, more likely, a "clone" (possibly modified) of the original actor. This replacement may be specified at any time. If the creation of the replacement is the first thing that the actor does (or at least before it is finished responding to its message), then the replacement may begin to respond to the next message while the original actor is continuing to service the first message. In servicing a message, the actor may send messages to other actors.

Several languages have been designed around the actor model. Following is some of these languages.

Acore is a Lisp-based Language [96]. As such, it incorporates ideas from both the actor model and from lambda calculus. The functional style of lambda calculus easily expresses divide and conquer algorithms in Acore. As such, both objects which can execute scripts concurrently and functions which evaluate their arguments concurrently are supported.

ACT++ [76] is a proposed concurrent OOP language which combines both the actor model of computation with the class structure of C++. It is intended for real-time applications. The main intentions of the project are to: determine if the flexibility of the actor model can be maintained when an inheritance discipline is added, determine the usefulness of a concurrent OOP language in general and of the actor model specifically for real-time applications, and design the kernel of an operating system (called the REACT kernel) to meet the needs of real-time system.

Actalk [24] is an implementation of Actors in Smalltalk–80. It is designed as a minimal extension which preserves the original Smalltalk–80 language, i.e., it is designed as a superset of Smalltalk. Standard Smalltalk–80 objects are passive, but actors are active and autonomous. An actor is built from a standard Smalltalk–80 object by associating a process with it and by serializing the message that it receives into a queue. For this purpose, two classes, *Actor* and *ActorBehavior* are added to the class library. It is intended as a platform to express, classify, and test various computation and execution models of actor languages.

Actra [141] is a project investigating future generation industrial computer systems based on the actor model. Its primary goal is to provide an integrated, multi-user, multi-processor object-oriented development environment for use in medium and high performance industrial applications dealing with complex man-machine systems. It was originally designed as an extension of Smalltalk–80, but the current system is an extension of Smalltalk/V Mac [44]. A new class of objects called *actors*

is added. An object of these classes, unlike Smalltalk processes, is a first-class object. They can sent messages and its behavior can be specialized.

Cantor [16] is a research tool developed at Caltech for experimenting with fine-grain message-passing concurrent computers called multicomputers. The physical platform being built is an ensemble of thousands of single-chip computing nodes in which each chip contains a processor, a memory system, and a message-passing interface. A message routing network is provided to connect the processors of message-passing interface together. Since memory is located on each chip, its size is fairly small. This small memory precludes the use of general models that have been developed for more conventional systems.

Plasma–II [92], also known as Plasma-parallel, is an actor language designed to be executed on a set of virtual machines which communicate via messages that can be distributed on diverse types of computer architectures. Each virtual machine is running on a fixed physical processor that can execute many actors in a time sharing mode. An actor can send a message to another one on any virtual machine. Both blocking and non-blocking message passing are supported. Plasma–II supports two types of actors: pure actors and serialized actors. A pure actor is created when it receives a message and then disappears after its script is completed. A serialized actor is what one would think of as an object in more conventional systems–it has a state which is maintained between servicing its messages.

Act 1 [94] is a proposed programming language based on the notion of actors, active objects that communicate by message passing. Act 1 incorporates two special actors: "future" which creates concurrency by dynamically allocating processing resources, and "serializers" which restricts concurrency by constraining the order in which events take place and has changeable local state. Act 1 exploits delegation for knowledge sharing rather than class inheritance. Whenever an actor, called "client", receives a message it cannot answer immediately on the basis of its local knowledge and expertise, it delegates the message to another actor called its "proxy". This avoids the need for duplicating common knowledge in every client actor.

Actor Based Concurrent Language (ABCL) [156, 158] takes the view that objects coexist with other more traditional types of data, i.e. user-defined objects are treated differently than system-supplied types. Objects communicate by passing messages to one another, while more conventional data are manipulated by functions and operations. It supports three types of message passing: "past" (send and no wait), "now" (send and wait), and "future" (reply to me later). Two modes of message passing are supported: "express" and "ordinary". An express message will interrupt the processing of an ordinary message, but not another express message that is being serviced. As such, each object has two message queues–an express queue and an ordinary queue.

The first implementation of the language, called ABCL/1, like the underlying actor model, did not support any form of inheritance, and is therefore an object-based language. However, when adding inheritance to the language, the concept of "inheritance anomaly" is introduced into object-oriented concurrent programming, [97, 98, 99, 100]. Inheritance anomaly is classified into three major categories:(1) Partioning of acceptable states (State Partitioning Anomaly), (2) History-only sensitiveness of acceptable states (History-only Sensitiveness Anomaly), and (3) Modification of acceptable states (State Modification Anomaly). It was shown that existing synchronization mechanisms are weak in one or more of these anomalies. The appearance of these anomalies has a great significance because, from now on, any forthcoming proposals for language tools in object-oriented concurrent programming can and should be demonstrated to successfully solve these critical cases.

2.3 Logic Based Languages

In the last few years, a large number of languages have been proposed which combine logic programming and object-oriented programming. We distinguish between three main groups of these mergers:(1) object-oriented languages extended with logical constructs, (2) two base languages (object-oriented and logic), interfaced, and (3) logic based object-oriented languages, either logic based languages extended with object-oriented constructs or higher object-oriented languages built on top of logic based languages.

This classification was motivated by similar ones proposed in various papers [145, 42, 43, 28].

The examples of object-oriented language extended with logical constructs is Orient84/K [69, 70] and LogiC++ [155]. In Orient84/K, deductive retrieval is facilitated through built-in methods. Besides, all logical constructs are viewed as objects. In LogiC++, a program is basically a C++ program. However, the Horn clauses in Prolog are used to express methods for the objects. Hence, the expressive power of Prolog based on declarative semantics is obtained for defining methods.

The example of a language interface is Prolog/Loops [88], where facilities have been provided for making calls to Loops objects from Prolog, and for setting Prolog goals form Loops (in the form of messages).

Mergers with a logic based language are the most numerous. The historically earliest proposal by Shapiro and Takeuchi [127] and Zaniolo [164] have had strong influence on the development of this group, and may thus be also viewed as belonging here.

In the following, the design and implementation issues concerning classes, objects, state, and inheritance will be discussed. Then, we present the proposals which integrates logic and objects.

Classes

A class is usually mapped to a set of Horn clauses in the underlying logic programming language [102]. In languages that support a module system, a class may be mapped into the module construct of the underlying logic programming language [7]. Classes can be defined as *Mixins*, [85, 54], whose purpose is solely to augment the structure or behavior of other classes. However, there are more varied mappings possible for instances of classes (objects), including sets of clauses, processes, or terms.

Objects

Objects are the run-time instances of classes. However, several languages do not distinguish between them at all; instead preferring to call everything an object [109, 113, 7].

The most popular way is to treat objects as encapsulated label collection of clauses as in [164]. In [79], terms is also treated as (immutable) objects which means that terms can be sent messages. Others, may allow objects to be parameterized [102, 164], called *generic objects* in [102].

Logic programming languages allow programs to have a declarative and procedure interpretation. In concurrent logic programming, a third interpretation is possible where a conjunction of goals can be regarded as a system of concurrent (proof) processes [43]. Each process is an executing recursive predicate, and the processes communicate by partially instantiating shared variables. This process interpretation is the basis of all the languages which combine object-oriented programming and concurrent logic programming. A seminal paper by Shapiro and Takeuchi was the first to investigate the connections between concurrent logic programming and object-oriented programming [127]. Their model, based on the actor model, can be summarized by the following points:

1. An object can be represented as a process which calls itself recursively and holds its internal state in unshared arguments.

2. Objects communicate with each other by instantiating shared variables. Usually, one of the arguments is a message stream to the object, represented as a lazy list (one whose tail can be undetermined).

3. An object becomes active when it receives a message, otherwise, it is suspended.

4. An object instance is created by process reduction.

5. A message is responded to either by binding a shared variable in the message, so-called incomplete messages, or by sending a dedicated message in the opposite direction.

6. Inheritance is implemented by forwarding unrecognized messages to another object.

There are a few things to note about this model. First, the type of inheritance that follows naturally from this model supposes that all the ancestors of an object from which it inherits properties are themselves fully fledged objects. This can be easily modeled within the concurrent logic programming languages: each object has separate private channels back to the ancestor through which it passes back the information. The hierarchical structure of the objects is reflected by the structure of the communication network that they form. This can also cope with multiple inheritance using several channels, though the mechanisms used become somewhat cumbersome.

Second, when we create an instance of an object, it is also necessary to create a fresh instances of all its ancestors. So instead of creating one object, we may need to create half a dozen separate objects, each of which has the normal object overhead.

Third, there is another perhaps more important difficulty with inheritance in this model which has to do with dynamic binding and the "self" variable. To be able to provide this facility with the explicit channel system that is used in this model, an inheritance path would need to have two channels, one to pass the message up, and the other to pass the self messages back down again. But when they reach the original object, there is a possibility for deadlock. It is currently awaiting a response to its original message and to do that it must defer the consideration of other incoming messages. But this message is itself an incoming message.

Fourth, verbose description of objects with state and communication. Each method must at the very least repeat the names of the state variables in both the head of the method and in the tail recursive call. Each method must explicitly fetch the next method from the stream and then recur on the stream of the remaining messages. Such tedious repetition easily results in subtle mistakes.

Fifth, relying on streams as a communication medium may cause problems. These problems are solved in AKL by introducing Ports [73], objects (agents) that communicate by posting and checking constraints upon bags.

Sixth, no syntactic support for OOP was proposed.

As shown, Shapiro and Takeuchi's work did not deal with some of the fundamental issues involved in OOP, such as multiple inheritance conflicts, self communication, the accessing of state variables and clauses. Consequently, the concurrent logic based OOP languages developed since can and should provide solutions to these problems.

State

One of the most difficult problems on the way to an integration of object-oriented and logic programming is the modeling of changeable object state since multiple assignment of values on logic variables cannot be performed [26]. To overcome this problem, some proposals simulates the assignment statement in logic programming by using the built-in non-logical operators *assert* and *retract* of Prolog[1]. There are many problems with assert/retract which motivate e.g. Warren [149] to talk about "The evils of assert" and which make programming using these operators a rather non-declarative effort [6]. One solution is proposed in [30] has utilized an *intentional* semantics to describe Horn clause logic, which allows intention variables to be used in programs. Such a variable is actually a function from a state to a set of values, and also contains a sequence of states which represents the partial history of the object. The state is changed at resolution and through stepwise instantiation of the history list. Whereas in [34], a new inference rule is introduced which in responding to a goal (message) consumes two literals: the object names and the procedure names and introduces a new object literal in a derived goal which models operations that change the state. Another satisfactory, even if limited solution, could be as in [102] to represent classes as *generic objects* with variable specified in their names. Such variables perform some of the same functions as instance variables in conventional object-oriented programming, although they do not change state. State change can be simulated by creating new instances. From the declarative point of view, as pointed out in [102], variables in the class name stands for infinitely many ground instantiations of the class determined by substituting terms of the Herbrand universe to the specified variables. In other proposals, supported by a concurrent logic programming language, state change is represented by incarnating the tail-recursion process with the new state instead of the old. This lack of side-effects leads to a simpler, clearer declarative semantics.

In concurrent languages, an important problem with state changes is that calculations do not interfere with other calculations using the old state. In [78], the operation that function on the successive state of the object is serialized. Serialization is achieved by using message queues. This is implemented as a definition of a "continuation" method that works on the new state. Another solution in [63] is to maintain a "threaded" state such that only one of the state-using sections in a method body can at the same time be entered. The threading is implemented by means of intermediate variables which correspond to state transitions.

Several languages contain the notion of data-driven programming, which allows daemons to be fired when certain events occur, such as state updates [31, 18].

[1] In some Prolog systems, only predicates declared as *dynamic* is allowed to be modified by other predicates.

Inheritance

Wegner [151, 152, 153] classify object-based programming languages into three categories: *object-based* languages, *class-based* languages, and *object-oriented languages*. A language is *object-based* if it supports objects as a language feature. A language is *class-based* if it requires all objects to belong to a class. A language is *object-oriented* if it requires classes to support inheritance. Liberman [93], differs in believing delegation and inheritance to be different mechanisms. Systems based on delegation are called *prototypes* and, it is noted that they serve both instances and templates for descendents. In these systems, an object can delegate, by forwarding, messages to one or more of its designated ancestor objects. There is a trade off of space versus execution time between class inheritance systems and prototype systems. Prototype systems require less space but more time to bind methods or obtain attribute values. By contrast, systems using class inheritance have faster method lookup but may require more space.

Most languages are based on the notion of prototypes [146, 79]. As mentioned before, with Shapiro and Takeuchi's model this may have the overhead of creating instances of the ancestors of the newly created object.

Multiple inheritance introduces the problem of deciding which method to use when there are methods with the same name but different or unrelated semantics inherited from different superclasses. This problem is often resolved in conventional OOP languages, by imposing a standard search order: the ancestors of a class are searched in some predetermined order and the first one which contains an implementation of the method is used. The search order is often based on the order of declaration of ancestors in a class. This is simply provided in [164, 110], as a default strategy, by relying upon the same searching principle of the underlying Prolog system. This search is a branch-first, left-to-right traversal of the object hierarchy starting at the object (or instance of an object) to which the message was originally sent. In [124], the problem is handled by forcing the user to choose which ancestor to inherit from.

In addition to inheritance, some languages allow an inverse of the inheritance relation to coexist. One example in [164] is the "sub" relation which returns the name of the object below the current one in the inheritance graph and through backtracking can return them all. This leads to another technique of searching the inheritance graph.

Conventional OOP languages presume *overriding inheritance* which in the logic context means that if a given class owns a definition for a called method, and the invocation fails, then no definition from superclasses will be searched for or tried. In mergers with logic, it seems equally reasonable to allow further pursuing until either success will be achieved, or the place of appropriate definitions will be used up. Such inheritance is called *cumulative* [145]. Other language designers have taken the view

that a message send should not fail, as in [33]. In [102], a *differential inheritance filter* capability is provided in order to be able to exclude inherited methods.

Inheritance is usually implemented by inheritance rules [31]. When the matching of a message with inheritance rules results in searching the inheritance graph at run-time, a much run-time overhead is added. Another implementation approach for inheritance, is to copy all the clauses of the inherited classes over to their inheritor at compile time [33, 79, 57]. In [124], all inheritance is computed at compile time. However, it does not resolve the collision that occurs when one class inherits the same method from two unrelated ancestors. In languages based on Shapiro and Takeuchi's model [79], a link between an object and its ancestor is usually implemented by a shared logical variable which acts as a stream for forwarding messages from the object. Multiple inheritance presents a problem since languages based on concurrent logic programming cannot use backtracking to search through the inherited objects for the right clause, and once a message has been forwarded to a particular object, the choice is fixed. One answer is to broadcast the same message to all the inherited objects, but this makes it possible for a variable in the message to be bound more than once, causing one or more of the objects to fail. In [41], broadcasting primitives exist which automatically make unique copies of a message for each inherited object. Even so, problems still remain about what to do if more than one copy of the message is successfully processed, or if all the copies fail to deal with.

Another important feature in OOP is the self communication, so-called *self-reference*, since it allows the meaning of the self application of a method from a class to depend on method definitions in the class of the object that was originally received the message, normally below it in the inheritance hierarchy. Typically, this is achieved through late binding, which identifies the object on whose behalf an operation is being executed rather than the textual module in which the self reference occurs. This means that a self message is always tried against methods in the class at the bottom of the inheritance graph for a particular invocation. Some languages disallows self communication, e.g. [117].

Self-reference may be represented as an extra argument in every message [7]. This extra term will contain the name of the initially invoked object, and then if a self message is necessary, the corresponding goal will use this name to invoke this object. In concurrent logic programming, self-reference is normally implemented by adding an extra (output) message stream to every object [41, 79]. Self streams can exist throughout the lifetime of the objects, but an alternative is to only set them up for the duration of the computation of the forwarded messages, as in [79]. In [63], self messages can be given priority over external communication.

In Prolog++, there is a "myself" message which is applied to the method in the class in which the myself occurs [109].

Languages that represent the state by clauses allows state to be inherited much the same like methods. Languages based on Shapiro and Takeuchi's model, [41], do not permit state to be inherited. This is due to the implementation of state as unshared argument of a process, which makes it difficult to share with other processes. The normal solution is to force the state in an inherited object to be explicitly manipulated by clauses which are activated by messages sent by the inheritor. However, the use of such messages may interact unfavorably with messages from other parts of the object network which manipulate the same state.

Zaniolo's objects [164] is the seminal work on combining Prolog with OOP. It is derived from work in AI on "isa" hierarchies. This represented parameterized objects with methods lists, expressed directly using Prolog operators. Inheritance is overriding. Ancestors are tried in the order in which they are declared. The implementation uses meta-interpretation in Prolog. Self-reference is not supported.

Extended Self–contained Prolog(ESP) [31, 32] is a language built on top of KL0, a Prolog like language, and the run-time is directly supported by PSI, a Personal Sequential Inference machine. It is designed as a system programming language for PSI. Objects are represented as vectors of KL0. Changes of instance variables are performed by assigning new values to these vectors; an efficient implementation of destructive updates similar to assert and retract operations. Names of classes to be inherited must be listed in a class definition in the order they are inherited. ESP imposes many restrictions on the use of logical variables, e.g. in a message passing, the variables which stands for the receiver must be bound before the execution. There is also a problem concerning the recovery of backtracking effect.

Prolog++ [109, 110, 146, 111, 7] is an extension of LPA Prolog with object-oriented programming. It is a prototype language. Objects are either static or dynamic. Dynamic objects can be created or augmented at run-time, while static objects are fixed and optimized at compile time. Methods are Prolog clauses. They can be defined as public or private, have multiple definitions and be truly dynamic, i.e. they can be augmented at run-time. Inheritance is overriding. Multiple inheritance is allowed. Prolog++ supports data driven programming through special objects called daemons. State is changed through the assert and retract primitives of LPA Prolog. Prolog++ provides a set of object variables. These are reserved words of the language which are used for representing arbitrary objects whose names are calculated at run-time. For example, "self" denotes the self object, "myself" denotes the object in which the word textually appears, and "sub" denotes a child object.

Logic and Objects [102, 101] has been developed at Imperial College, London, by McCabe. It is a very expressive language system incorporating ideas from Prolog, object-oriented programming systems, and functional programming. Both *overriding* and *cumulative* inheritance are supported. The system is based on a concept called "class templates". These are classes which are parameterized and when they

are called enabling a declarative approach to generic descriptions of objects. This concept has proposed a solution to the treatment of state which relies on the instance's label being returned at the end of message processing. The label will contain new state values as its parameters, and these can be used as the initial state for the instance when it is next invoked. However, all other instances must be sent this label in order for them to use the changed instance. Multiple inheritance is supported. The resolving of the conflicts that may be caused because of multiple inheritance is left to the programmer. A differential inheritance filter capability is provided in order to be able to exclude inherited methods.

SICStus Prolog Objects System(Prolog Objects) [7] is an extension of SICStus Prolog with object-oriented programming. It is a prototype language. An object is a named collection of Prolog predicate definitions. In addition an object may have attributes that are modifiable through built-in access and update methods. State is changed through the assert and retract primitives of SICStus prolog. In addition, generic objects is supported which defines objects with variables which have a global scope to the object being defined. Objects may be defined in a file, or dynamically created during the execution of a program. Inheritance is overriding. Multiple inheritance is allowed as well as light weight objects called instances. Such an object may only inherits from one object and have only attributes derived from its super object. Self-reference is allowed and the self object may be retrieved in any method using the method "self". Every object is translated to a SICStus module. Each object that has a subclass posses automatically the "sub" relation which returns its subclass.

Logic, Inheritance, Functions, Equations(LIFE) [5] is based on [4] which represents a rather different approach to incorporating objects into logic. LIFE uses the so-called ψ-terms instead of conventional terms to represent classes and objects. Arguments in a complex ψ-term (corresponding to a normal compound term) are indicated by attribute/value pairs rather than by their positions. A labeling schema over terms is employed to logically link objects into "isa" hierarchies. Attribute inheritance is then achieved by overloading unification to take the term hierarchy into account when attempting to match two terms. Assignment is supported, including assignment to terms, though no semantics is as yet forthcoming. There is not possibility of overriding. Nor can we express the notions which involve the use of self.

SPOOL [53] is an object-oriented language built on top of VM/Programming in Logic, an IBM implementation of the Prolog. From the viewpoint of Prolog, the language supplies Prolog with the facilities for modularizing program database. Methods are treated as Prolog clauses and messages to objects as goal invocations. State information is stored as extra clauses, which are asserted and retracted when the state of an instance needs to be changed. A special type of messages called "anonymous" is supported in order to be able to send a message without explicitly speci-

fying the receiver. The object which can serve this message will respond. Multiple inheritance is allowed. Inheritance is overriding. Self-reference is not allowed. The compiler attempts to reduced the number of asserts and retracts by analyzing message passing and collecting together sets of state accesses and changes.

Structured Concurrent Object Oriented Prolog(SCOOP) [147] is an experimental language implemented in Prolog that tries to combine the best of logic, object-oriented and concurrent programming. Classes represent independent Prolog programs. Some clauses are immutable and fixed for all objects; but others, can be asserted and retracted by the object. The fixed (static) clauses act as methods whereas the dynamic ones act as state variables. Objects can be augmented with new clauses passed as parameters at creation time or asserted/retracted during program execution. Dynamic predicates are limited to facts for efficiency reasons. SCOOP support processes, synchronized by the exchange of messages. Self object may be retrieved in any method using the method "thisobject". Inheritance is overriding. Messages are distinguished as "local call" that accesses predicates internal to the object or "remote call" that accesses predicates of other objects. In order to protect locality and protection for object's state, SCOOP does not allow the assert or retract as goals in a remote call. This restricts the modification of object's state to its own context. Management and access of dynamic clauses are more complex than for static clauses. In particular, dynamic clauses are replicated in each object whereas static clauses exist in only one copy.

O–CPU [104, 105] combines logic and object-based programming in a concurrent language. Objects, called P-units, are expressed as Prolog databases, i.e. as a collection of Horn clauses. Methods are represented as a collection of Prolog clauses. P-units activate "demonstration processes" to solve goals, i.e. to serve messages received, and they can activate parallel demonstration processes when needed. Similar to the actor systems, O–CPU supports passive demonstration processes(PDP) and active demonstration processes(ADP). A PDP is like an unserialized actor-it does not have a private state. Lack of an internal state allows a high degree of concurrency to be achieved, but also makes the PDP of limited usefulness. An ADP does have an internal state, which means that synchronization may be required. O–CPU does not allow logical variables to be shared between different ADPs. For this reason, ADP interactions cannot be expressed in terms of streams and annotated variables. Communication between ADPs has been implemented by using particular passive objects, Prolog terms, that behave like queues.

Linear Objects [12, 13] is based on linear logic [55]. This language utilizes the usual idea of equating an object with a process, and state with process arguments, but this is encoded using clauses containing multiple literals in their heads (cf. [34]). Each clause can be thought of as a method, while the head literals correspond to object states. A method is selected if its head matches the goal corresponding to the object in its current state. A crucial aspect of the matching is that it is done in terms of

multi-set inclusion (i.e. unordered lists) such that the object must contain at least all the states in the head. This permits an object to contain more state values than the method it is using, and so the object can be thought of as a more specialized instance of the class whose methods it is utilizing. This is a very elegant way of encoding inheritance since an object will only consist of structured process. This is in contrast to an approach which uses delegation as its inheritance mechanism, where there will be as many processes as there are classes, and a complex communication protocol is required. Another criticism is that a method does not belong to a distinguished class explicitly by the program; its owner is only determined at run-time by the set of object literals which the method lists in its head.

Sandra [46, 47] is an object-oriented language built on top of the concurrent logic language core-Sandra [45, 48]. Core-Sandra is based on the Shapiro and Takeuchi's model. Before a predicate is defined, the types and the mode of instantiation of its arguments must be declared. Sandra implements inheritance by delegation. The designer of a guardian, an object definition in Sandra, is allowed to choose a subset (possibly empty) of the interface of the ancestor guardian. Self-reference is allowed. Multiple inheritance is allowed. However, it is the designer's responsibility of the inheriting guardian to resolve any name conflicts due to multiple inheritance.

Vulcan [78, 79] is an object-oriented language built on top of *Concurrent Prolog*. It is based on the Shapiro and Takeuchi's model. The main purpose in building Vulcan is to overcome the shortcoming of object-oriented programming in *Concurrent Prolog*. This concerning its verbosity and the poor linguistic support. Vulcan reduces verbosity by allowing the user programs declare the state variables once and methods are written in a concise notation and expanded into *Concurrent Prolog*. The pseudo-variable "Self" is a stream which refers to the receiver of the message. Message sending is serialized such that messages are function on successive state of the object. Vulcan packages any expressions lexically after a send to Self into a "continuation" method in order to refer to the new state. Vulcan offers a uniform way of dealing with objects as processes and terms. This has lead to the treatment of terms as immutable objects which means that terms can send messages. Vulcan implements inheritance in two ways. The first is description copying semantics corresponding to subclassing. The second is inheritance by delegation to parts described by Shapiro and Takeuchi's model. For subclassing, the subclass is created with source copies of all methods and instance variables inherited from its superclasses. For delegation, each method is translated into two clauses: one for handling normal messages and another for handling delegation messages. Multiple inheritance is allowed. However, it has the same criticism concerning the multiple inheritance problems in Shapiro and Takeuchi's model.

Polka [41] is an object-oriented language built on top of Parlog. It is based on the Shapiro and Takeuchi's model. Polka treats terms as classes, called "term classes". Term classes are terms of the underlying logic programming language which are

manipulated by the meta-interpretations. Whereas, other constructs are based on the committed choice process view. State variables are assigned new values using an operation called "becomes". This allows state variables to be encoded in much less verbose way than the underlying language. Polka also has a special type of message called a "suspendable" message. This is an ordinary message tagged with a list of logical variables which will not be delivered to its destination until all the variables on its associated list become bound. This is the way to handle sending messages in a batch. Polka allows an object to execute a method without waiting for a message to arrive. This is useful when encoding clock, producers, or other types of generator. Self-reference is allowed. Multiple inheritance is allowed. However, it has the same criticism concerning the multiple inheritance problems in Shapiro and Takeuchi's model.

Logic Programs with Inheritance [57] is introduced in order to remedy the main weakness in Shapiro and Takeuchi's model [79]– verbose description of state change and of stream communication (cf. [78, 79]), and the lack of class-like inheritance. A procedure is a class definition. Each clause of the procedure is viewed as a method. State is unshared arguments. The concept of "implicit arguments" is introduced in order to reduce the proliferation of variable names. The new style differentiates between global and local (hidden) arguments. Multiple inheritance is allowed. In addition, an inherited class can inherit the same class several times with different instantiation of the class parameters. Self-reference is not allowed and "self" is used as a sugared syntax for the tail recursive call that simulates a state change. Inheritance is implemented by code copying. The resolving of the conflicts that may be caused because of multiple inheritance is not supported.

Oz [63, 62] integrates the features of OOP to kernel Oz [131]. It is a very expressive language system incorporating ideas from constraints and concurrent logic programming. The state of objects and messages are represented as records, a more general structure than Prolog terms. Each state record has the label "state" and consists of fixed number of attributes. The order of attributes is not significant. State change is provided by the side-effect free predefined operation "adjoinAt" which constructs a new state record by adjoining to a given state record a given value at a given attribute. Objects are created as instances of classes. Classes define methods and attributes. A class can inherit from one or several classes. Objects can only be created from classes that inherit directly or indirectly from the predefined class "UrObject". The label of a message specifies the method to be applied. A method is modeled as a procedure, called abstraction, computing an output state from an input state, a message, and the receiving object. The body of a method threads the input state of the method through state access and update procedures and returns it as the output state. The construct "method application" gives messages a priority over external communication. This applies the message directly to the available state instead of scheduling the message to the object where other messages may be served first. Using method application, batch methods can be defined as a way to

enforce an order on messages. Oz makes it possible to express lexically scoped privateness of attributes and methods that should not be visible outside. Unrecognized messages is forwarded to the method "otherwise" defined in UrObject for reporting an error. Two types of objects are supported: cell-based objects and port-based objects (cf. [73]). The "cell" construct provides cell objects with a minimal form of state change. A Port object enhancing stream-based objects by eliminating the need for stream merging in case of many-to-one communication and also guarantees that streams are written consistently. Objects are concurrent due to the inherent concurrency of Oz. Objects are synchronized using messages that works upon the availability of the state. Multiple inheritance brings the complication that several ancestors of a class may define a method for the same label. The general rule for resolving such inheritance conflicts is that the closest ancestor defining the method is chosen. Obviously there are situations where the closest ancestor defining a method is not unique. In these situations classes appearing more to the right in superclasses declaration are given precedence over classes appearing more to the left (cf. CLOS [21, 82]). If these additional precedences do not suffice to resolve an inheritance conflict, Oz will reject the class definition and print an error message.

2.4 AKL+

AKL+ is the main contribution of this thesis. It is built on top of AKL. AKL is a major landmark in the new field of concurrent constraint programming incorporating ideas from constraints and concurrent logic programming. The concurrent constraint model can accommodate object-oriented programming along the lines of Shapiro and Takeuchi's model. Unfortunately, this model is intolerably low-level. AKL introduced the "port" construct that efficiently solves the problems with streams in this model [73]. AKL+ utilizes a very simple and flexible concepts that supports all features that are needed for any object-oriented application.

An AKL+ class can be defined with attributes, methods, access control of methods, and superclasses. It supports generic classes. A parameter of a generic class lies in four categories: class abstraction, constant, object or method abstraction. The passing of a class parameter is not resolved at compile time but rather all references are bound at run-time. This is due to the dynamic binding of parameters with the actual code. A set of implicit behavior is defined for each defined class. These are supported for error handling, attributes manipulation, and class membership. Classes, methods and instances of classes can be expressed as first-class values in the language which may be passed as arguments, returned as results and stored in attributes of objects. Defining classes and methods as abstractions provides for all higher-order programming techniques.

One of the most important design issues, in AKL+, is to supply programmers with the minimum set of efficient and effective built-in standard classes (library). This fulfills the users computational needs as it provides him with a simpler, and easier-to-use computing environment. AKL+ standard classes are classified into two categories: state representation standard classes and object type standard classes. AKL+ provides two standard state classes: the "state_hash" class and "state_array" class as an important piece of global information that is used by instances or other clients of the class. These standard classes greatly improve the efficiency in manipulating attributes. AKL+ provides three standard object type classes: the "object" class for creating port objects, the "cell" class for creating objects as light-weight objects that provides a minimal form of state change, and a "sync_object" class for synchronizing the acceptance of messages to the objects. These standard classes create objects with encapsulated state. AKL+ provides two standard classes for programming the synchronization constraints namely: "synchronizers" and "transitions". The main advantage to the synchronization constraints schemes in AKL+ is the clean separation of concurrency control and the method specification such that they can be inherited, overridden, or extended separately without affecting each other. Furthermore, one scheme can be integrated and composed with other schemes. These standard protocols are provided to support synchronization schemes for resolving the "inheritance anomaly" [97, 98, 99, 100].

In AKL+, each method is executed in the context of a class, called "base class". This class may not be the class where the method is defined. The current contextual class is used to determine dynamically which methods are called. Within a method, the base class is referred to by the reserved context variable "Self" and the state is referred to by the reserved variable "State". Self is bound to the abstraction of the dispatch table of the base class which when applied to a method invokes its method definition. State is bound to an object.

Method definitions can be called in two ways: "method delegation" and "method invocation". The difference between a method invocation and delegation is that the called class of the method invocation will be the base class of the invocation while the base class of the method delegation will depend on the context, denoted by the reserved variable Self, of the calling definition. This affects the dispatching of methods in case of base class application, *base-class reference*. In Simula [38], this notion of context is called the *qualification* of reference. These calls applies the message directly to the available state. The body of a method threads the input state of the method through the method calls and returns it as the output state. The target object is available under the special attribute "self" so that methods can send messages back to their target objects.

Objects can be explicitly allocated or destroyed using meta level operations provided by the language. Objects can share a common object by passing it as a parameter

to their classes at creation time. AKL+ achieves a uniform message sending which means that objects of different types can receive messages in the same manner.

The language is very flexible to support data-driven programming like, specifying daemons, default methods, and class-specific methods. A daemon can be specified as an active constraints on attributes values or as a method triggered when an object is no longer in use. A default method is automatically invoked when there is no message selector matches with any methods of the interface. A class-specific method makes it possible to invoke methods on the basis of the identity of classes.

AKL+ supports *differential inheritance* where the designer is able to be selective about what is inherited (directly or indirectly) from a superclass. AKL+ supports multiple inheritance. Operations are inherited along the inheritance graph, excluding differentially inherited and private operations encountered, until redefined in a class. If a class inherits operations with the same selector from more than one superclass, a default (implicit) differential inheritance is applied; excluding all the methods with the same selector occurring further on the right. In AKL+ the designer is able to resolve the conflict in different ways. One way is to redefine the operation in the subclass. An alternative way is to differentially inherit the conflicting operation.

Since efficiency is one of our prime motivations, the class dispatcher is computed at compile time and this will result in dispatching methods in a constant time rather than searching the class hierarchy each time a method is invoked. This run-time overhead is unpredictable and dependent on the depth of the inheritance path.

Comparing AKL+ with Other Languages

In what follows we compare our language (AKL+) with the most related ones, i.e. with those languages that are based on concurrent logic programming. We will compare with Vulcan, Polka, Sandra, Logic Program with Inheritance, and Oz.

Vulcan supports inheritance by code copying. This reduces the run-time costs, but increases the size of classes. In AKL+, the inheritance is computed at compile time in such a way that the class dispatcher is cleanly captured. Like Vulcan, both inheritance and delegation are supported. Vulcan resolves the interference of state change by an auxiliary method definition that works on the new sate. Unlike Vulcan, AKL+ utilizes the state threading through the method body. Another difference is that AKL+ provides mechanisms for resolving multiple inheritance conflicts in at least two ways. An AKL+ instances of class "object" is like instances in Vulcan. However, state is encoded as a separate object whose identity is held as the unshared argument in the process of the object, which is generalized to be included in other types of objects. This has the advantage that if state is represented as an object

at the language level then uniform access and encapsulation are achieved. Another similarity is that verbose description of state change is avoided.

Polka supports delegation where the class hierarchy is searched each time a method is invoked. With multiple inheritance, this causes a problem since languages based on concurrent logic programming cannot use backtracking to search through the inherited objects for the right clause, and once a message has been forwarded to a particular object, the choice is fixed. Polka utilizes broadcasting primitives which automatically make unique copies of a message for each inherited object. Even so, problems still remain about what to do if more than one copy of the message is successfully processed, or if all the copies fail to deal with. In AKL+, the method dispatching mechanism does not allow such a situation. Another difference is that Polka does not provide any solution for resolving multiple inheritance conflicts. An AKL+ instance of class "object" is like an instance in Vulcan but the state is held as unshared argument in the process of the object.

Sandra supports delegation where the class hierarchy is searched each time a method is invoked. Unlike AKL+, it is the designer's responsibility of the inheriting guardian to resolve any name conflicts due to multiple inheritance. Another difference is that in Sandra it is required to define the types and the mode of instantiation of the method's arguments.

Logic Program with Inheritance supports inheritance by code copying. In AKL+, the inheritance is computed at compile time in such a way that the class dispatcher is cleanly captured. Self-reference is not allowed and "self" is used as a sugared syntax for the tail recursive call that simulates a state change. In AKL+, both self-reference and base-reference are allowed. Another difference is that Logic Program with Inheritance does not suggest any capabilities to resolve the multiple inheritance conflicts. Like AKL+, Logic Program with Inheritance differentiates between global and local arguments. Global arguments in both languages have the same treatment. However, local arguments in AKL+ have a scope limited to a method definition rather than the entire class definition. In Logic Program with Inheritance an inherited class can inherit the same class several times with different instantiation of the class parameters. This is not allowed in AKL+.

Oz is perhaps the most closest to AKL+. There are some similarities between Oz and AKL+. Both use an efficient data-structure to represent the object's state. Both resolve the interference of state change by threading the state through the method body. The method application construct of Oz is equivalent to method delegation of AKL+. Both can express privateness. Both support cell based and port based objects with the same functionality. Both support abstractions that provides for all higher-order programming techniques. Both adopt class based inheritance. However, we differ in the approach used in handling multiple inheritance. Oz strategy specifies a linear, overall order of classes, and then specifies that application of a class method or attribute starts from the most specific class. As pointed out by

Snyder [133], the main problem with this approach is that the ordering of superclasses in a class declaration has significant semantic implications. In AKL+, we don't flatten the inheritance graph into a linear chain, and then deals with this chain using the rules for single inheritance but instead we model the inheritance graph directly. AKL+ provides mechanisms for resolving multiple inheritance conflicts in at least two ways.

Other differences which hold with Oz and all the above languages are the following. AKL+ supports features which are unique to the above related languages. AKL+ provides the synchronization schemes that resolve the "inheritance anomaly". AKL+ supports *differential inheritance* where the designer is able to be selective about what is inherited. AKL+ supports the *qualification* of reference where each method is executed in the context of class that can be changed. AKL+ supports data-driven programming like, specifying daemons, default methods, and class-specific methods.

2.5 Summary

The discussions above illustrated a wide range of object-oriented programming languages. In our presentation, we have classified these languages as conventional languages, actor based languages, and logic based languages. The logic based languages are of our main interest. We have classified mergers which combine object-oriented and logic programming into three groups. We have focused our attention to the mergers which either logic based languages extended with object-oriented constructs or higher object-oriented languages built on top of logic based languages. These languages are discussed by examining the design and implementation issues concerning classes, objects, state, and inheritance. These issues are examined because of their central roles in object-oriented programming.

At the end of our discussion, we have presented our language (AKL+) and contrasted it with Vulcan, Polka, Sandra, Logic Program with Inheritance, and Oz. The reason of the comparison is that these languages are attempt to combine features of object-oriented programming with concurrent logic programming.

Vulcan, Polka, Sandra and Logic Program with Inheritance do not support mechanisms for multiple inheritance conflict resolution. Furthermore, they do not provide uniform access to the object's state. Vulcan, Polka and Sandra use delegation where the class hierarchy is searched each time a method is invoked. This run-time overhead is unpredictable and dependent on the depth of the inheritance path. Vulcan and Logic Program with Inheritance use inheritance by code copying. This reduces the run-time costs, but increases the size of classes. Oz and AKL+, as object-oriented languages, have many features in common: classes, objects, treatment of state, and privateness. AKL+ differs from Oz in the approach used to handle multi-

ple inheritance. Oz uses the "linear" approach to handle multiple inheritance where the order of parents is significant and defines inheritance. Another difficulty is that in some cases it will not be possible to construct a class precedence list and the class definition disallowed. AKL+ uses the "graph" approach to handle multiple inheritance. In AKL+, the only relationships among classes are those defined by the subclass–superclass relationship. A salient advantage to the AKL+ approach to inheritance is that it allows greater flexibility in resolving any conflicts due to multiple inheritance. One way is to redefine the operation in the subclass. An alternative way is to use the *differential inheritance* feature to exclude the conflicting operation.

As pointed out in [97, 98, 99, 100], concurrent object-oriented languages can and should support mechanisms to solve "inheritance anomaly", protocols are provided in AKL+ to support synchronization schemes for resolving this type of anomaly. Furthermore, other features which are also unique to the aforementioned related languages are: the *qualification* of reference where each method is executed in the context of a class that can be changed, and data-driven programming like, specifying daemons, default methods, and class-specific methods. Since efficiency is one of our prime motivations, the class dispatcher is computed at compile time and this will result in dispatching methods in a constant time.

As shown, AKL+ utilizes a very simple and flexible concepts that supports all features that are needed for any object-oriented application.

⋆ ⋆ ★ ⋆ ⋆

3

AKL

In this chapter, we will present AKL (Agents Kernel Language[1]) because it was chosen as concurrent constraint logic programming element of our language (AKL+). The reasons for this include its use of deep guards, its use of don't know nondeterministic capabilities of Prolog and the constraint logic programming languages with the process-describing capabilities of concurrent logic languages such as GHC, and the simplicity and flexibility in its support of multiple programming paradigms, such as concurrent, object, functional, logic, and constraint programming. In addition, AKL offers a large potential for automatic parallel execution.

The chapter is organized as follows. In Section 3.1 we give an overview of concurrent constraint programming. However, it is not intended to constitute an introduction of this topic, and suitable reading material will be referred to during discussion. In Section 3.2 we give a brief and informal summary of AKL. In Section 3.3 the basic techniques that allows us to do object-oriented programming in AKL are reviewed. In Section 3.4 we present the syntactic sugar that enable us to express programs into clausal syntax, which may be used to give definitions the familiar reading of the logic programming community. This syntax is also available in AKL+.

3.1 Concurrent Constraint Programming

AKL is based on the concept of Concurrent Constraint Programming (CCP), a paradigm distinguished by its elegant notions of communication and synchronization based on constraints [123]. It has been proposed as an alternative to committed choice languages [128] and and/or parallel execution models. The CCP languages offer parallelism that is programmer-controlled, processes are loosely coupled, multiparadigm applications can easily be built, constraints can be utilized in a coherent way and last but not least, existing sequential code can be incorporated in this new concurrent framework [81].

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*, see Fig. 3.1. These two operations provide the necessary primitives for concurrent communication and synchronization. The information in the store is expressed in terms of *constraints*,

[1] AKL was formerly known by the name Andorra Kernel Language.

which are statements in some constraint language, usually based on first-order logic, e.g.,

$$\mathrm{X} < 1,\ \mathrm{Y} = \mathrm{Z} + \mathrm{X},\ \mathrm{W} = [\mathrm{a},\ \mathrm{b},\ \mathrm{c}],\ \ldots$$

If telling makes a store inconsistent, the computation fails. 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. The notion of constraints in AKL

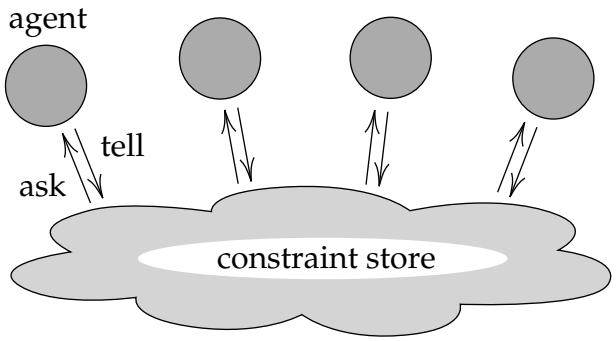

Figure 3.1: Agents interacting with a constraint store

is generic. The range of constraints that may be used in a program is defined by the current *constraint system*, which in AKL, in principle, may be any first-order theory. Constraint systems as such are not discussed here, for more details see [29]. For the purpose of this introduction, we will use a simple constraint system with a few obvious constraints, which is essentially that of Prolog [136] and GHC [143].

Thus, constraints in AKL will be formulas of the form

$$\langle expression \rangle = \langle expression \rangle$$
$$\langle expression \rangle \neq \langle expression \rangle$$
$$\langle expression \rangle < \langle expression \rangle$$

and the like. Equality constraints, e.g., $X = 1$, are often called *bindings*, suggesting that the variable X is *bound* to *1* by the constraint. Correspondingly, the act of telling a binding on a variable is called *binding* the variable. Expressions are either *variables* (alpha-numeric symbols with an upper case initial letter), e.g.,

X, Y, Z, X1, Y1, Z1, ...

or *numbers*, e.g.,

1, 3.1415, -42, ...

or *arithmetic expressions*, e.g.,

1 + X, -Y, X * Y, ...

or *constants*, e.g.,

a, b, c, ...

or *constructor expressions* of the form

⟨*name*⟩(⟨*expression*⟩, ..., ⟨*expression*⟩)

where ⟨*name*⟩ is an alpha-numeric symbol with a lower case initial letter, e.g.,

s(s(0)), tree(X, L, R), ...

There is also the constant *[]*, which denotes the empty list, and the list constructor *[*⟨expression⟩|⟨expression⟩*]*. A syntactic convention used in the following is that, e.g., the expression *[a|[b|[c|d]]]* may be written as *[a, b, c|d]*, and the expression *[a|[b|[c|[]]]]* may be written as *[a, b, c]*. In addition the constraints *true* and *false* are available, which are independent of the constraint system and may be identified with their corresponding logical constants.

3.2 Language Design

In this section AKL is introduced one language construct at a time, also explaining its behavior. For a formal definition of the computation model see elsewhere [58, 74, 50, 51].

3.2.1 Basic Concepts

The agents of concurrent constraint programming correspond to statements being executed concurrently. Constraints, as described in the previous section, are atomic statements known as *constraint atoms* (or just constraints). When they are asked and when they are told is discussed in the following.

A *procedure atom* statement of the form

$\langle name \rangle$(X1, ..., Xn)

is a defined agent. In a procedure atom, $\langle name \rangle$ is the functor, an alpha-numeric symbol, and n is the arity, the number of arguments, of the atom. The variables *X1, ..., Xn* are the *actual parameters* of the atom. Occurrences of procedure atoms in programs are sometimes referred to as *calls*. Atoms of the above form may be referred to as $\langle$name$\rangle/n$ atoms, which uniquely identifies the corresponding procedure atom, e.g.,

plus(X, Y, Z)

is a *plus/3* atom.

The behavior of atoms is given by *procedure (agent) definitions* of the form

$\langle name \rangle$(X1, ..., Xn) := $\langle statement \rangle$.

The variables *X1, ..., Xn* must be different. During execution, any atom matching the left hand side will be replaced by the statement on the right hand side. For example,

plus(X, Y, Z) := Z = X + Y.

is a definition of *plus/3*.

A *composition* statement of the form

$\langle statement \rangle$, ..., $\langle statement \rangle$

builds a composite agent from a sequence of agents. Its behavior is to replace itself with the concurrently executing agents corresponding to its components.

A *hiding* statement of the form

X1, ..., Xn : $\langle statement \rangle$

introduces variables with local scope. The behavior of a hiding statement is to replace itself with its component statement, in which the variables *X1, ..., Xn* have been replaced by new variables.

A *conditional choice* statement of the form

($\langle$statement$\rangle \rightarrow \langle$statement$\rangle$
; ...
; $\langle$statement$\rangle \rightarrow \langle$statement$\rangle$)

is used to express conditional execution. Its components are called *(guarded) clauses* and the components of a clause *guard* and *body*. A clause may be enclosed in hiding.

The behavior of a conditional choice statement is as follows. Its guards are executed with corresponding local constraint stores. If the union of a local store with the external stores is unsatisfiable, the guard fails, and the corresponding clause is deleted. If all clauses are deleted, the choice statement fails. If the first (remaining) guard is successfully reduced to a store which is entailed by the union of external stores, the conditional choice statement is replaced with the composition of the constraints with the body of the corresponding clause.

AKL exploits the module system facilities, programs can be divided into different modules. Each module has its own independent procedure name space. Each procedure in the module system, whether built-in or user-defined, belongs to a module. A procedure is generally only visible in the module where it is defined. However, a procedure may be imported by another module. It is thereby visible in that module too. Procedures declared as public in a module declaration are *exported*, e.g.

```
:- module clac.
:- public plus/3.
```

exports the definition of *plus/3* defined in module *calc*. Normally only exported procedures may be *imported*, e.g.

```
calc.plus(X,Y,Z)
```

calls the agent *plus/3* in the module *calc*.

It is now time for a first small example, illustrating the nature of concurrent computation in AKL.

Example 3.1 The following definitions will create a list of numbers, and add together a list of numbers, respectively.

```
:- module foo.
:- public list/2.
:- public sum/2.

list(N, L) :=
    ( N = 0 → L = []
    ; L1,N1 : N > 0 → L = [N|L1], list(N1, L1), N1 is N - 1).

sum(L, N) :=
    ( L = [] → N = 0
    ; M, L1, N1 : L = [M|L1] → sum(L1, N1), N is N1 + M ).
```

□

The following computation is possible. In the examples, computations will be shown by performing rewriting steps on the state at hand, unfolding definitions and substituting values for variables, etc., where appropriate, which should be intuitive. In this example we avoid details by showing only the relevant atoms and the collection of constraints on the output variable N. Intermediate computation steps are skipped. Thus,

foo.list(3, L), foo.sum(L, N)

is rewritten to

list(2, L1), sum([3|L1], N)

by unfolding the list atom, executing the choice statement, and substituting values for variables according to equality constraints. This result may in its turn be rewritten to

list(1, L2), sum([2|L2], N1), N is 3 + N1

by similar manipulations of the list and sum atoms. Further possible states are

list(0, L3), sum([1|L3], N2), N is 5 + N2
sum([], N3), N is 6 + N3
N is 6

with final state N = 6.

The list/2 agent produces a list, and the sum/2 agent is there to consume its parts as soon as they are created. If the tail of the list being consumed by the sum/2 call is unconstrained, the sum/2 agent will wait for it to be produced (in this case by the list/2 agent).

The simple set of constructs introduced so far is a fairly complete programming language, quite comparable in expressive power to, e.g., functional programming languages.

In the following sections, we will introduce constructs that address the specific needs of important programming paradigms, such as processes and process communication, relational programming, and constraints, see also [74, 71]. In particular, we will need the ability to choose between alternative computations in a manner more flexible than that provided by conditional choice.

3.2.1.1 Don't know nondeterminism

Many problems, especially frequent in the field of Artificial Intelligence, and also found elsewhere, e.g., in operations research, are currently solvable only by resorting to some form of search. Many of these admit very concise solutions if the programming language abstracts away the details of search by providing don't know nondeterminism.

For this, AKL provides the *nondeterminate choice* statement.

(⟨*statement*⟩ ? ⟨*statement*⟩
; ...
; ⟨*statement*⟩ ? ⟨*statement*⟩)

The symbol '?' is read *wait*. The statement is otherwise like the conditional choice statement.

The behavior of a nondeterminate choice statement is as follows. Its guards are executed with corresponding local constraint stores. If the union of a local store with the external stores is unsatisfiable, the guard fails, and the corresponding clause is deleted. If all clauses are deleted, the choice statement fails. If only one clause remains, and its guard is successfully reduced to a store which is consistent with the union of external stores, the choice statement is said to be *determinate*. Then, the nondeterminate choice statement is replaced with the composition of the constraints with the body of the corresponding clause.

Otherwise, if there is more than one clause left, the choice statement is said to be *nondeterminate*, and it will wait. Subsequent telling of other agents may make it determinate. If eventually a state is reached in which no other computation step is possible, each of the remaining clauses may be tried in different copies of the state. The alternative computation paths are explored concurrently.

Let us first consider a very simple example, an agent that accepts either of the constants *a* or *b*, and then does nothing.

```
p(X) :=
    ( X = a ? true
    ; X = b ? true ).
```

The interesting thing happens when the agent *p/1* is called with an unconstrained variable as an argument. That is, we expect it to produce output. Let us call *p/1* together with an agent *q/2* examining the output of *p/1*.

```
q(X, Y) :=
    ( X = a → Y = 1
    ; true → Y = 0 ).
```

Then the following is one possible computation starting from

p(X), q(X, Y)

First *p/1* and *q/2* are both unfolded.

(X = a ? **true** ; X = b ? **true**),
(X = a → Y = 1 ; **true** → Y = 0)

At this point in the computation, the nondeterminate choice statement is nondeterminate, and the conditional choice statement cannot establish the truth or falsity of its condition. The computation can now only proceed by trying the clauses of the nondeterminate choice in different copies of the computation state. Thus,

X = a, (X = a → Y = 1 ; **true** → Y = 0)
Y = 1

and

X = b, (X = a → Y = 1 ; **true** → Y = 0)
Y = 0

are the two possible computations. Observe that the nondeterminate alternatives are ordered in the order of the clauses in the nondeterminate choice statement.

The constructs introduced so far give us (constraint) logic programming in addition to functional programming.

Up to this point, the constructs introduced belong to the strictly logical subset of AKL, which has a straight-forward interpretation in first-order logic both in terms of success and failure.

3.2.1.2 Don't care nondeterminism

In concurrent programming, processes should be able to react to incoming communication from different sources. In constraint programming, constraint propagating agents should be able to react to different conditions. Both of these cases can be expressed as a number of possibly non-exclusive conditions with corresponding branches. If one condition is satisfied, its branch is chosen.

For this, AKL provides the *committed choice* statement

(⟨*statement*⟩ | ⟨*statement*⟩
; ...
; ⟨*statement*⟩ | ⟨*statement*⟩)

The symbol '|' is read commit. The statement is otherwise like the conditional choice statement.

The behavior of a committed choice statement is as follows. Its guards are executed with corresponding local constraint stores. If the union of a local store with the external stores is unsatisfiable, the guard fails, and the corresponding clause is deleted. If all clauses are deleted, the choice statement fails. If any of the (remaining) guards is successfully reduced to a store which is entailed by the union of external stores, the committed choice statement is replaced with the composition of the constraints with the body of the corresponding clause.

Example 3.2 List merging may be expressed as follows, as an example of an agent receiving input from two different sources.

```
:- module list_merge.
:- public merge/3.

merge(X, Y, Z) :=
    ( X = [] | Z = Y
    ; Y = [] | Z = X
    ; E, X1, Z1 : X = [E|X1] | Z = [E|Z1], merge(X1, Y, Z1)
    ; E, Y1, Z1 : Y = [E|Y1] | Z = [E|Z1], merge(X, Y1, Z1) ).
```

A merge agent can react as soon as either X or Y is given a value. In the last two guarded statements, hiding introduces variables that are used for "matching" in the guard, as discussed above. These variables are constrained to be equal to the corresponding list components. □

3.2.1.3 Encapsulated computations

To avoid unwanted interactions between don't know nondeterministic and process-oriented parts of a program, the nondeterministic part can be encapsulated in a statement that hides nondeterminism. Nondeterminism is encapsulated in the guard of a conditional or committed choice and in the solution aggregation constructs provided by AKL[2].

The scope of don't know nondeterminism in a guard is limited to its corresponding clause. New alternative computations for a guard will be introduced as new alternative clauses. This will be illustrated using the following simple nondeterminate agent.

[2]The aggregate statement, *bagof/3* is built-in in AKL. Its behavior is to build lists of sequences of alternative results. It provides powerful means of interaction between determinate and nondeterminate code. It is similar to the corresponding construct in Prolog, and a generalization of the list comprehension primitive found in functional languages.

```
or(X, Y) :=
    ( X = 1 ? true
    ; Y = 1 ? true ).
```

Let us start with the statement

```
    ( or(X, Y) | q )
```

The *or/2* atom is unfolded, giving

```
    ( ( X = 1 ? true ; Y = 1 ? true ) | q )
```

Since no other step is possible, we may try the alternatives of the nondeterminate choice in different copies of the closest enclosing clause, which is duplicated as follows.

```
    ( X = 1 | q
    ; Y = 1 | q )
```

Other choice statements are handled analogously.

Example 3.3 As a more involved example of embedded computations in AKL, consider the following N-queens program where calls to *nqueens* are made in guards, thus encapsulating a deep constraint computation. The *bagof/3* agent is a solution aggregate agent, collecting all solutions to a nondeterminate agent. The call *bagof(Q,nqueens(N,Q),Sols)* outputs all solutions to *nqueens* applied to *N* in *Sols*.

The *model/2* procedure waits for input messages on *M*, i.e. it suspends until either *M* is empty or it receives one of the messages *all(N)* or *one(N)* on *M*. If one solution is required the *model* program suspends until the call *nqueens(N,Q)* is entailed, then proceeding by outputting *Q* on *S*.

```
:- module nqueen.
:- public model/2.

model(M, S) :=
    ( M = [] →S = []
    ; N, Sols, M1, S1 : M = [all(N)|M1] →
      bagof(Q, nqueens(N, Q), Sols),
      S = [all(Sols)|S1],
      model(M1, S1)
    ; N, M1, S1 : M = [one(N)|M1] →
```

```
( Q : nqueens(N, Q) →
  S = [one(Q)|S1]
; true →S = [none|S1] ),
model(M1, S1) ).
```

□

Before leaving the subject of don't know nondeterminism in guards, it should be clarified exactly when alternatives may be tried. A (possibly local) state with agents and their store is (locally) stable if no computation step other than splitting a nondeterminate choice is possible, and no such computation step can be made possible by adding constraints to external constraint stores (if any). Splitting may then be applied to the leftmost possible nondeterminate choice in a stable state. Stability is basically an independence property which guarantees that the computations to be performed locally cannot be affected by agents operating on external stores until the local nondeterminate computation proceeds.

3.2.2 Definitions

Fig. 3.2 shows the (abstract) syntactic category pertaining to programs in AKL. We use the following notations in describing the syntax of programs: letters in bold are keywords, letters between ⟨ and ⟩ are nonterminal symbols, letters between ′ and ′ are terminal symbols, and the symbol | separates alternatives.

3.3 Basic Object-Oriented Style

In this section, the basic techniques that allows us to do object-oriented programming in AKL are reviewed. Like the programming techniques in the previous section, they belong to logic programming folklore.

There is more than one way to map the abstract concept of an object onto corresponding concepts in a concurrent constraint language. The first and most widespread of these will be described here in detail. It is based on the process reading of logic programs [127].

$$
\begin{array}{lcl}
\langle program\rangle & ::= & \langle set\ of\ definitions\ of\ modules\rangle \\
\langle module\ definition\rangle & ::= & \langle module\ heading\rangle\langle set\ of\ procedure\ definitions\rangle \\
\langle module\ heading\rangle & ::= & ':-'\ \textbf{module}\ \langle module\ name\rangle \\
 & & \langle set\ of\ public\ definitions\rangle \\
\langle public\ definition\rangle & ::= & ':-'\ \textbf{public}\ \langle functor\rangle'/'\langle arity\rangle \\
\langle procedure\ definition\rangle & ::= & \langle head\rangle\ ':='\ \langle body\rangle \\
\langle head\rangle & ::= & \langle procedure\ atom\rangle \\
\langle body\rangle & ::= & \langle statement\rangle \\
\langle statement\rangle & ::= & \langle atom\rangle\ |\langle composition\rangle\ |\langle hiding\rangle \\
 & | & \langle choice\rangle\ |\langle aggregate\rangle\ |\langle abstraction\rangle \\
 & | & \langle abstraction\ application\rangle\ |\langle message\ send\rangle \\
\langle atom\rangle & ::= & \langle constraint\ atom\rangle\ |\ \langle procedure\ atom\rangle \\
\langle composition\rangle & ::= & \langle statement\rangle','\langle statement\rangle \\
\langle hiding\rangle & ::= & \langle set\ of\ variables\rangle' :'\langle statement\rangle \\
\langle choice\rangle & ::= & \langle sequence\ of\ clauses\ with\ the\ same\ guard\ operator\rangle \\
\langle clause\rangle & ::= & \langle set\ of\ vars\rangle' :'\langle statement\rangle\langle guard\ operator\rangle \\
 & & \langle statement\rangle \\
\langle guard\ operator\rangle & ::= & '\rightarrow'\ |\ '|'\ |\ '?' \\
\langle aggregate\rangle & ::= & aggregate(\langle variable\rangle','\langle statement\rangle','\langle variable\rangle) \\
\langle abstraction\rangle & ::= & \langle variable\rangle\ '='\ (\langle set\ of\ variables\rangle)\ '\backslash'\ \langle body\rangle \\
\langle abstraction\ application\rangle & ::= & 'akl.'\langle variable\rangle(\langle sequence\ of\ variables\rangle) \\
\langle message\ send\rangle & ::= & 'akl.send'(\langle method\ atom\rangle','\ \langle variable\rangle) \\
 & | & 'akl.send'(\langle method\ atom\rangle','\ \langle variable\rangle','\langle variable\rangle)
\end{array}
$$

Figure 3.2: Abstract syntax of AKL program

3.3.1 Objects

Objects are realized as processes[3] that take as input a stream of requests. The list is by far most popular communication medium in concurrent logic programming. In this context lists are usually called *streams*. The stream identifies the object, i.e. the *identity* of the object is preserved by its input stream. The data associated with the objects are held in the arguments of the process. An object definition typically has one clause per type of request, which performs the corresponding service, and one clause for terminating (or deallocating) the object, see Fig. 3.3. Thus, clauses correspond to methods. The requests are typically expressions of the form *name(A, B, C)*, where the constructor "*name*" identifies the request, and A, B, and C are the arguments of the request.

The process description, the agent definition, is the class, the implementation of the object. The individual calls to this agent are the instances.

[3]Agents may be thought of as processes, and telling constraints on shared variables may be thought of as communicating on a shared channel.

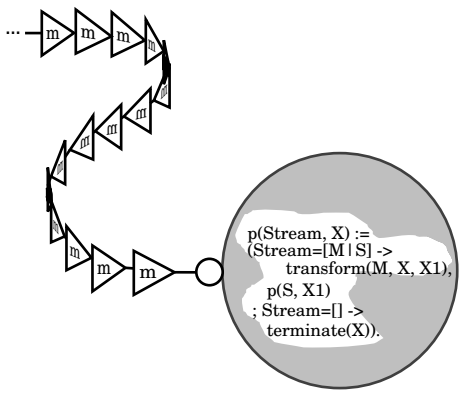

Figure 3.3: An object consuming a list of messages

Example 3.4 A standard example of an object is the bank account, providing withdrawal, deposits, etc.

```
:- module bank.
:- public make_bank_account/1.

make_bank_account(S) :=
    (true → bank_account(S, 0)).

bank_account(Stream, N) :=
    ( Stream = [] → true
    ; A,R,N1: Stream = [withdraw(A)|R] → N1 is N - A,
      bank_account(R, N1)
    ; A,R,N1: Stream = [deposit(A)|R] → N1 is N + A,
      bank_account(R, N1)
    ; M,R: Stream = [balance(M)|R] → M = N, bank_account(R, N)).
```

□

A computation starting with

```
bank.make_bank_account(S),
S = [balance(B1), deposit(7), withdraw(3), balance(B2)]
```

yields

$$B1 = 0, B2 = 4$$

A bank-account object is created by starting a process *bank_account(S, 0)* given as initial input an unspecified stream S (a variable) and a zero balance. The stream S is used to identify the object. A service *deposit(5)* is requested by binding S to

[deposit(5)|S1]. The next request is added to *S1*, and so on. In the above example, only one clause will match any given request. When it is applied, some computation is performed in its body and a new *bank_account* process replaces the original one. The requests in the above example are processed as follows. Let us start in the middle.

bank_account(S, 0), S = [deposit(7), withdraw(3), balance(B2)].

The *bank_account* process is reduced by the clause matching the first deposit request, leaving some computation to be performed.

N = 0+7, bank_account(S1, N), S1 = [withdraw(3), balance(B2)].

This leaves us with.

bank_account(S, 7), S1 = [withdraw(3), balance(B2)].

The rest of the requests are processed similarly.

Finally, there are a few things to note about these objects. First, they are automatically *encapsulated*. Clients are prevented from directly accessing the data associated with an object. In imperative languages, this is not as self-evident, as the object is often confused with the storage used to store its internal data, and the object identifier is a pointer to this storage, which may often be used for any purpose.

Second, requests are dynamically bound, so-called *late binding*. The expression that identifies a request may be interpreted differently, and may therefore involve the execution of different code, depending on the object. This does not involve mandatory declarations in some shared (abstract or virtual) ancestor class, as in many other languages.

Third, *delegation* can be achieved by creating instances of the ancestor objects. The object identifier of (the stream to) this ancestor object is held as an argument of the derived object. The object corresponding to the ancestor could appropriately be called a subobject of the derived object. The derived object filters incoming requests and delegates the appropriate requests to its subobject. Two difficulties may be observed in this setting. It is cumbersome to handle the notion of self communication correctly. It is quite easy to model an interface hierarchy, however, it is difficult to achieve specialization and, hence, overriding. There are also some unresolved issues like multiple inheritance conflicts and the efficiency in executing requests served by ancestors, for more details see Section 2.3.

3.3.2 Ports for Objects

Ports are a special form of constraints, which, when added to AKL, or to any concurrent logic programming language, will solve a number of problems with the approach to object-oriented programming presented above, problems that we have avoided mentioning so far. This section provides a preliminary introduction to ports. They, and the problems they solve, are described in great detail elsewhere [73].

A port is a binary constraint on a bag (a multi-set) of messages and a corresponding stream of these messages. It simply states that they contain the same messages, in any order. A bag connected to a stream by a port is usually identified with the port, and is referred to as a port. The *akl.open_port(P, S)* operation relates a bag P to a stream S, and connects them through a port. The stream S will usually be connected to an object. Instead of using the stream to access the object, we will send messages by adding them to the port. The *akl.send(M, P)* operation sends a message M to a port P. To satisfy the port constraint, a message sent to a port will immediately be added to its associated stream, first come first served.

When a port is no longer referenced from other parts of the computation state, when it becomes garbage, it is assumed that it contains no more messages, and its associated stream is automatically closed. When the stream is closed, any object consuming it is thereby notified that there are no more clients requesting its services.

Thus, to summarize: a port is created with an associated stream (to an object). Messages are sent to the port, and appear on the stream in any order. When the port is no longer in use, the stream is closed, and the object may choose to terminate.

A simple example follows.

akl.open_port(P, S), akl.send(a, P), akl.send(b, P)

yields

P = ⟨*a port*⟩, S = [a, b]

Here we create a port and a related stream, and send two messages. The messages appear in S in the order of the send operations in the composition, but it could just as well have been reversed. The stream is closed when the messages have been sent, since there are no more references to the port.

Ports serve three orthogonal purposes:

- *communication and identification*: The medium links the sender and receiver and provides the sender with a unique identity of the receiver.

- *synchronization*: Many-to-1 communication is enabled by allowing the receiver to nondeterministically choose one of the incoming messages while leaving the others waiting in the medium.

- *buffering*: Messages are stored in the medium until the receiver, residing in a different branch of concurrent computation in form of a suspending clause, is ready to pick them up.

Ports solve a number of problems that are implicit in the use of streams. The following are the most obvious.

- If several clients are to access the same object, their streams of messages have to be merged into a single input stream. With ports, no merger has to be created. Any client can send a message on the same port.

- If objects are to be embedded in other data structures, creating, e.g., an array of objects, streams have to be put in these structures. Such structures cannot be shared, since several messages cannot be sent on the same stream by different clients. However, several messages can be sent on the same port, which means that ports can be embedded.

- With naive binary merging of streams, message sending delay is variable. With ports, message sending delay is constant.

- Objects based on streams require that the streams are closed when the clients stop using them. This is similar to decrementing a reference counter, and has similar problems, besides being unnecessarily explicit and low-level. A port is automatically closed when there are no more potential senders, thus notifying the object consuming messages.

- Messages to an object can be sequenced. For this purpose, AKL provides the derived port operation *akl.send(M,P0,P1)*. The behavior of this operation is that *P1* is bound to *P0* after message *M* is added to the stream associated with *P0*.

AKL Built-In Port Objects

Object types in common use, such as arrays and hash tables, can be implemented as built-in types of ports, with a corresponding built-in treatment of messages. This may allow an efficient implementation of mutable data-structures, see [72]. Ports can also serve as interfaces to files in order to enhance I/O operations.

Arrays

AKL provides the definition of the built-in array port object in order to obtain array operations efficiently. The invocation *akl.new_array(Size, Array)* creates the object *Array* of type array with the given *Size* starting from *0* position. It is also possible to create arrays with a specific starting index by the invocation *akl.new_array(Size, Start, Array)*. The indices of the array run from the starting index plus the size minus one. The elements of the array are initialized to *[]*. To store *Value* at position *Index* in *Array* the message *akl.send(set(Index,Value), Array)* is used. To retrieve *Value* at position *Index* in *Array* the message *akl.send(get(Index,Value), Array)* is used. For more details we refer to [75].

Hash Tables

AKL provides the definition of the built-in hash table port object in order to obtain hash table operations efficiently. The invocation *akl.new_hash_table(HashTable)* creates the object *HashTable* of type hash table. It is also possible to create hash tables with an initial size by the invocation *akl.new_array(Size, Hashtable)*; the hash table automatically expands to accommodate further elements. To store *Value* at the key *Key* in *HashTable* the message *akl.send(set(Key,Value), HashTable)* is used. To retrieve *Value* at key *Key* in *HashTable* the message *akl.send(get(Key,Value), Array)* is used. For more details we refer to [75].

I/O

In AKL, ports play much the same role for input and output as "streams" in some other systems, e.g., most Prolog implementations. A port can be associated with a file or terminal, and messages can be sent to the port to obtain input or output operations. For more details we refer to [75]. For example, the following operations creates an I/O port,

- *io.stdin(Keyboard)*: creates a port *Keyboard* associated with the standard input device.
- *io.stdout(Screen)*: creates a port *Screen* associated with the standard output device.
- *io.fopen(File, Type, IOPort)*: creates a port *IOPort* associated with the file *File* that is opened for access type *Type*.

3.3.3 Data Objects

Besides port objects, a process based objects, AKL can also support data objects as a light-weight objects. AKL supports two types of data objects namely: immutable objects and procedure abstractions. These objects become garbage when their surrounding code, i.e. context, become also garbage.

Immutable objects are a special form of constructors of expressions, called *records*. They have a static state which can only be accessed, e.g. *line(StartPoint,EndPoint)* line segment in a graphics package. It provides a compact representation of objects the cost of which is similar to AKL data structures, see [75].

A procedure abstraction provides objects with a light-weight message sending the cost of which is similar to a procedure call.

Example 3.5 As an example for procedure abstraction objects, consider the following *abstraction* program that defines the representation of an object of a given type. This representation has a minimal form of encapsulated state, an embedded array cell. Message sending is a direct application to an abstraction. Also, messages can be sequenced.

```
:- module abstraction.
:- public dispatch/5.
:- public new/2.
:- public send/2.

new(Object,State,Class):=
    ( C,Msg,Flag: true →
      akl.new_array(1,C),
      akl.send(set(0,State),C),
      Object = (Msg,Flag)\abstraction.dispatch(Msg,Class,C,Object,Flag)).

dispatch(Msg,Class,C,Object,Flag):=
    ( State0,State1:akl.port(C) →
      Flag = Object,
      akl.send(set(0,State0,State1),C),
      akl.Class(Msg,State0,State1)).

send(Msg,Object0):=
    (true → akl.Object0(Msg,Object1)).

send(Msg,Object0,Object1):=
    (true → akl.Object0(Msg,Object1)).
```

□

Note, as functional languages, AKL allows the evaluation of recursive expressions [81].

3.4 Syntactic Sugar

Analogously to what is usually done for functional languages, we now introduce syntactic sugar that is convenient when the guards in choice statements consist mainly of pattern matching against the arguments, as is often the case.

A definition of the form

p(X1, ..., Xn) :=
    (g1 % b1
    ; ...
    ; g_k % b_k).

where % is either →, |, or *?*, may be broken up into several clauses

p(X1, ..., Xn) :– g1 % b1.
...
p(X1, ..., Xn) :– g_k % b_k.

which together stand for the above definition.

The main point of this transformation into clausal definitions is that the following additional syntactic sugar may be introduced, which will be exemplified below: (1) Free variables are implicitly hidden, but here the hiding statement encloses the right hand side of the clause (i.e., to the right of "*:–*"), and not the entire definition. (2) Equality constraints on the arguments in the guard part of a clause may be folded back into the heads *p(X1, ..., Xn)* of these clauses. (3) If the remainder of the guard is the null statement "*true*", it may be omitted. (4) If the guard is omitted and the guard operator is wait "*?*", it may also be omitted. (5) If the guard operator is omitted, and the body is the null statement "*true*", a clause may be abbreviated to a head.

As an example, the definition

member(X, Y) :=
    (Y1 : Y = [X|Y1] ? **true**
    ; X1, Y1 : Y = [X1|Y1] ? member(X, Y1)).

may be transformed to clauses

```
member(X, Y) :–
        Y = [X|Y1]
    ?   true.
member(X, Y) :–
        Y = [X1|Y1]
    ?   member(X, Y1).
```

where hiding is implicit according to (1). The equality constraints may then be folded back into the head according to (2), and the remaining null guards may be omitted according to (3), giving

```
member(X, [X|Y1]) :–
    ?   true.
member(X, [X1|Y1]) :–
    ?   member(X, Y1).
```

which may be further abbreviated to

```
member(X, [X|Y1]).
member(X, [X1|Y1]) :–
        member(X, Y1).
```

according to (4) and (5). We exemplify also with the append and merge definitions.

```
append([], Y, Z) :–
    →   Y = Z.
append(X, Y, X) :–
    →   X = [E|X1],
        Z = [E|Z1],
        append(X1, Y, Z1).
merge([], Y, Z) :–
    |   Y = Z.
merge(X, [], Z) :–
    |   X = Z.
merge([E|X], Y, Z) :–
    |   Z = [E|Z1],
        merge(X, Y, Z1).
merge(X, [E|Y], Z) :–
    |   Z = [E|Z1],
        merge(X, Y, Z1).
```

The examples should make it clear that some additional clarity is gained with the clausal syntax, which prevails in the logic programming community. A few additional remarks about the syntax is given below.

As syntactic sugar, the underscore symbol "_" may be used in place of a variable that has a single occurrence in a clause. All occurrences of "_" in a definition denote different variables.

A special syntactic sugar is provided for the common situation where a pair of arguments represent the input and output version of some changing data that is passed along from one procedure to another. An example is a program that accumulates items in a list:

```
items(Tree,L):-
    A = [],
    prelude(Tree,A,A1),
    contents(Tree,A1,A2),
    postlude(Tree,A2,A3),
    L = A3.
```

In order to reduce the proliferation of variable names, a preprocessor allows you to use the same variable name for all of the list variables as follows:

```
items(Tree,L):-
    A = [],
    prelude(Tree)-A,
    contents(Tree)-A,
    postlude(Tree)-A,
    L = A.
```

Here each *-A* represents a pair of variables which are inserted as arguments. The first variable in the pair are called the *input argument* and is syntactically equivalent to an occurrence of *A* at that point. The second is a new variable, called the *output argument*, and it is syntactically identified with *A* in the rest of the clause.

In an implementation of AKL, the character set restricts our syntax. The then symbol "→" is there written as "->". For example, append/3 would be written as

```
append([], Y, Z) :-
    -> Y = Z.
append(X, Y, Z) :-
    -> X = [E|X1],
       append(X1, Y, Z1),
       Z = [E|Z1].
```

However, to make programs as readable as possible, we will continue to use "→".

3.5 Summary

Conceptually, a concurrent constraint programming is an ideal vehicle for object-based programming. The modularity and simplicity of the object-oriented paradigm are very suitable for concurrent computing while constraints is very suitable to message sending.

AKL is a CCP language with deep guards [74]. The constraint system is an independent parameter of the language description. AKL offers a large potential for parallel execution. AKL is a *multiparadigm* programming language. The basic model is that of agents communicating over a constraint store allowing for different readings, depending on the context, where agents compute functions or relations, serve a user-defined constraints, or as object in object-oriented computing. A major point of AKL is that its paradigms can be combined. AKL provides constructs for don't care nondeterminism and don't know nondeterminism. From object-oriented view AKL greatly facilitates communication between objects through ports, a novel primitive in AKL, and provides means for light-weight representation of objects as higher-order abstraction of object methods. AKL objects is encapsulated and their message sending can be sequenced. AKL allows to use the clausal syntax for definitions to have the familiar reading of the logic programming community. AKL uses a flat, not hierarchical, module system with access control mechanisms for exporting methods.

As its name suggests, AKL is a programming language *kernel*. AKL supports the basic object-oriented style. This enables us to design and efficiently implement a complete language on top of AKL with a proper linguistic support and semantics. An incremental compilation facility is also designed and implemented.

⋆ ⋆ ★ ⋆ ⋆

4

AKL+

AKL+ is a concurrent object-oriented language based on the concepts of classes, generic classes, metaclasses, multiple inheritance, delegation and abstractions of classes and methods. Fig. 4.1 illustrates the AKL+ object-oriented system. It is built on top of the concurrent constraint language AKL. Classes, methods and instances of classes can be expressed as first-class values in the language which may be passed as arguments, returned as results and stored in attributes of objects. Objects are the run-time instances of classes, the behavior of these objects is defined by methods associated with classes. State is an object and this has the advantage that uniform access and encapsulation is achieved. The language provides for both higher-order and data-driven programming techniques.

The architecture of the language allows it to support object-oriented programming at two levels. The highest level consists of a set of standard classes for most common operations, such as creating objects. These classes act as the building blocks of AKL+ classes and are designed to meet the needs of most users. They provide a general-purpose object-oriented language that embody the default behavior of the language. The second level is the functional interface to AKL+, which allows the default behavior to be customized on which the programmer can build his applications.

In this chapter, we informally present AKL+ and its computation model. Several examples show the main features of AKL+ as well as a number of programming techniques. In Section 4.1 we describe the definition of classes and the implicit behavior provided by AKL+. In Section 4.2 we discuss objects, message passing, and synchronization. In Section 4.3 we describe the set of standard classes provided by AKL+. Section 4.4 is devoted to discuss our schemes to support synchronization constraints on message acceptance. In Section 4.5 we discuss our approach to handle multiple inheritance, differential inheritance, and inheritance anomaly.

4.1 Classes

The notion of a *class* is central to AKL+: every object is an instance[1] of a class. An AKL+ class is an object that determines the structure and behavior of the objects that are its instances. While every object is an instance of a class, it is not necessary, however, that a class has any instances. Thus, although AKL+ makes no distinction

[1] We use instance and object as synonyms.

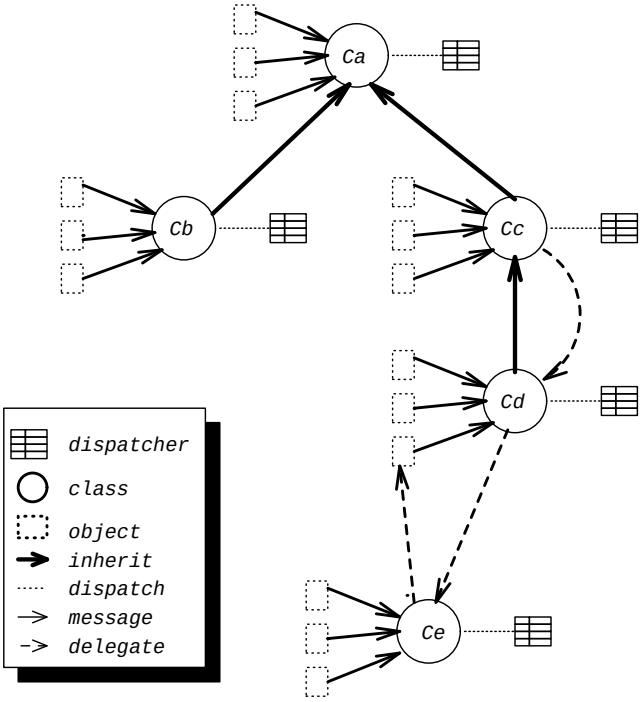

Figure 4.1: The AKL+ Object Oriented System

between classes that can and cannot be directly instantiated, it supports the use of *Mixins* classes [85, 54]– whose purpose is solely to augment the structure or behavior of other classes.

Classes are first-class objects and are themselves instances of classes. The class of a class is called *metaclass*. The representation of an instance is controlled by its derived metaclasses, which determines the actual storage structures that are used. By programming at the metaclass level, it is possible to define alternative representation for class instances.

4.1.1 Defining Classes

When a class is defined, a set of *superclasses* can be designated as classes from which that class is to inherit structure and behavior. A class whose definition specifies such a set of superclasses is said to be a *subclass* of each of those classes. The subclass-superclass relationship among classes, and hence the inheritance relationship, is transitive: a class inherits from its indirect as well as its direct classes. A class can be neither a direct nor an indirect superclass of itself. Thus, a class hierarchy forms a direct acyclic graph.

The *behavior* of instances of a class is determined by methods associated with that class and its superclasses. The globally visible methods determines its *interface*.

The state components of an instance is determined by attributes associated with its derived class as well as its superclasses.

In AKL+, a class is declared by writing it in the form

$$
\begin{array}{ll}
:- \textbf{class} & \langle class\ name \rangle. \\
:- \textbf{supers} & [\,\langle super_1 \rangle, \cdots, \langle super_a \rangle]. \\
:- \textbf{attributes} & [\,\langle attribute_1 \rangle, \cdots, \langle attribute_b \rangle\ ]. \\
:- \textbf{private} & [\,\langle selector_1 \rangle, \cdots, \langle selector_c \rangle\ ].
\end{array}
$$

$$
\begin{array}{l}
\langle method_1 \rangle. \\
\langle method_2 \rangle. \\
\ldots \\
\langle method_m \rangle.
\end{array}
$$

with the exception of the **class** declaration, all parts of the class definition are optional. The $\langle class\ name \rangle$ is an expression of the form $\langle identifier \rangle(X_1, \cdots, X_n)$, $n >= 0$, and $X_1, \cdots, X_n$ represent variables, when $n{=}0$ arguments are omitted. The $\langle identifier \rangle$ is an alpha-numeric symbol. $\langle method_1 \rangle, \cdots, \langle method_m \rangle, (m >= 0)$, are *method definitions*.

We can define a class in terms of an existing class. The new class is called the *subclass*, and the existing class is called the *superclass*. The superclasses of a class are the classes given in the **supers** declaration of the class definition. When a class C_1 is defined to be a subclass of another class C_2, objects of class C_1 have access, not only to the methods of C_1 but also to others inherited from C_2. Nevertheless, it is possible to be selective about what is inherited (directly or indirectly) from a superclass by writing the superclass in the following form

$$\langle super \rangle - [\langle selector_1 \rangle, \cdots, \langle selector_d \rangle]$$

where $\langle selector_i \rangle$[2] consists of the functor (name) and arity (number of arguments) of the excluded method that uniquely identify the method. Besides excluding a method from an inherited class, it is possible to hide the definition of a method and make it invisible. An invisible method of a class is the one that its selector is given in the **private** declaration.

AKL+ makes it possible to declare a list of attributes (data items), with their initializations. Attributes of a class are the attributes given in the **attributes** declaration. The attribute initialization is a method that computes the attribute's initial value. At object creation time, the attribute initialization methods is applied to determine the initial state of the object. The attribute declaration takes the form

[2]We use the Prolog functor/arity convention for identifying the method definition, so foo(X,Y) and foo(X,Y,Z) are distinguished.

$$\langle attribute_i \rangle = (X)\backslash\backslash \langle statement \rangle$$

where $\langle attribute_i \rangle$, an alpha-numeric symbol denotes an attribute name, will be assigned its initial value returned through the (output) argument X after applying $\langle statement \rangle$. Note that the definition of the initialization method is bound to an attribute's storage (state variable), tagged with the attribute name, which we call *method abstraction*. The difference between a class method and a method abstraction is that a class method belongs to a context, the class in which the definition textually appears, while a method abstraction has no statically bound context. A method abstraction takes the general form

$$M = (X_1, \ldots, X_n)\backslash\backslash \langle statement \rangle$$

and may have the syntactic sugar

$$M(X_1, \ldots, X_n) := \langle statement \rangle$$

The variables $X_1, \ldots, X_n$ must be different and are called *formal parameters*. These variables have a scope limited to $\langle statement \rangle$.

Not only we can define a method abstraction as first-class value but also we can define a class as first-class value. An abstraction of a class takes the form

$$X = \texttt{\#}\ \langle class\ name \rangle$$

such that the $\langle class\ name \rangle$ can be passed along to any method or class. Its methods can be activated by direct application to this parameter. Defining classes and methods as abstractions provides for all higher-order programming techniques. The class abstraction and dynamic binding greatly facilitates delegating to methods from outside of their context. Like class abstraction, method abstraction can be passed along as parameters. The difference between the application of the class abstraction and the application of the method abstraction is the requirement of a method name in the application of a class abstraction.

The behavior of an object is given by method definitions of the form

$$\langle identifier \rangle(X_1, \cdots, X_n) := \langle statement \rangle$$

The $\langle identifier \rangle(X_1, \cdots, X_n)$ is called the *head* of the method and $\langle statement \rangle$ is called its *body*. The ⟨identifier⟩ is an alpha-numeric symbol denotes the functor of the methods and n denotes its arity. The variables $X_1, \ldots, X_n$ must be different and are called *formal parameters*.

The method definition has the normal control structures of the AKL procedure definition (see Section 3.2):

- a *composition* statement of the form

  ⟨statement⟩, ..., ⟨statement⟩

  builds a composite agent from a sequence of agents. An agent correspond to statement being executed concurrently. Its behavior is to replace itself with the concurrently executing agents corresponding to its components.

- a *hiding* statement of the form

  X_1, ..., X_n : ⟨statement⟩

  introduces variables with local scope. The behavior of a hiding statement is to replace itself with its component statement, in which the variables X_1, ..., X_n have been replaced by new variables.

- a *choice* statement of the form

  (⟨statement⟩ % ⟨statement⟩
  ; ...
  ; ⟨statement⟩ % ⟨statement⟩)

  The symbol % is one of →, *?*, | and to these correspond *conditional* choice, *nondeterminate* choice, and *commit* choice of clauses, respectively. The components of the choice statement are called *(guarded) clauses*, the components of a clause *guard* and *body*, and a clause may be enclosed in hiding.

AKL+ inherits the concurrent constraint programming of its host language. The range of constraints that may be used in a program is defined by the current *constraint system*, which in AKL, in principle, may be any first-order theory. Atomic statements in method body are described below. Variables that appear in these statements are called *actual parameters*. Their occurrences in programs are sometimes referred to as *calls*.

Reserved variables. In AKL+, each method is executed in the context of a class, called "base class". This class may not be the class where the method is defined. The current contextual class is used to determine dynamically which methods are called. This leads to a mechanism called dynamic binding. Dynamic binding means that the association between a method call and the code executed has to be done at run-time [19]. Within a method, the base class is referred to by the reserved context variable *Self* and the state is referred to by the reserved variable *State*. *Self* is bound to the abstraction of the dispatch table of the base class which when applied on a method invokes its method definition. A base class is either a target object's class, the class of the object that received the message, or the called class of a method

invocation. *State* is bound to the current state of the object through which values of an attribute name can be updated, returned, or passed around.

Procedure call. From the object-oriented system perspective, calling a procedure in a module behaves like calling a method in a class. An invocation to the AKL procedure definition p/n which is defined in module m takes the form

$$m.p(X_1, \ldots, X_n)$$

Two kinds of method calls. Method definitions can be called in two ways: *method delegation* and *method invocation*. The difference between a method invocation and delegation is that the called class of the method invocation will be the base class of the invocation while the base class of the method delegation will depend on the context, denoted by the reserved variable *Self*, of the calling definition. This affects the dispatching of methods in case of base class application. In Simula [38], this notion of context is called the *qualification* of reference.

Method delegation. A delegation to a method definition takes the form

$$p(X_1, \ldots, X_n) \ \# \ q$$

where q is a class name. The method p/n may be defined in q or in any ancestor class of q. Within p/n, the *Self* variable will be bound to the same context as the calling definition. In other words, *Self* will be passed as is to the called definition. Delegation preserves *Self*.

Method invocation. An invocation of a method definition takes the form

$$p(X_1, \ldots, X_n) \ <\# \ q$$

The method p/n may be defined in q or in any ancestor class of q. Within p/n, the *Self* variable will be bound to the abstraction of the called class.

Class application. An application of the method p/n to the class abstraction Q takes the form

$$p(X_1, \ldots, X_n) \ \# \ Q$$

where Q is a variable that is to be bound to a class abstraction. A key feature is the possibility to call methods of the base class, so-called *base-class reference*, see Fig. 4.2. Simply, the base-class reference is an application of a method to *Self* which takes the form

$$p(X_1, \ldots, X_n) \ \# \ Self$$

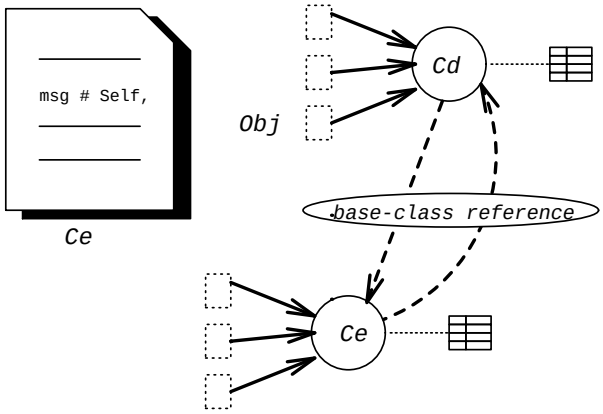

Figure 4.2: base-class reference

and may have the syntactic sugar

$$p(X_1, \ldots, X_n)$$

Method application. An application of a method abstraction bound to a variable Y, with the actual parameters $X_1, \ldots, X_n$ takes the form

$$Y(X_1, \ldots, X_n)$$

Message sending. A class can have a create method, e.g. *new*, for generating instances, see Section 4.3.2. An instance incorporates both the data representing its current state and has access to methods to perform its processing. Methods can send messages to other objects, or, using the *self* attribute, back to their target object, see Fig. 4.3. The message send takes the form

$$p(X_1, \ldots, X_n)\hat{}\, Object$$

At any point in time, the object holds a state called its *current state*. When an object is applied to a message, the object advances to a possibly new state by applying the method identified by the message.

$$State_0 \overset{message_1}{\longrightarrow} State_1 \overset{message_2}{\longrightarrow} \ldots \overset{message_n}{\longrightarrow} State_n$$

In AKL+, state is an object able to access and update attribute values only through messages. Consequently, encapsulation of state and uniform access are realized. Two standard classes are provided with AKL+: *state_array* and *state_hash*. The general behavior of these classes is described in Section 4.3.1. Each class can choose to inherit from any of them. For each attribute definition, a set of methods is implicitly defined. The behavior of these methods is discussed in Section 4.1.2.

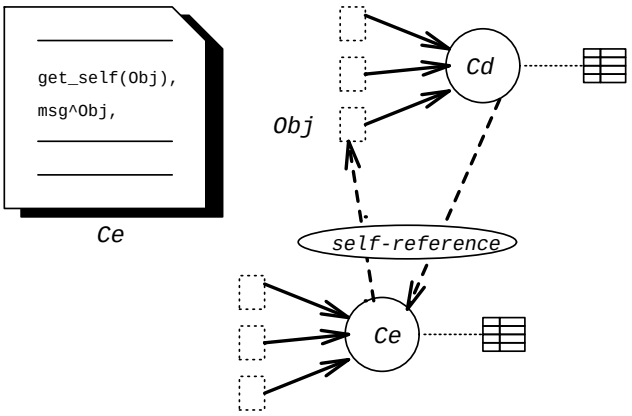

Figure 4.3: self-reference

The initial class. All classes derive directly or indirectly from an initial class that provides functionality as if it had at least the following definitions:

- a creator method for creating an instance of a given class.
- a destructor method that is to be activated when the instance is no longer in use.
- a state creator method for creating a state object.
- a state initializer method for determining the initial state of an instance at its creation time.
- an attribute *self* that refers to the target object.

In AKL+, *ur_object* is a predefined initial class which derived from the standard state class *state_hash* and the standard object type class *cell*, see Section 4.3. In this case, definitions that realizes the above functionality are given by the methods: *new/1* as a creator, *close/0* as a destructor, *new_state/1* as a state creator, and *initialize_attributes/3* as a state initializer.

It should be noted that users can also define and implement alternative state representation and object type classes for class instances by the virtue of programming at the meta level.

4.1.2 Attribute Methods

Many popular object-oriented languages, e.g. Smalltalk, Flavors [108], and Objective-C [120], allow free access to inherited instance variables by descendant classes, thus denying the designer the freedom to compatibly change the representation of a class

without affecting clients. It is encountering that several newer languages (CommonObjects [132], Trellis/Owl[125], and C++) correct this deficiency by restricting access to inherited instance variables. Where access to inherited instance variables is needed, it should be provided in the form of operations. In AKL+, The state is a first class-object which has a behavior describing how attributes are referenced, initialized, accessed, and updated. An example of a definition of a *counter* class follows.

Example 4.1 Consider the class *counter* with class *ur_object*, an initial class, as a superclass. Class *counter* defines the attribute *val* and the method *inc/0* that increments the current value of *val* by one. The prefixes *get_, set_,* and *init_* of an attribute name with arity one are chosen for attribute access, update, and initialize methods, respectively. Consequently, the attribute *val* is accessed, updated, and initialized by sending the messages *get_val/1*, *set_val/1*, and *init_val/1* to an object of class *counter*, respectively.

```
:- class counter.
:- supers [ur_object].
:- attributes [val=(V)\\(V=0)].

inc :=
    ( true → get_val(V),
      math.inc(V,V1),
      set_val(V1)).
```

□

The attributes definition of class *counter* implicitly defines the following methods:

- the attributes reference method *domain/1*. This method returns a list of all attribute names.
- the attribute initialization methods: *init_self/1* for the *self* attribute inherited from *ur_object* and *init_val/1* for the *val* attribute. For example, sending the message *init_val(X)* to the object *O* of class *counter* through:

  init_val(X)^ O

  binds *X* with *0*.

- the attribute property method: *attribute_property/4*. This method tabulates the method abstractions of initialize, access, and update methods of each attribute. An activation of this method given such an attribute will return these abstractions. An application of any of these abstractions to its argument will execute the corresponding definition. For example, sending the message *attribute_property(val,Init,Get,Set)* to the object *O* of class *counter* through:

  attribute_property(val,Init,Get,Set)^O

  binds *Init* with *(V)\\init_val(V)*, *Get* with *(V)\\get_val(V)*, and *Set* with *(V)\\set_val(V)*, respectively.

  The method *attribute_property/4* is very useful in defining generic attribute access and update methods where the attribute name may not known until run-time. Simply, the following is the definition of the methods *get/2* and *set/2* that realizes this behavior:

```
get(A,V):=
      ( true → attribute_property(A,_Init,Get,_Set),
        Get(V)).

set(A,V):=
      ( true → attribute_property(A,_Init,_Get,Set),
        Set(V)).
```

4.1.3 The Default Behavior

Sometimes it is useful to declare a class with a default behavior. Default behavior is a catch-all method. It is automatically invoked when the received message is not previously defined or inherited in the class. In AKL+, default behavior is provided in the form of *message not understood* and *user-defined default* methods.

Message not understood. For each class, this method is implicitly defined unless a user default method is defined or inherited. The *message not understood* method is very useful in exception handling. It reports that the message is not understood by the class that handles the message.

Example 4.2 Consider delegating the method *add(3)* from class *countUp* to the class *counter* when the base class is *countUP*

```
:- class countUp.
:- supers [ur_object].
```

```
...
add(3) # counter
...
```

This will report the following

> **Message not understood:** add(3)
> **Self:** countUp
> **Handler:** counter

which indicates that the message *add(3)* cannot be understood by class *counter*, the handler, when the base class, referred to by *Self*, was *countUp*. In other words, since the message *add/1* is not part of the counter interface, *counter* cannot serve this message. □

User-defined default method. The method *$default/1* is chosen to denote a user-defined default method. For example, the following default method delegates the unkown messages to the target object

```
'$default'(Msg) :=
    ( true → get_self(Obj),
      unkown(Msg)^Obj).
```

Or as a syntactic sugar, the user-defined default method can be recognized when it has as its head an AKL+ variable.

4.1.4 The Class Membership Method

A definition of the method *typeof/1* is implicitly defined for each class definition. This method is used to determine the identity of the class of an object. In other words, the class membership property that belongs objects to their classes can be asked about by sending these objects the message *typeof/1*. For example, sending the message *typeof(Type)* to the object *C* of class *counter* will bind *Type* to *counter*. The support of this method by AKL+ is very useful. A main usage of this method, is the possibility of defining *class-specific methods* [15]. When a class-specific method is invoked the appropriate method is executed on the basis of the identity of the target classes.

Example 4.3 consider the following definitions for classes representing geometric solids:

```
:- class solid.
:- supers [ur_object].

:- class sphere.
:- supers [solid].
:- attributes [radius=(V)\\(V=0)].

:- class cube.
:- supers [solid].
:- attributes [edge=(V)\\(V=0)].

:- class cone.
:- supers [solid].
:- attributes [radius=(V)\\(V=0),height=(V)\\(V=0)].
```

We might define class-specific behavior for *spheres*, *cubes* and *cones* as follows:

```
volume(V):=
    ( true → get_self(GeometricSolid),
      typeof(ClassType)^GeometricSolid,
      volume(ClassType,V,GeometricSolid)).

volume(SolidClass,V,GeometricSolid):=
    ( SolidClass = sphere | get_radius(R)^GeometricSolid,
      V is 4/3*3.14*R*R*R
    ; SolidClass = cube | get_edge(E)^GeometricSolid,
      V is E*E*E
    ; SolidClass = cone | get_radius(R)^GeometricSolid,
      get_height(H)^GeometricSolid,
      V is 3.14*R*R*H/3).
```

These method definitions result in the creation of the class-specific method *volume/[1-3]*. When this method is applied to an instance of *sphere, cube*, or *cone*, the appropriate method is selected and called. □

4.1.5 Method dispatcher

A definition of the method dispatcher is implicitly defined for each class definition. The clauses of the method dispatcher is the entry point to the method handler. When a message M, is sent to object O of class C, we apply the method *dispatch(M)* on class C.

Example 4.4 The following is the definition of *dispatch/1* for class *counter*

```
dispatch(Msg) :=
    ( Msg = attribute_property(_,_,_,_) → Msg # counter
    ; Msg = domain(_) → Msg # counter
    ; Msg = get_self(_) → Msg # counter
    ; Msg = inc → Msg # counter
    ; Msg = init_self(_) → Msg # counter
    ; Msg = set_self(_) → Msg # counter
    ; Msg = typeof(_) → Msg # counter
    ; Msg = initialize_attributes(_,_,_) → Msg # state_hash
    ; Msg = new_state(_) → Msg # state_hash
    ; Msg = new(_) → Msg # ur_object
    ; true → typeof(Class),
      akl.stdout(S),
      io.format('~nMessage not understood: ~w ~nSelf: ~w ~nHandler: ~w ~n',
      [Msg,Class,counter],S,_)) .
```

□

The method dispatcher is discussed in Section 5.3 and Section 6.3.2.

4.1.6 Generic Classes

A class gains a generic property by associating it with parameters. The parameter passing of a class parameter is not resolved at compile time and all references are bound at run-time. In this sense a parameterized class is considered a *generic class*. The scope of this parameter is the class methods. A parameter of a parameterized class lies in four categories: class abstraction, constant, object, or method abstraction. There are situations in which we can make good use of parameterized classes:

- an inheriting class may inherit from the same class in several different ways, using different parameters.
- the farther of passing a parameter up in the class hierarchy the more general the class hierarchy is.
- when the parameter is a class abstraction, both objects of this class can be created and methods of this class can be invoked in the parameterized class methods.
- the dynamic binding of class parameters makes it possible to execute different code at different points in time.

Example 4.5 A good example for showing the usefulness of generic classes is a general sorting algorithm. Consider the following definition of a class *sorter* that sorts a list of elements into an *ascending* order:

```
:- class sorter.
:- supers [ur_object].
:- private [insert/3].

sort(L0, L):=
    ( L0 = [] → L = []
    ; L0 = [El |List] → insert(El, List1, L),
      sort(List, List1)).

insert(El, L0, L):=
    ( L0 = [] → L = [El]
    ; L0 = [E | L1] ,
      El < E → L = [El,E | L1]
    ; L0 = [E | L1] → L = [E | L2],
      insert(El, L1, L2)).
```

Imagine different classes that needs sorted list according to several criteria such as ascending, descending, cartesian product of two domains, and so on. Sorting the list with respect to any criterion only differs in the way the elements of the list are compared. So, good software design is to write one general sorting algorithm and several comparing algorithms and pass the appropriate comparison for each sorting application. Using a parameterized *sorter* class it is possible to generalize the above sorting algorithm in four different ways. Figure 4.4 shows that the alternatives *a*, *b*, and *c* require to ensure a naming convention of the desired method or operator in both the general algorithm (*sorter*) and the passed parameter. However, this is not the case with alternative *d* since we pass the abstracted comparing method and all is to be done is to apply this method without the need to have such a naming convention. The designer is free to choose which alternative deems appropriate to his application.

Note that passing the comparing function to the sorting algorithm is one way to emulate higher order functions [90]. □

Generic classes are interested in its own right. The generality of classes is increased through the addition of parameters [114], so that various instantiations of a class parameter correspond to different uses. Applications can be "tuned" by choosing different parameter values. Moreover, They provide contextual variables without having to add such variables as additional context arguments to each method clause that potentially uses or passes these arguments.

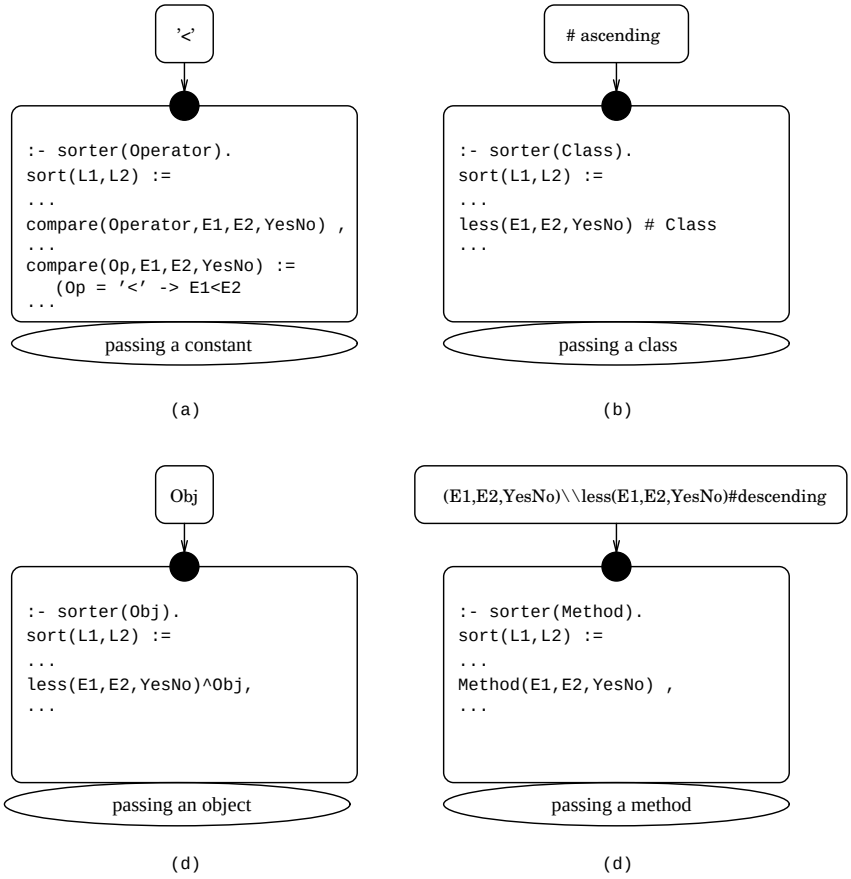

Figure 4.4: The passing of parameters to a parameterized class

4.1.7 Metaclasses

Earlier discussions described objects as instances of classes. A class contains the description of the structure and behavior of its instances. In most object-oriented languages, classes are factories that create and initialize instances. For example, an instance of class *counter* can be created through:

new(Counter) <**#** counter

Here a class is treated as an object that can instantiate (create instances) other objects. Therefore, there are two types of objects:

1. *class objects*–objects that can act as templates and create instances of themselves.

2. *terminal objects*–objects that can only be instantiated but cannot instantiate other objects.

If the class describes the attributes and methods that are applicable to its instances, who or what class describes the state and the general behavior (object type) of the class as an object? More specifically, what is the class of the class?

MetaClass is a class that describes the characteristics of another class. Several, but not all, object-oriented languages support the concept of a metaclass, for example, C++ doesn't support metaclasses. AKL+ supports metaclasses for two purposes: creating object types and creating the object's state.

In supporting metaclasses, existing object-oriented languages pursue either [84]:

1. *Explicit support for creating and instantiating metaclasses.* For example, ObjVlisp [35]. This object-oriented language treats objects, classes, and metaclasses as objects. In ObjVlisp, the user takes the burden of instantiating classes and metaclasses in two separate steps; to manipulate three levels of objects.

2. *Implicit support of metaclasses.* Smalltalk was the first object-oriented language to introduce metaclasses. They are hidden form the user. In Smalltalk, metaclasses cannot be declared and created explicitly. Metaclasses are anonymous and *do not* have an existence independent of their classes. Thus there is one-to-one correspondence between classes and metaclasses and the definition of metaclasses is intertwined in the definition of the class.

There are at least two advantages in treating classes as objects in AKL+. The first is that classes can be used for storing group information. If a class is treated as an object, then information global to all the attributes of the class can be stored in the state class. Access and update methods associated with the class can be used to retrieve or update values of the class attributes. For example, discussed in Section 4.3, the classes *state_hash* and *state_array* are important pieces of global information that can be used by instances or other clients of the class.

The second advantage (and most common usage) of class objects is their use in creation of new instances of the class, see Section 4.2.2. The message *new/1* which may be sent to class objects to create instances can do the initialization of the attributes values of the newly created instance.

4.2 Objects

Objects are independent entities which are capable of responding to a request for processing. Objects are dynamically created during run-time. An object must incorporate both the data representing its current state and has access to the methods to perform its processing. An object should be encapsulated such that it is accessible only via an interface supporting its defined operations, see Fig. 4.5.

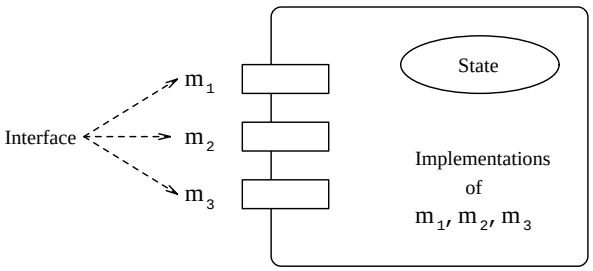

Figure 4.5: An object

4.2.1 Message Sending vs Method Call

Objects are concurrent agents that communicate with each other by sending messages. The behavior of an object is described in a class by a set of operations called methods. An object has a state containing attribute/value pairs that may change over time. A method is executed if the object receives a message. Methods are definitions that provide the functional abstraction of the state transition:

$$\boxed{\mathcal{M}ethod : message \times Self \times state \rightarrow state}$$

Fig. 4.6 illustrates an object receiving three messages out of its funnel. The object applies the appropriate method to its state, dispatches the method, resulting in a new state. The reserved context variable *Self* is bound to the abstraction of the base class. Thus we can set the value of the attribute *val* for an object *O* of class *counter* to *100* by sending the message:

set_val(100)^O

This message sending selects the method of *O* with the functor *set_val* and arity one. This method is applied to *Self*, *O*'s current state, resulting in a state whose attribute *val* is of value *100*. Each method has in addition to *Self* another two reserved context variables for the incoming and the outgoing state indicated by the accumulator pair [3] *State*. The method execution results in applying to the current state the method that matches the method functor and arity. The state that results from this application is used for further computation. A method clause maintains a threaded state in order to keep the consistency of the object's internal state. Each method serializes the method calls on the availability of an object's state. By state threading, the state is released when the last method call in a method's body has been promoted. Hence, only one of the state-using sections in a method body can at the same time be entered. An alternative to the method call is the *self-reference* message sending. The effect, however, is quite different. The method call is performed immediately

[3] An accumulator pair is a syntactic variable that indicates a pair of arguments. The first argument is called the *input argument* and the second is the called the *output argument*.

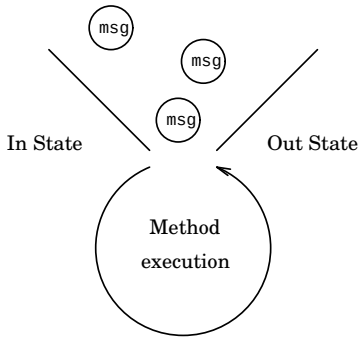

Figure 4.6: The method execution

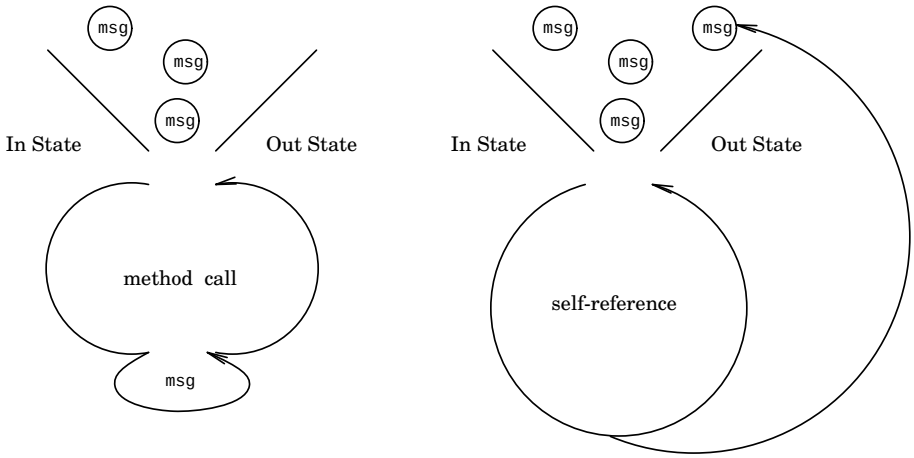

Figure 4.7: Method call vs self-reference

on the current state, whereas other messages may be taken and change the state before the *self-reference* message is received. Fig. 4.7 depicts the difference.

Messages to objects are always dynamically bound since objects are dynamically created and will not be known until run-time. A method call to a named class is statically bound and invoking a method in this manner is more efficient than invocations relying on dynamic binding. Dynamic binding, also called *late* binding, is one of the most frequently cited advantages of the object-oriented style of programming. Dynamic binding means the system binds message selectors to the methods that implement them at run-time (instead of compile time). The particular method used in the binding depends on the context, the current base class. This provides us with the ability to execute different code at different points in time. An important use of dynamic binding in object-oriented languages is the ability to apply methods of the base class and send messages back to the target object.

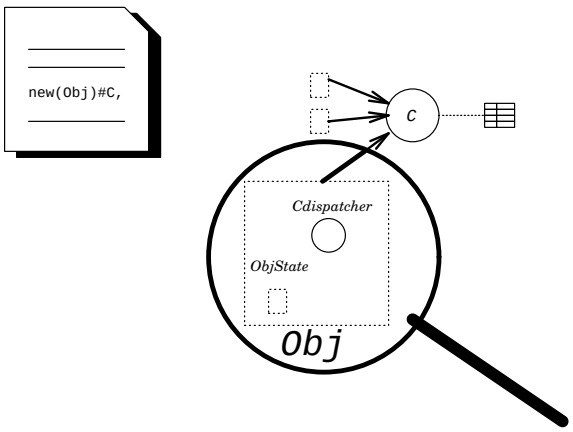

Figure 4.8: Creating an object

4.2.2 Creating and Destroying Objects

Like many other object-oriented languages, objects (instances) of a class are created through invoking the method *new/1*, à la Smalltalk, to this class. The *new/1* method creates a new object and sends an initialization message to its state object before handing it back to the caller, who requested the new object. Fig. 4.8 illustrates creating an object *Obj* of class *C*. *Obj* has an encapsulated state as well as a context, an abstraction of its base class *C*. For example, the following statement will create a newly initialized object of class *counter*. This object has a type of class *cell* and state of class *state_hash*.

```
new(Counter) <# counter
```

An object refers to itself through the attribute *self*. The value of this attribute can be used for *self-reference* sort of message passing.

Not only classes can create instances of its own but also it can create instances of other classes. This powerful features allows the class to create instances from different contexts.

Example 4.6 Consider the following declaration of class *car_washer*. Its attribute *car_queue* is initialized to an instance of class *resource* with a context *resource* rather than *car_washer*. Hence, all invocations on the base class through the attribute *car_queue* is executed in the context of *resource*.

```
:- class car_washer(NoOfWorkers,ServiceTime,Out,Scheduler).
:- supers [simulation(Scheduler)].
:- attributes [car_queue=(Q)\\(new(Q)<# resource(Scheduler),
                avail_workers=(N)\\(N=NoOfWorkers)]).
   ...
```

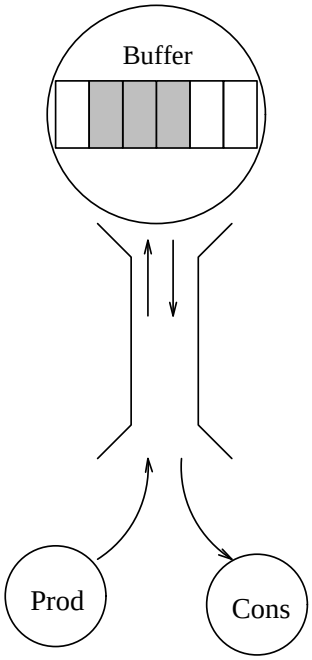

Figure 4.9: An example of object sharing

Note that "method invocation" of *new/1* is necessary in order to create an object whose base class is *resource*. □

Destroying an Object. Similar to Smalltalk, an AKL+ object becomes garbage and storage is reclaimed, if there are no more objects or variables referencing an object. It is also possible to explicitly dispose an object when no longer needed, e.g. objects derived from the standard class *object*, by sending the message *close/0* to it. In this sense objects in AKL+ greatly simulate real life objects.

4.2.3 Sharing Objects

With object sharing, multiple applications or objects share a common object [40]. Object sharing is realized in AKL+ by passing an object as a parameter to classes at creation time of their instances. For example, consider the class *buffer_sync*, see Example 4.7, that defines a buffer object with two operations *put/1* to store an item and *get/1* to remove an item, see Fig. 4.9. To create, a shared buffer object *Buffer* of class *buffer_sync* to the objects *Prod* and *Cons* of classes producer and consumer, respectively, we write the composition

```
...
new(Buffer) <# buffer_sync(3),
new(Prod) <# producer(Buffer),
new(Cons) <# consumer(Buffer),
```

...

At any time, *Prod* object can produce an item by sending the message *put/1* to the object *Buffer* while the *Cons* object can remove an item by sending the message *get/1* to the same object *Buffer*.

4.2.4 Synchronization

In object-oriented concurrent computing every object runs concurrently with others. An interaction between objects is divided into two parts: *information exchange* and *synchronization* [160]. Information exchange is to influence other object. Synchronization is to control the timing of information exchange among objects and to control the order of object execution. Thus, in order to perform coordinated computing, objects must be able to synchronize. AKL+ is very flexible to support various types of synchronization. AKL+ provides the following basic mechanisms for synchronization of concurrent activities:

1. *Messages synchronized by passing them in a batch.* With batch, we can send several messages to an object via a single message sending. A message batch guarantees that the elements are executed in the given order without intervening messages sent from elsewhere. For example, if you want to guarantee that the object C of class *counter* is set to *0*, incremented twice, and then ask for its value without giving someone else the chance to access C, you write

   send_batch([set_val(0),inc,inc,get_val(X)])^C

   which is guaranteed to bind X with *2*. In AKL+, The method *send_batch/1* is provided to process a message batch. This method is defined in the standard class *object*.

   In a concurrent framework this ability to specify a block of messages to be processed is quite useful in real life [80]. For example, to transfer an amount of funds between two accounts, the withdrawal will be processed just before the deposit.

2. *Messages synchronized by serializing them.* With message serialization, the messages are connected into a chain which preserves their relative order. However, it does not guarantee that some other messages may come between any two messages. This technique of synchronizing messages sent to an object is similar to the constructor *akl.send/3* used for sending messages to the AKL port primitive. For example, if we want to serialize sending the messages *set_val(0),inc,inc,* and, *get_val(X)* to the object C of class *counter*, we may write the composition

set_val(0)^C,
inc^C,
inc^C,
get_val(X)^C,
...etc...

This will bind X with *2* only if there is no other message that changes the state of C is executed before the execution of the message *get_val(X)^C*.

Without the ability to serialize messages to an object, it is difficult to model even simple things as concurrent objects. For example, a computer terminal or displaying device because the order of text lines which are sent by a terminal handling program must be preserved when they are received.

3. *objects synchronized by the exchange of messages between a sender and a receiver objects.* The basic technique is illustrated as follows:

```
:- class acknowledge(Receiver).
:- supers [ur_object].
   ...
   get_self(Sender),
   callback(()\\(ack(Ack,InfoOut)^Sender),InfoIn)^Receiver
   ...
   ack(Ack,Reply) :=
   (Ack = ok →
   ...
```

The idea is that the sender of a message sends the message *callback* to the receiver with the message abstracting the acknowledge from the receiver and suspends until a call back is received. The identity of the sender is held in the abstraction such that when the receiver applies the abstraction, the acknowledge is executed on the sender side.

4. *Objects Synchronized by satisfying constraints on the acceptance of their messages.* When a concurrent object is in a certain state, it can accept only a subset of its entire set of messages in order to maintain its internal integrity. Such a restriction on acceptable messages is called the *synchronization constraints* of the concurrent object [97]. This synchronization mechanism is discussed in Section 4.4.

4.3 Standard Classes

In AKL+ a set of comprehensive standard classes define the default behavior of the system. These classes are the most general classes. They are provided for most common operations, such as creating an object. These classes act as the building blocks of AKL+ classes which are designed to meet the needs of most programmers. AKL+ standard classes are classified into two categories: state representation standard classes and object type standard classes. The objects created by the standard classes are first-class values: they may be created interactively at any time by a program, passed around as arguments, returned as values, and stored in the attributes of objects. The programmer of the language can use, specialize, or extend some of these classes to define an initial class for the class hierarchy at the topmost level of the class hierarchy. Fig. 4.10 depicts a conceptual view of an initial class. For example, the aforementioned class *ur_object* is an initial class with the standard superclasses *cell* and *state_hash*. It has the attribute *self* that can be used for delegating messages back to the target object. That is, when an object *Obj* receives a message, say *message*, sending messages to *Obj* during the course of executing the message *message* is through retrieving the value of the attribute *self*, then sending the desired message to this value. The following is the definition of *ur_object*

```
:- class ur_object.
:- supers [cell,state_hash].
:- attributes [self=(V)\\(V=nil)].

new(Obj) :=
    ( true → new(Obj) # cell,
      set_self(Obj) ^ Obj).
```

4.3.1 State Representation Classes

State class is a model or pattern that defines the creation of state object and the initialization of attributes. State object is a simple object able to access and update attribute values. Attributes in AKL+ are placeholders to data-items. Attributes of a class are the attributes given in the **attributes** declaration. The attribute initialization is specified by a method abstraction that returns the attribute's initial value. It is often not satisfactory that objects are created without initializing them. After creating an object, the message sending to get any of its attribute value results in an object error, since the attribute has no value. Even always sending an initializing message like in

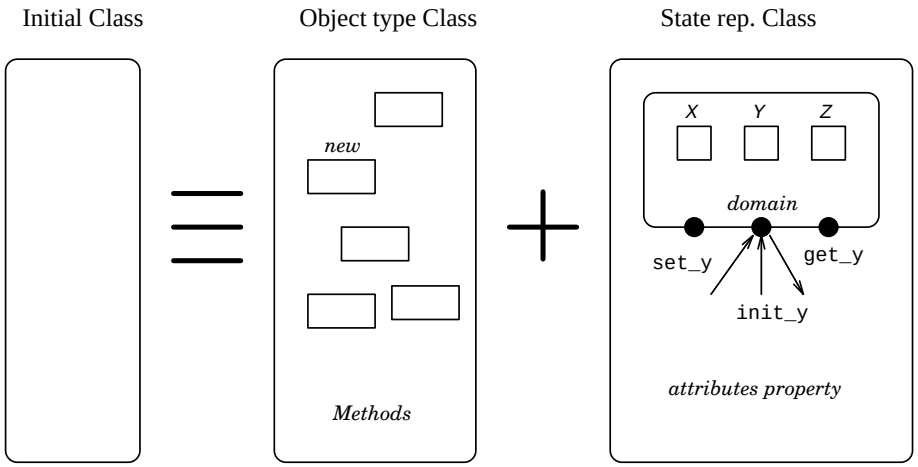

Figure 4.10: A conceptual view of an initial class

set_attribute(InitialValue)^O

does not help in general since concurrent execution does not prevent other messages from being received before *set_attribute(InitialValue)*. Therefore, in AKL+, attributes are initialized with object creation.

AKL+ provided two standard state classes: the *state_hash* class and *state_array* class as important pieces of global information that can be used by instances or other clients of the class. These standard classes greatly improve the efficiency in manipulating attributes.

The state_hash class. This class is provided in order to obtain hash table operations for the state. This representation is based on the built-in *hash table* port object of AKL that supports these operations efficiently. Each attribute will be a key in a hash table and each key's value will be the value of the corresponding attribute. The *state_hash* class provides a considerable efficiency in the constant (key/value) access of the attributes values. The following is the definition of *state_hash* standard class:

```
:- class state_hash.

new_state(S):=
    ( true → domain(Attributes),
      list.length(Attributes,Size),
      akl.new_hash_table(Size,HashTable),
      initialize_attributes(Attributes)–HashTable,
      S = HashTable).

initialize_attributes(Attributes)–HashTable:=
    ( Attributes = [] → true
```

```
; Attributes = [A|As] → attribute_property(A,Init,_Get,Set),
  Init(V),
  akl.Set(V,Self)–HashTable,
  initialize_attributes(As)–HashTable).
```

where:

- *new_state/1* is the method defined to create an empty hash table state object. The empty state is initialized such that the returned object from *new_state/1* will be a newly initialized state object.
- *initialize_attributes/3* is the method defined to initialize attributes. For each attribute, the *attribute_property/4* is invoked to determine its initialize and update methods. The attribute's initial value returned from the initialize method is passed to the update method to set the attribute's initial value.

The methods *get/2* and *set/2*, discussed in Section, 4.1.2 can be defined efficiently in terms of *state_hash* objects as follows:

```
set(Att,V) :=
    (true → set(Att,V)^State).

get(Att,V) :=
    (true → get(Att,V)^State).
```

The state_array class. This class is provided in order to obtain array operations for the state. This representation is based on the built-in *array* port object of AKL that supports these operations efficiently. The system automatically generates an array index for each attribute. The prefix *attribute_* of an attribute name with arity one is chosen for the method that returns this index. For example, the index of the attribute *val* can be returned through the method *attribute_val/1*. The value of an attribute is the element stored at its index in the array. The *state_array* class is an alternative representation of state to *state_hash* class. The *state_array* class provides a considerable efficiency in the constant (indexed) access of the attributes values. The following is the definition of the *state_array* standard class:

```
:- class state_array.

new_state(S):=
    ( true → domain(Attributes),
      list.length(Attributes,Size),
      akl.new_array(Size,Array),
```

```
      initialize_attributes(Attributes)–Array,
      S = Array).

initialize_attributes(Attributes)–Array:=
    ( Attributes = [] → true
    ; Attributes = [A|As] → attribute_property(A,Init,_Get,Set),
      Init(V),
      akl.Set(V,Self)–Array,
      initialize_attributes(As)–Array).
```

4.3.2 Object type Classes

AKL+ supports the basic object-oriented style by allowing the definition of both *port* based objects and *cell* based objects as an abstraction for describing the concurrent activity. A *port object* is a process that acts as a consumer of a communication medium while a *cell object* is a data structure. A *port object* is an active object which is a medium linking the sender with the receiver through a stream. The sender leaves its message on the stream and proceeds. This incurs an inevitable memory overhead, in this case a list construction. Clearly, for light-weigh objects, such as window components, one would prefer to execute the appropriate method directly on the sender's side without suspension or scheduling the message to the object. For the efficiency reasons, AKL+ supports cell based objects as a very fine-grain (light-weight) objects and switch to port objects only for heavy weight objects.

AKL+ provides three standard object type classes: the *object* class for creating port objects, the *cell* class for creating objects as data-structure, and a *sync_object* class for synchronizing the acceptance of messages to the objects. These standard classes create objects with encapsulated state.

The object class. This class is based on the AKL port objects, see Section 3.3.2. The following definition exhibits the behavior of this class:

```
:- class object.

close:=
    ( true → akl.stdout(S),
      io.format('Object Terminated ~n')–S).

send_batch(MsgList):=
    ( MsgList = [] → true
    ; MsgList = [Message|Rest] → Message,
      send_batch(Rest)).
```

```
new(Object):=
    ( true → akl.open_port(Object,Stream),
      new_state(S),
      O = (M)\\(object(M) # object),
      akl.O(Stream,Self)–S).

object(Stream) :=
    ( Stream = [] → close
    ; Stream = [close|_Stream] → close
    ; Stream = [Message|Next] → Message,
      object(Next) # object).
```

where:

- *new/1* is the method defined to create a newly initialized port object. To set a communication medium, a stream is created.
- *object/1* is the method defined for processing a stream of messages. When an object receives a message, the appropriate method from its dispatcher is applied to its state, resulting in a new state. The new state will be used for further computation.
- *send_batch/1* is the method defined for processing a batch of messages in a consecutive order. These messages will be executed in a non interleaved manner. Sending a batch of messages is discussed in Section 4.2.4.
- *close/0* is the method defined to report the termination of the port object.

The cell class. This class defines a representation of a concurrent construct that provides a minimal form of encapsulated state. This representation suffices to efficiently express a very fine-grain (light-weight) objects in AKL+. The following definition exhibits the behavior of the *cell* class:

```
:- class cell.
new(Object) :=
    ( true → new_state(S),
      meta.cell(C),
      set(0,S)^C,
      meta.create_cell_object(C,Self,Object)).
```

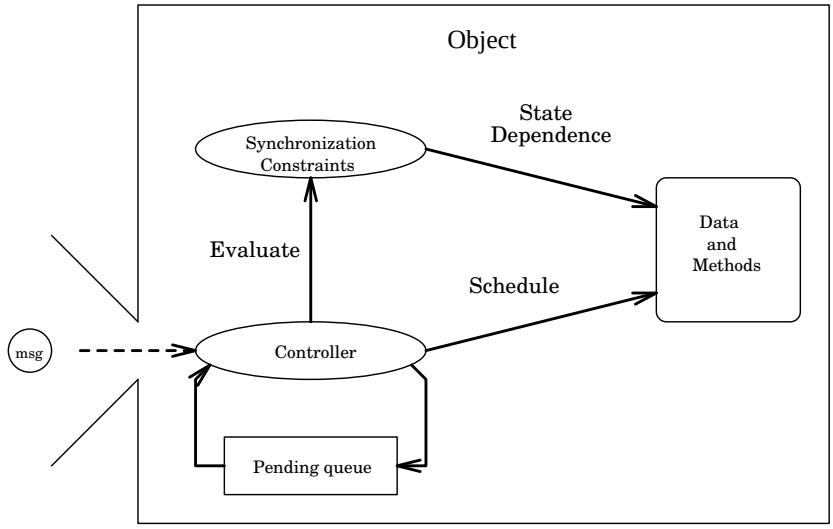

Figure 4.11: The structure of a synchronized object

The method *new/1* defines the creation of cell objects. Cell object is an abstraction with a state which is able to synchronize its operations. The method *new/1* creates a new state S and a memory cell C. The state S is encapsulated within the memory cell C. Then, an abstraction is created to hold the memory cell and the class dispatcher. The primitive operations of the AKL+ cell construct are defined in module *meta*, discussed in Section 6.5

The sync_object class. This class is provided for specifying objects that require satisfying constraints on its acceptable messages, and characterize the behavior of an object with respect to message acceptance. The receiving object may be of any other two object types described above. For example, the following definition combines this behavior with the standard class *object*.

```
:- class sync_object.
:- attributes [message_queue=(V)\\(V=[])].

new(Object):=
    ( true → akl.open_port(Object,Stream),
      new_state(S),
      O = (M)\\(object(M) # sync_object),
      akl.O(Stream,Self)–S).

object(Stream) :=
    ( Stream = [] → close
    ; Stream = [close|_Next] → close
    ; Stream = [Message|Next] → sync(Message)–Next,
      object(Next) # sync_object).
```

```
wake(MethodSet,NewMethodSet,Q,NewQ,Stream,Next):=
    ( MethodSet = NewMethodSet →
      NewQ = Q,
      Next = Stream
    ; true → collect_wake(NewMethodSet,Q,NewQ,Stream,Next)).

collect_wake(NewMethodSet,Q,NewQ,Stream,Next):=
    ( Q = [] → NewQ = [],
      Next = Stream
    ; Q = [Message|QRest],
      akl.record_name(Message,F),
      akl.record_size(Message,N),
      sets.set_member(F/N,NewMethodSet) →
      Stream1 = [Message|Stream],
      collect_wake(NewMethodSet,QRest,NewQ,Stream1,Next)
    ; Q = [Message|QRest] →
      NewQ = [Message|NewQs],
      collect_wake(NewMethodSet,QRest,NewQs,Stream,Next)).

close:=
    ( true → get_message_queue(Q),
      close(Q)).

close(Q):=
    ( Q = [] → akl.stdout(S),
      io.format('** Object Terminated successfully **~n')–S
    ; true →
      io.format('Object Terminated–the following messages not processed~n~w ~n',[Q])).
```

where:

- *new/1* is the method defined to create a newly initialized synchronized object. By synchronized object, we mean object that can accept or delay a message according to a synchronization constraints on its message acceptance.

- *object/1* is the method defined for processing a stream of messages. When an object receives a message, it invokes the *controller*, see Fig. 4.11, which makes sure that the incoming request are scheduled according to the synchronization constraints of the object. If an invocation request cannot be scheduled right away, it is put into a pending queue (*message_queue*) by the controller. The controller may attempt to (re)schedule invocation requests in the pending queue at a later point in time (e.g. after each method invocation).

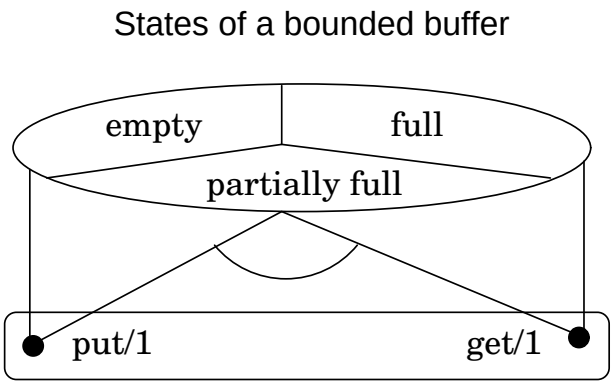

Figure 4.12: Message acceptance of a bounded buffer

- *wake/6* is the method defined for (re)scheduling the invocation requests when the acceptance condition is changed.
- *collect_wake/5* is the method defined for collecting the (re)scheduled requests from the pending queue.
- *close/[0-1]* is method defined to report the termination of the object.

The basic synchronization schemes is discussed in Section 4.2.4 and the synchronization of message acceptance is discussed in 4.4.

4.4 Synchronization Constraints

Synchronization constraints is a mechanism used for maintaining the internal integrity of the concurrent object. It allows an object to accept only a subset of its messages in a certain state. A well-known example is a bounded buffer with methods *put/1* and *get/1*, where *put/1* stores an item in the buffer and *get/1* removes the oldest one; then the synchronization is that one cannot *get/1* from a buffer whose state is *empty* and cannot *put/1* into a buffer whose state *full* is likewise prohibited, see Fig. 4.12. The satisfaction of constraints is not achieved automatically; the user must somehow program the methods to implement the object behavior that satisfy the synchronization constraints. A method that may be invoked according to the synchronization constraints is said to be *enabled*. *Synchronization code* is the portion of code where such synchronization is controlled.

In order to program synchronization code, a concurrent object-oriented language provides some primitives and/or general schemes for object-wise synchronization, such as method guards [36]. In [98], the scheme for achieving object-wise synchronization using those primitives in the language is called *synchronization scheme* of the language.

Inheritance is the key structuring mechanism in object-oriented languages. In concurrent object-oriented languages, it is desirable to inherit synchronization constraints whenever possible, to avoid reimplementation of superclass synchronization constraints in subclasses [52]. However, the possibility of inheriting the synchronization code may become difficult and result in restricting the reusability. This problem and our proposal to its solution are discussed in Section 4.5.3.

The main advantage to the synchronization constraints schemes in AKL+ is the clean separation of concurrency control and the method specification such that they can be inherited, overridden, or extended separately without affecting each other. One schemes can be integrated and composed with other schemes.

AKL+ provides two types of synchronization schemes: *synchronizers* and *transitions* comparable to Matsuoka's synchronization schemes [98]. The programmer writes down explicit specification of what we call the *accept method set*. The accept method set is an enumeration of the methods whose invocations do not violate the assertion that must hold for a given state of the object. That is, the set of method selectors (the method functor and its arity) of the acceptable methods of each state. All the states must be identifiable within the synchronization code of classes, with which users control the accept set depending on each state. Only one accept method set will be enabled for execution in the next state of the object. The programmer defines the *synchronizer/transition* that specifies the synchronization constraints that is used to compute the next accept method set. An occurrence of a message selector as an element of this set means that its method is enabled. A message not satisfying the constraints is buffered until such time when the object's state satisfies the constraints.

4.4.1 Accept Method Set

The accept method set is a finite set of method selectors for an instance of a class, for which the associated methods are acceptable for its current state. Each set naturally satisfies the following condition: for any element in the set, the method corresponding to the element is already defined at that class or its superclasses. Each method can be associated with an identifier, for example, the accept method set of the bounded buffer when it becomes *full* is {*get/1*}. The method *mset/2* is chosen to define the accept method set of class instances.

Primitive accept method set construct has the form *[Selector, ...]* . There are also some primitive set operations, such as union, difference, etc. For example, the *partially full* state of the bounded buffer is the union of the *empty* set and the *full* set. The current accept method set of an object can be obtained through retrieving the value of the attribute *mset*. The initial value of this attribute is specified by:

mset(SetId, MSet):=

```
(SetId = initially →
...).
```

where the (output) argument *MSet* will return the initial accept method set and *initially* is a special keyword that is used to indicate which accept method set is enabled initially upon object creation.

4.4.2 Synchronizers

A *synchronizer* is a combination of a guard specification (an activation condition for a method), enabling specifier and a list of accept method sets. In essence, this synchronization scheme is similar to a guarded method but is more flexible in that a single guard can be assigned to multiple methods in the accept method sets. The following definition exhibits the behavior of the standard class *synchronizers*:

```
:- class synchronizers.
:- supers [sync_object].
:- attributes [mset=(V)\\mset(initially,V)].

sync(Message)-Stream:=
    ( true → get_mset(MethodSet),
      sync(Message,MethodSet)–Stream).

sync(Message,MethodSet)–Stream:=
    ( akl.record_name(Message, Functor),
      akl.record_size(Message, Arity),
      sets.set_member(Functor/Arity,MethodSet) →
      Message,
      synchronizer(MethodSet,NewEnables),
      interpret_mset(NewEnables,NewMethodSet),
      get_message_queue(Q),
      wake(MethodSet,NewMethodSet)–Q–Stream,
      set_message_queue(Q),
      set_mset(NewMethodSet)
    ; true →
      get_message_queue(Q),
      NewQ = [Message|Q],
      set_message_queue(NewQ)).

interpret_mset(NewEnables,NewMethodSet):=
    ( NewEnables = [_|_] →
```

```
    list.sort(NewEnables,NewMethodSet)
  ; true → mset(NewEnables,NewMethodSet)).
```

where:

- *sync/[3-4]* is the method defined for executing or delaying an invocation request by the receiving object according to the current accept method set. The current accept method method set is indicated by the attribute *mset*. If the method selector of the invocation request is an element of this set, the method is executed; otherwise, the method is delayed (enqueued). After invoking the method, the *synchronizer/2* method is invoked to compute the new accept method set. Then, the *message-queue* is scanned to reschedule the delayed messages that its methods become enabled in the new accept method set.
- *interpret_mset/2* is the method defined to determine the accept method set. If the accept method set is indicated by an identifier the method *mset* is invoked and the enabled methods is returned; otherwise, the enabled methods is the one at hand and is returned.

Example 4.7 Consider the definition of the bounded buffer, *buffer_sync*, class with *synchronizers*. It is a first-in first-out buffer that can contain at most *MaxSize* items. It has two public methods *put/1* and *get/1*. The method *put/1* stores one item in the *buffer*, an array, and *get/1* removes the oldest one. Two attributes *in* and *out* that act as indices into the buffer. Upon creation, the buffer is in the empty state and the only message acceptable is *put/1*; arriving *get/1* messages are not accepted but kept in the message queue unprocessed. When a *put/1* message is processed, the buffer is no longer empty and can accept both *put/1* and *get/1* messages, reaching a "partial" (non-empty and non-full) state. When the buffer is full, it can only accept *get/1*, and after processing the *get/1* message, it becomes partial again. The method *mset/2* defines the possible accept method set of the bounded buffer with their identifiers. The method *synchronizer/2* specifies the enabling of methods for each state of the bounded buffer.

```
:- class buffer_sync(MaxSize).
:- supers [synchronizers,state_hash].
:- attributes[in=(V)\\(V=0),out=(V)\\(V=0),size=(V)\\(V=0),
              buffer=(Array)\\(akl.new_array(MaxSize,0,Array))].

mset(SetId,Mset):=
    ( SetId = initially → mset(empty,Mset) # buffer_sync(MaxSize)
    ; SetId = empty → Mset= [put/1]
    ; SetId = full → Mset = [get/1]
    ; SetId = partial →
```

```
    mset(empty,Mset1) # buffer_sync(MaxSize),
    mset(full,Mset2) # buffer_sync(MaxSize),
    sets.set_union(Mset1, Mset2, Mset)).

synchronizer(MethodSet,Enables):=
    ( true → get_size(Size),
      enable(Size,MaxSize,MethodSet,Enables)).

enable(Size,Max,MethodSet,Enables):=
    ( Size > 0, Size < Max → Enables = partial
    ; Size = 0 → Enables = empty
    ; Size = Max → Enables = full).

put(Item) := / * store an item * /

get(Item) := / * remove an item * /
```

□

Note that the code for accessing the local array storage for insertion and removal is omitted through out this chapter for brevity. However, this is the piece of code that represents the part which is to be inherited rather than re-implemented, i.e. overridden.

4.4.3 Transitions

Transitions can be used as an alternative to the synchronization scheme *synchronizers*. A *transition* specifies the transitional behavior of an object's accept method set, that reflects the synchronization constraint dictated by the internal state of the object.

The transitions are specified on a method-by-method basis, via method *transition/3*. Each class can have a default transition, indicated by the keyword *default*. The following definition exhibits the behavior of the standard class *transitions*:

```
:- class transitions.
:- supers [sync_object].
:- attributes [mset=(V)\\mset(initially,V)].

sync(Message)–Stream:=
    ( true → get_mset(MethodSet),
      sync(Message,MethodSet)–Stream).
```

```
sync(Message,MethodSet)–Stream:=
    ( akl.record_name(Message, Functor),
      akl.record_size(Message, Arity),
      sets.set_member(Functor/Arity,MethodSet) →
      Message,
      MsgSelector= Functor/Arity,
      transition(MsgSelector,MethodSet,NewEnables),
      interpret_mset(NewEnables,NewMethodSet),
      get_message_queue(Q),
      wake(MethodSet,NewMethodSet)–Q–Stream,
      set_message_queue(Q),
      set_mset(NewMethodSet),
    ; true → get_message_queue(Q),
      NewQ = [Message|Q],
      set_message_queue(NewQ)).
interpret_mset(NewEnables,NewMethodSet):=
    ( NewEnables = [_|_] →
      list.sort(NewEnables,NewMethodSet)
    ; true → mset(NewEnables,NewMethodSet)).
```

where:

- *sync/[3-4]* is the method defined for executing or delaying an invocation request by the receiving object according to the current accept method set. The current accept method method set is indicated by the attribute *mset*. If the method selector of the invocation request is an element of this set, the method is executed; otherwise, the method is delayed (enqueued). After invoking the method, the *transition/3* method is invoked to compute the new accept method set. Then, the *message-queue* is scanned to reschedule the delayed messages that its methods become enabled in the new accept method set.

- *interpret_mset* is the method defined to determine the accept method set. If the accept method set is indicated by an identifier the method *mset* is invoked and the enabled methods is returned; otherwise, the enabled methods is the one at hand and is returned.

Example 4.8 Consider the definition of the bounded buffer, *buffer_trans*, class with *transitions*. It has the same definition of *attributes*, *put/1, get/1, enable/4*, and *mset/2* of Example 4.7. The method *transition/3* specifies the enabling of methods for each state on a method-by-method basis. The example shows the possibility of having a default transition.

```
:- class buffer_trans(MaxSize).
    ...
transition(M,MethodSet,Enables):=
    ( M = default → get_size(Size),
      enable(Size,MaxSize,MethodSet,Enables) # buffer_trans(MaxSize)
    ; M = getb/1 → transition(default,MethodSet,Enables)
    ; M = putb/1 → transition(default,MethodSet,Enables)).
    ...
```

□

4.5 Inheritance

Object orientation attempts to model real-world applications as closely as possible. Object orientation also attempts to achieve software reusability. The powerful object-oriented concept that provides this capability is inheritance. This avoids redesigning and recoding everything from scratch. According to Cox [37], "Without inheritance every class would be a free-standing unit, each developed from the ground-up." A class may inherit operations from superclasses and may have its operations inherited by subclasses.

4.5.1 Class Inheritance

Inheritance of classes is established through explicit declaration of the inheritance relationships. For example, the classes given in the **supers** declaration of the class *counter* declares *ur_object* as a superclass of class *counter*, see Example 4.1. A class can inherit from more than one superclass. For example, the class *ur_object* has two superclasses: *object* and *state_hash*. Thus when an object O is created from class *counter*. Then O is a member of class *counter* and every class reachable (through superclass relationship) from *counter*. The interface of *counter* (the subclass) is a *superset* of the interface of its superclasses.

Inheriting attributes. In all object-oriented languages, objects of a subclass must retain the same type of information as objects of their superclass. Like CLOS [21], AKL+ allows inheriting attributes of the superclass. Inheriting an attribute automatically results in inheriting its attribute methods, see Section 4.1.2, so that methods in the subclass can access and manipulate the attributes of its superclasses through their accessor and update methods. For example, the class *counter* declares the attribute *val* in addition to attribute *self* declared by its superclass *ur_object*.

Inheriting methods. As indicated earlier, a class defines both the structure and behavior of a collection of objects. The behavior is specified by method definitions.

In an inheritance hierarchy, a method defined for a class is inherited by its subclasses. Thus, the inherited methods are part of the interface manipulating the objects of the subclass.

Method Overriding. A subclass can override an inherited method. By overriding, we mean that a method with the same selector is defined in the class. In other words, a method called *M* in class *C* can be overridden by a method called *M* in a subclass *C'* of *C*. Thus when a message with selector *M* is sent to an object *O* of class *C'*, then *M* will be bound to the method with the selector in the most specialized class of *O*.

In addition, a subclass can override an attribute access or update method that implicitly defined by the system. The ability to override the attribute access and update methods adds considerable flexibility to handle exceptions and special cases.

Example 4.9 The following class *jug* ensures that modifying the value of the *contents* attribute will not exceed the maximum limit of the jug object, see Fig. 4.13.

```
:- class jug.
:- supers [ur_object].
:- attributes [contents=(V)\\(V=0),capacity=(V)\\(V=100)].

set_contents(V):=
    ( true → get(capacity,Max),
      set_contents(V,Max).

set_contents(V,Max):=
    ( V =< Max → set(contents,V)^State
    ; true → io.format('Error: jug will exceeds its limit ~w~n', [V]).
```

□

The checking of the validity of the update operation is defined as an active constraint. Where the update is not performed if the constraint is violated. The definition of *jug* does not impose a static declaration of a jug capacity. Moreover, it is also possible to create objects with various capacity and the constraint violation checking is valid. This phenomenon belongs to a data-driven programming technique called *daemons*. Within AKL+ there are two types of events which may cause a daemon to be summoned. These are:

1. accessing or updating attributes values.
2. creating new objects or closing an object.

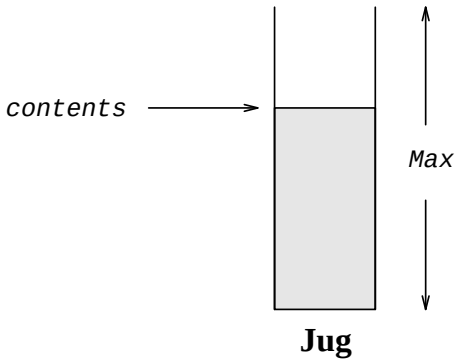

Figure 4.13: An example of method overriding

The methods which react to these events are a specialization of the most general operations such that whenever one of the above events is encountered the daemon is activated. Other than its special status with respect to certain events happening, the daemon is specified and behaves just like any other methods in AKL+. For example, the definition of *set_contents/[1-2]* in class *jug* specializes its default definition in order to define an attribute update daemon. Daemons can also be defined as a *before* or *after* actions depending on the way the daemon is reacted. For example, the definition of *set_contents/[1-2]* in *jug* class is a before daemon whereas the definition of *new/1* in *ur_object* is an after daemon. Note that the method *new/1* of *ur_object* delegates to the overridden method *new/1* of its superclass *cell*.

4.5.2 Differential Inheritance

Inheritance typically extends the interface of a subclass through defining new methods. Inherited methods can also be overridden explicitly through excluding some of the inherited methods in the subclass definition, so-called *differential inheritance* [102]. For example, assume the following methods have been defined for the class *bag*:

:- **class** bag.
insert(...) := ...
delete(...) := ...
union(...) := ...
difference(...) := ...
cartiesianProduct(...) := ...
numberOfOccurances(...) := ...

A bag is a collection of objects that can have duplicates (that is, the *same* object can occur in a bag more than once). To evaluate the number of occurrences of an

object in a bag we use the method *numberOfOccurances* which is a message sent to a bag object with an element object as argument. For example, the message:

```
numberOfOccurances(O)^B
```

will return the number of occurrences of the object O in the bag B.

We want to create the subclass *set* that will inherit from *bag*. A set is also a collection, except *an object can occur at most once* in the collection. Therefore, the subclass *set* specializes *bag* through overriding the method *insert*: before performing the actual insert we will check if the object is already in the set. If yes, we will return a message saying "object already in the set." Thus *insert* has an entirely different implementation in *set*.

With *numberOfOccurances* there is a problem. Conceptually, this method must not be part of the interface of *set*. The value of *numberOfOccurances* for set objects is either 0 (the element is not in the set) or 1 (the element is in the set). Instead, we can have a predicate *isElement* that returns true if an object is in a set. What is most important is that it is highly desirable to exclude *numberOfOccurances* from the interface of the class *set*.

There are two basic ways for excluding inherited methods:

1. *Override the method* and send a diagnostic message when it is invoked on an object of a subclass. This strategy is very general and can be used with almost any object-oriented language, since most languages support method overriding in the inheritance hierarchy. For example, the declaration of *numberOfOccurances* will be:

   ```
   :- class set.
   :- supers [bag].
   insert(...) := ...
   numberOfOccurances(...) := /* a diagnostic message */
   ```

   The disadvantage of this strategy is that it is at best a "hack": the method is still (a useless) part of the interface of the subclass and can lead to unnecessary name collisions when considering multiple inheritance [133].

   It should be noted that *Hiding* the method and making it subclass invisible will not solve the problem. For example, consider the private declaration of *numberOfOccurances* in class *bag*:

   ```
   :- class bag.
   ```

:- **private** [numberOfOccurances/1].
insert(...) := ...
delete(...) := ...
union(...) := ...
difference(...) := ...
cartiesianProduct(...) := ...
numberOfOccurances(...) := ...

The problem that may arise is that the method *numberOfOccurances* could not be inherited by all subclasses of class *bag*.

2. *specify explicitly* that the inherited method should not be inherited. This is supported through the *Differential Inheritance* feature of our language. For example, the declaration of the class *set* will be:

:- **class** set.
:- **supers** [bag-[numberOfOccurances/1]].
insert(...) := ...
...

This possibility is much cleaner. It enables better modeling since the interface of a class is *cleanly* captured.

AKL+ supports the exclusion of an inherited operation with the ability of the subclass to directly invoke the excluded operation to the named superclass.

4.5.3 Synchronization Constraints and Inheritance

Objects are instantiated from classes. Classes may be hierarchically organized according to an inheritance relation. Synchronization constraints are specified on per class basis and all objects instantiated from the same class have synchronization constraints with similar functionality.

Having separate specification of synchronization constraints is generally desirable to avoid the "inheritance anomaly". The term "inheritance anomaly" has been coined by Satoshi Matsuoka and Akinori Yonezawa [97, 99]. The anomaly manifests itself as subclasses in which specification of correct synchronization constraints require redefinition of method behavior that would otherwise be reusable. The anomaly may, for example, occur if synchronization constraints are specified as part of method behaviors. In that case, superclass synchronization constraints cannot be changed

in subclasses without also changing the methods of which the synchronization constraints are part. Different proposals have led to different synchronization schemes, the basic language features for programming the synchronization on message acceptance. Matsuoka noticed that the occurrence of the anomaly depends on the synchronization scheme of the language. That is, the re-definition would be required for the language that adopted certain synchronization scheme, while it could be safely inherited in another language that adopts an entirely different synchronization scheme. There are two directions in providing synchronization schemes [77]. One approach uses the *method guards* where a guard (activation condition) is attached to each method. Each method is responsible for ensuring certain condition before executing its body. The other approach uses what is called *interface control*, where a method execution is initiated only when the method is allowed to access the internal state of the object, i.e. enabled. The "anomaly" has been illustrated by a number of counter examples in the literature. For a more elaborated discussion of the inheritance anomaly and its causes, refer to [99]. We adopt those examples to illustrate the characteristic of our support of the synchronization constraints.

The approach taken in AKL+ is based on method guards, a very suitable synchronization scheme to our computation model, where a method choice is committed if its guard is reduced to true. The approach is developed from Matsuoka's proposal to support synchronization schemes for resolving the anomaly. However, our approach differs in the following:

- The programmer has the flexibility to program the synchronization schemes as well as constraints on the acceptable set of messages that suits his problem.
- The object synchronization is naturally considered as a behavior of the concurrent object. This includes not only the synchronization scheme but also the acceptance constraints on messages.
- There is no separate inheritance rules for accept method sets, synchronizers, and transitions.
- The acceptance of messages depends on dynamic operation on the accept method set, e.g. union, difference, etc. This will result in minimizing code re-definitions when multiple methods are affected, e.g. locking arbitrary operations.

AKL+ provides two synchronization mechanisms namely: *synchronizers* and *transitions*, discussed in Sections 4.4.2 and 4.4.3. There are two situations in which the anomaly can appear with method guards: history-only sensitiveness of acceptable state and modification of acceptable states [99]. To clarify history-only sensitiveness anomaly, consider creating a class *gb_buffer*, a subclass of bounded buffer. *gb_buffer* has one additional method *gget/1*. *gget/1* is almost identical to *get/1* with the sole

exception that it cannot be accepted immediately after the invocation of *put/1*. Such a condition for invocation cannot be distinguished with method guards and the set of attributes available in bounded buffer alone; we need to define extra attribute *after_put*. As a consequence, both *get/1* and *put/1* must be redefined. The following example shows that our approach can solve this problem.

Example 4.10 The following is the definition of class *gb_buffer* with class *buffer_trans* as a superclass, see Example 4.8, using our approach.

```
:- class gb_buffer(MaxSize).
:- supers [buffer_trans(MaxSize)].

mset(SetId,Mset) :=
    ( SetId = partial → mset(partial,Mset1) # buffer_trans(MaxSize),
      sets.set_union(Mset1, [gget/1], Mset)
    ; SetId = after_put → mset(partial,Mset) # buffer_trans(MaxSize)
    ; true → mset(SetId,Mset) # buffer_trans(MaxSize)).

transition(M,MethodSet,Enables):=
    ( M = put/1 → get_size(Size),
      enable(Size,MaxSize,MethodSet,Enables)
    ; true → transition(default,MethodSet,Enables) # buffer_trans(MaxSize)).

enable(Size,Max,_MethodSet,Enables):=
    ( Size > 0, Size < Max → Enables = after_put
    ; Size = Max → Enables = full).

gget(Item) := / * remove an item * /
```

Note that only the *mset/2* that are needed to distinguish between the acceptable constraints for *gget/2* is redefined. □

To clarify the second type of anomaly, consider the *lock class*, which is an abstract *Mixin* class. Direct instances of *lock* are not created; rather, the purpose of *lock* is to be "mixed-into" other classes in order to add the capability of locking an object. In *lock*, a pair of methods *lock/0* and *unlock/0* have the following functionality: an object upon accepting the *lock/0* message, will be "locked", i.e. will suspend the reception of further messages until it receives and accepts the *unlock/0* message. Its synchronization constraint is localized, i.e. it is not affected by methods of the class it is being mixed to. when *lock* class is mixed into the definition of the bounded buffer to create the class *lb_buf*, we are likely to assume that it would not affect the definition of other methods. However, this is not the case, we must add an attribute *locked* which indicates whether the current object is locked or unlocked. Then, the

inherited methods such as *put/1* and *get/1* must be overridden in order to account for locked. Note that the anomaly occurred because *lock/0* and *unlock/0* methods modify the synchronization constraints of the methods that are already defined, in this case both *put/1* and *get/1*. The following example shows that our approach can solve this problem. Another example is described in Section 7.4.

Example 4.11 Consider the following definition of class *write_lock* and with subclass *write_lock_buffer*. *write_lock* defines a two-level lock where the method *lock/0* locks the object exclusively so that no other methods can access it until it receives a corresponding *unlock/0* message, *write_lock_buffer* class is a bounded buffer that allows locking.

```
:- class write_lock.
:- attributes [lock_var=(V)\\(V=unlocked_lock),method_set=(M)\\(M=[])].

mset(SetId,Mset):=
    ( SetId = locked → Mset = [unlock/0]
    ; SetId = unlocked → get_method_set(Mset)).

transition(M,MethodSet,Enables):=
    ( M = default → get_lock_var(LOCK_VAR),
      enable(LOCK_VAR,MethodSet,Enables) # write_lock
    ; true → transition(default,MethodSet,Enables) # write_lock).

enable(LOCK_VAR,MethodSet,Enables):=
    (LOCK_VAR = locked_lock → Enables = locked,
     set_method_set(MethodSet)
    ;LOCK_VAR = unlocked_lock → Enables = unlocked).

lock:=
    ( true → set_lock_var(locked_lock),
      io.format('—— lock —— ~n')).

unlock:=
    ( true → set_lock_var(unlocked_lock),
      io.format('—— unlock —— ~n')).

:- class write_lock_buffer(MaxSize).
:- supers [buffer_trans(MaxSize),write_lock].

mset(SetId,Mset):=
    ( SetId = locked → mset(locked,Mset) # write_lock
    ; SetId = unlocked → mset(unlocked,Mset) # write_lock
```

```
    ; SetId = always → Mset = [lock/0]
    ; true → mset(SetId,Mset1) # buffer_trans(MaxSize),
      mset(always, Mset2),
      sets.set_union(Mset1, Mset2, Mset)).

transition(M,MethodSet,Enables):=
    ( M = lock/0 → transition(M,MethodSet,Enables) # write_lock
    ; M = uanlock/0 → transition(M,MethodSet,Enables) # write_lock
    ; true → transition(M,MethodSet,Enables) # buffer_trans(MaxSize)).
```

□

Note how the dynamic operation on the accept method set allows easy re-use of existing lock code.

4.5.4 Multiple Inheritance

Inheritance is characterized as single or multiple depending on the number of classes a class can inherit. *Single inheritance* is simple but restrictive. Single Inheritance forces designers to choose to inherit from one of several equally attractive superclasses and then duplicate the code from other superclasses that could not be inherited. Single inheritance result in a strict tree hierarchy. *Multiple inheritance* on the other hand is complex but allows a more natural definition of relationships between classes. Multiple Inheritance results in a Directed Acyclic Graph(DAG) of classes (*inheritance graph*). In this graph each class is a node and there is a directed arc from each class to each of its parents. Figure 4.14 shows an inheritance graph with multiple inheritance. There are four important use of multiple inheritance [130]:

- *Multiple independent protocols*: This covers situations where a class is created by combining completely independent superclasses.
- *MixIn-based inheritance*: Here several classes are specifically created for subsequent combination. These special classes are also known as *mixins* [54].
- *Submodularity*: This covers situations where while creating a system, modularity of subparts is noticed and factored out for good system design.

However, with the power of multiple inheritance also comes complexity. When a class inherits from more than one parent there is the possibility for conflicts, which are methods with the same name or selector but different or unrelated semantics inherited from different superclasses [86]. Meyer makes the following two important observations about this problem [107]. First, it is a syntactical problem rather than a

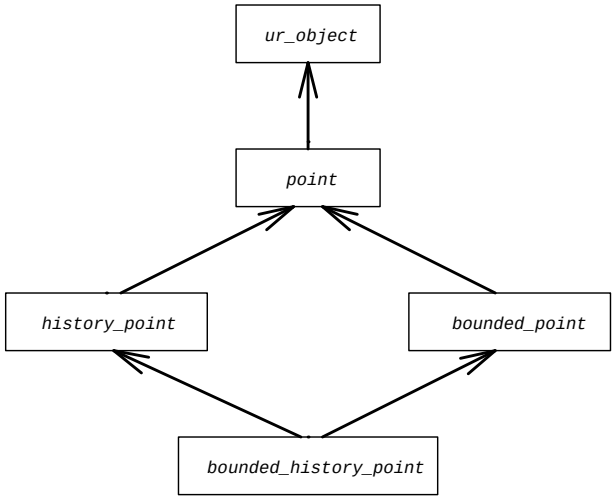

Figure 4.14: An example of multiple inheritance

fundamental property of inheritance and classes. The problem is due to a confliction in the choice of names. Second, the problem is not with parent classes. Each parent class is consistent as it stands. The problem is caused by the common descendant class that combines the incompatible parent classes. Therefore, the descendant is the most logical place to resolve the problem.

A language that supports multiple inheritance must provide strategies to resolve the conflict of unrelated methods. The following are the basic strategies for resolving this conflict:

- *linearization*: This strategy specifies a linear, overall order of classes, and then specifies that application of a class method or attribute starts from the most specific class. One possible way of linearization to the *point* class hierarchy depicted in Fig. 4.14 is shown in Fig. 4.15. This is the approach taken in Flavors [108], CLOS [21, 82] and CommonLoops [20]. As pointed out by Snyder [133], the main problem with this approach is the ordering of superclasses in a class declaration has significant semantic implications.

- *Forbidding conflicts–Renaming Strategies*: This is the simplest and most obvious strategy. An error is issued for totally unrelated attributes or methods. This requires the programmer to rename the conflicting methods. The approach is similar to the one taken in Eiffel, Trellis/Owl.

- *Qualifying attributes and methods*: This strategy provides a straight forward solution to the conflicts by the qualification of attribute or method names with the name of the class. With this strategy, whenever there is an ambiguity in the *access* or *usage* of an instance variable or method, it must be resolved through qualifying the variable or method with the appropriate class name. C++ uses this strategy in supporting multiple inheritance.

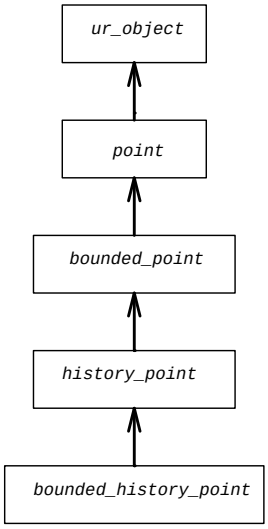

Figure 4.15: An example of linearized multiple inheritance

- *Excluding operations*: This strategy provides the ability for a subclass to exclude an inherited operation. This facility is used in CommonObjects [132]. In CommonObjects the inheritance graph is converted into a tree by duplicating nodes.

In our support to handle multiple inheritance, we model the inheritance graph directly. In other words, we don't flatten the inheritance graph into a linear chain, and then deals with this chain using the rules for single inheritance. Operations are inherited along the inheritance graph, excluding differentially inherited and private operations encountered, until redefined in a class. If a class inherits operations with the same selector from more than one superclass, a default (implicit) differential inheritance is applied; excluding all the methods with the same selector occurring further on the right. Inheriting an operation is not a strict depth-first traversal of the inheritance graph since a definition for a differential inheritance operation will affect the inheritance path.

The attribute specification of classes are inherited according to the same rules as methods.

In AKL+ the designer is able to resolve the conflict in different ways. One way is to redefine the operation in the subclass. An alternative way is to differentially inherit the conflicting operation.

4.6 Summary

AKL+ and its computation model are informally presented. Several simple examples are used to illustrate the main features and programming techniques. Classes can

be defined with attributes, methods, access control of methods, and superclasses. Classes can be parameterized. A parameter of a parameterized class lies in four categories: class abstraction, constant, object or method abstraction. The passing of a class parameter is not resolved at compile time but rather all references are bound at run-time. This is due to the dynamic binding of parameters with the actual code. Mechanisms for resolving multiply inherited definitions, such as differential inheritance, are supported. A set of implicit behavior is defined for each defined class. These are supported for error handling, attributes manipulation, and class membership. Classes, methods and instances of classes can be expressed as first-class values in the language which may be passed as arguments, returned as results and stored in attributes of objects. Defining classes and methods as abstractions provide for all higher-order programming techniques. State, which is an object, is encapsulated and attributes are only accessible via message sending. Objects can be explicitly allocated or destroyed using meta level operations provided by the language. Objects can share a common object by passing it as a parameter to their classes at creation time. AKL+ achieves a uniform message sending which means that objects of different types can receive messages in the same manner.

Two implicit arguments are passed to every method: the base class so that method calls to this class can be applied and an accumulator pair to account for the incoming state and the outgoing state that suffices to achieve state change. The body of a method threads the input state of the method through the method calls and returns it as the output state. The target object is available under the special attribute *self* so that methods can send messages back to their target object.

One of the most important design issues, in AKL+, is to supply programmers with the minimum set of efficient and effective built-in standard classes (library). This fulfills the users computational needs as it provides him with a simpler, and easier-to-use computing environment.

The language is very flexible to support data-driven programming like, specifying daemons, default methods, and class-specific methods. A daemon can be specified as an active constraints on attributes values or as a method triggered when an object is no longer in use. A default method is automatically invoked when there is no message selector matches with any methods of the interface. A class-specific method makes it possible to call methods on the basis of the identity of classes.

The language supports the basic synchronization schemes that achieve the concurrency control for a concurrent object. This is realized by providing the concurrency control mechanisms for sending messages in a batch, serializing messages, acknowledgment of messages between a sender and a receiver objects, and by specifying the synchronization constraints for an object to accept or delay its messages according to its current state.

The concept of inheritance anomaly has been introduced into object-oriented concurrent programming in [97] and further defined in [98, 99]. It was shown that existing synchronization schemes are weak in one or more of these anomalies. The appearance of these anomalies has a great significance because, from now on, any forthcoming proposals for language tools in object-oriented concurrent programming can and should be demonstrated to successfully solve these critical cases. In our language, we have provided two standard protocols, transitions and synchronizers, to support synchronization schemes for resolving the anomaly.

⋆ ⋆ ★ ⋆ ⋆

Definition of AKL+

AKL+ is a concurrent language supporting object-oriented programming. It is built on top of AKL (Agents Kernel Language) [74]. AKL is the base for a formal definition of AKL+. The semantics of AKL+ is defined in this chapter by their stepwise translation (expansion) to AKL.

This chapter is organized as follows. In Section 5.1 we present the abstract syntax of AKL+. In Section 5.2 a simple example is used to illustrate how a class is translated into AKL. In Section 5.3 we give the translation rules to expand a class definition into AKL.

5.1 Definitions and Programs

We use the following notations in describing the syntax of programs: letters in bold are keywords, letters between ⟨ and ⟩ are nonterminal symbols, letters between ′ and ′ are terminal symbols, and the symbol | separates alternatives.

Assume given sets of *variables, constraint names, procedure atom names, method atom names, module names, attribute names, and class names.* A constraint atom is an expression of the form $c(X_1, \ldots, X_n)$ where c is a constraint name and $X_1, \ldots, X_n$ are variables. Two primitive constraint atoms are provided directly: *fail* and *true* which indicates the failure and success, respectively. Similarly, *procedure atom* and *method atom* are expressions of the form $p(X_1, \ldots, X_n)$ where p is either a *procedure atom name* or a *method atom name* and $X_1, \ldots, X_n$ are different variables. An *atom* is a constraint atom, procedure atom, or a method atom. The name of an atom is called *functor*. The variables in an atom are called *parameters*. The number of parameters–determined by p or c– is called the *arity* of the atom. The *selector* is an expression of the form p/n which uniquely identifies the definition.

We assume an equality constraint name of the form = provided by the underlying constraints system, e.g. $X = 1$, is often called a *binding*, suggesting that the variable X is "bound" to *1* by the constraint. A variable, e.g. Y, bound to a definition is called an *abstraction*. An application (call) to this abstraction is an expression of the form $Y(X_1, \ldots, X_n)$, where $X_1, \ldots, X_n$, are the actual parameters of the definition.

A declaration of a procedure as visible, is an expression of the form *:-* **public** p/n. A call to p/n defined in module *mod* is an expression of the form $mod.p(X_1, \ldots, X_n)$.

A *class atom* is an expression of the form $cname(X_1, \ldots, X_n)$ where $X_1, \ldots, X_n$ are variables of global scope to the class definition. A declaration of *superclasses* is an expression of the form :- **supers** $[Super_1 - [d_1, \cdots, d_m], \ldots, Super_n - [d_1, \cdots, d_o]]$, where $Super_i$ is a class atom, and d_i is a method selector.

A declaration of a hidden (invisible) method is an expression of the form :- **private** $[d_1, \cdots, d_n]$, where d_i is a method selector.

A declaration of attributes is an expression of the form :- **attributes** $[a_1 = m_1, \ldots, a_n = m_n]$, where a_i is an attribute name and d_i is a method abstraction.

Sending a message to an object is an expression of the form $p(X_1, \ldots, X_n)\hat{}\, O$, where O is a variable and p/n is a method atom.

A class abstraction is an expression of the from $Y = \#q$, where Y is a variable and q is a class atom. An application to this abstraction is an expression of the form $p(X_1, \ldots, X_n)\#Y$, where p/n is a method atom.

A method invocation is an expression of the form $p(X_1, \ldots, X_n) <\# \ q$, where q is a class atom and p/n is a method atom. A method delegation is an expression of the form $p(X_1, \ldots, X_n)\# \ q$, where q is a class atom and p/n is a method atom.

The remaining (abstract) syntactic category pertaining to programs is shown in Fig. 5.1.

A *hiding* statement introduces variables with local scope.

The clauses of a choice statement have the same guard operator. To the guard operators correspond *conditional* choice ('→'), *committed* choice ('|'), and nondeterminate choice ('?') statements, respectively. The symbol **';'** separates clasuses within the choice statement. The parameters of a head atom are called *formal parameters*. A head of a definition introduces variables of local scope to its definition.

A *program* is a finite set of classes and module definitions, satisfying the condition that every method atom, procedure atom, module name and class atom occurring in the program has a definition.

5.2 A Simple Example

The class expansion is transparent to the user. Every defined class will translate to a definition of an AKL module with the same name as the functor of the class atom.

As a simple example that shows how AKL+ code expanded to AKL code, consider the definition of the class *ord_list* that follows. This class has four method definitions: *insert_element/1, insert/3, insert_aux/5,* and *less/3* and an attribute *list* for storing an ordered list. The method *insert_element* inserts an element in *list*. The

⟨*program*⟩	::=	⟨*set of definitions of classes and modules*⟩
⟨*module definition*⟩	::=	⟨*module heading*⟩⟨*set of procedure definitions*⟩
⟨*class definition*⟩	::=	⟨*class heading*⟩⟨*set of method definitions*⟩
⟨*module heading*⟩	::=	$'{:}-'$ **module** ⟨*module name*⟩
		⟨*set of public definitions*⟩
⟨*public definition*⟩	::=	$'{:}-'$ **public** ⟨*selector*⟩
⟨*selector*⟩	::=	⟨*functor*⟩$'/'$⟨*arity*⟩
⟨*class heading*⟩	::=	$'{:}-'$ **class** ⟨*class atom*⟩
		$'{:}-'$ **supers** ⟨*sequence of class atoms*⟩
		$'{:}-'$ **attributes** ⟨*set of attribute declarations*⟩
		$'{:}-'$ **private** ⟨*set of selectors*⟩
⟨*procedure definition*⟩	::=	⟨*definition*⟩
⟨*method definition*⟩	::=	⟨*definition*⟩
⟨*definition*⟩	::=	⟨*head*⟩ $':='$ ⟨*body*⟩
⟨*head*⟩	::=	⟨*procedure atom*⟩ | ⟨*method atom*⟩
⟨*body*⟩	::=	⟨*statement*⟩
⟨*statement*⟩	::=	⟨*atom*⟩ | ⟨*composition*⟩ | ⟨*hiding*⟩
	|	⟨*choice*⟩ | ⟨*aggregate*⟩ | ⟨*abstraction*⟩
	|	⟨*abstraction application*⟩ | ⟨*method invocation*⟩
	|	⟨*method delegation*⟩ | ⟨*message send*⟩
⟨*atom*⟩	::=	⟨*constraint atom*⟩ | ⟨*procedure atom*⟩ | ⟨*method atom*⟩
⟨*composition*⟩	::=	⟨*statement*⟩$','$⟨*statement*⟩
⟨*hiding*⟩	::=	⟨*set of variables*⟩$':'$⟨*statement*⟩
⟨*choice*⟩	::=	⟨*sequence of clauses with the same guard operator*⟩
⟨*clause*⟩	::=	⟨*set of vars*⟩$':'$⟨*statement*⟩⟨*guard operator*⟩⟨*statement*⟩
⟨*guard operator*⟩	::=	$'\rightarrow'$ | $'\mid'$ | $'?'$
⟨*aggregate*⟩	::=	*aggregate*(⟨*variable*⟩$','$⟨*statement*⟩$','$⟨*variable*⟩)
⟨*abstraction*⟩	::=	⟨*method abstraction*⟩ | ⟨*class abstraction*⟩
	|	⟨ *procedure abstraction*⟩
⟨*method abstraction*⟩	::=	⟨*variable*⟩ $'='$ ⟨*nameless definition*⟩
⟨*class abstraction*⟩	::=	⟨*variable*⟩ $'='$ $'\#'$ ⟨*class atom*⟩
⟨*procedure abstraction*⟩	::=	⟨*variable*⟩ $'='$ (⟨*set of variables*⟩) $'\backslash'$ ⟨*body*⟩
⟨*abstraction application*⟩	::=	⟨*class application*⟩ | ⟨*method application*⟩
	|	⟨*procedure application*⟩
⟨*class application*⟩	::=	⟨*method atom*⟩ $'\#'$ ⟨*variable*⟩
⟨*method application*⟩	::=	⟨*variable*⟩(⟨*sequence of variables*⟩)
⟨*procedure application*⟩	::=	$'akl.'$⟨*variable*⟩(⟨*sequence of variables*⟩)
⟨*method invocation*⟩	::=	⟨*method atom*⟩ $'<\#'$ ⟨*class atom*⟩
⟨*method delegation*⟩	::=	⟨*method atom*⟩ $'\#'$ ⟨*class atom*⟩
⟨*message send*⟩	::=	⟨*method atom*⟩ $'\hat{}'$ ⟨*variable*⟩
⟨*nameless definition*⟩	::=	(⟨*set of variables*⟩) $'\backslash\backslash'$ ⟨*body*⟩
⟨*attribute declaration*⟩	::=	⟨*attribute name*⟩ $'='$ ⟨*nameless definition*⟩

Figure 5.1: Abstract syntax of AKL+ program

method *insert/3* inserts an element in an ordered list yielding a new ordered list as a result. The method *insert_aux/5* is an auxiliary definition of *insert/3*; hence, it is a private one. The method *less/3* defines the criterion which compares two elements in order to insert an element at the correct position. The activation of *insert/3* by *insert_aux/5* and the activation of *insert_aux/5* by *insert/3* are local invocations. The *less/3* invocation is an application to the base class. For instance, when the *insert/3* is invoked and the base class, bound by the reserved variable *Self*, is *ord_list*, the *less/3* of *ord_list* is invoked, i.e. the *less/3* invocation by the *insert/3* definition is determined through the dispatcher of the base class. The expression *"–State"* is a syntactic variable that indicates a pair of arguments which couples variables that are needed to cater the incoming and the outgoing state.

```
:- class ord_list.
:- private [insert_aux/5].
:- attributes [list=(V)\\(V=[])].

insert_element(El):=
    ( true ? get(list, L0),
      set(list, L),
      insert(El,L0,L)).

insert(El, L0, L):=
    ( L0 = [] → L = [El]
    ; L0 = [E|L1] → less(El,E,YesNo) # Self,
       insert_aux(El, E, YesNo, L1, L) # ord_list).

insert_aux(El, E, YesNo, L1, L):=
    ( YesNo = yes → L = [El,E|L1]
    ; YesNo = no → L = [E|L2],
      insert(El, L1, L2) # ord_list ).

less(I1, I2, YesNo):=
    ( I1 < I2 → YesNo = yes
    ; true → YesNo = no).
```

The above definitions are expanded to AKL code, as follows:

```
:- module ord_list.
:- public dispatch/5.
:- public typeof/5.
:- public less/5.
```

```
:- public insert_element/5.
:- public insert/5.
:- public init_list/5.
:- public domain/5.
:- public attribute_property/5.

typeof(typeof(Class),Myself,Self)–State:=
    (true ? Class = ord_list, Myself = ord_list ).

init_list(init_list(V),Myself,Self)–State:=
    (true ? V = []).

attribute_property(attribute_property(Att,MethInit,MethGet,MethSet),
Myself,Self)–State:=
    ( Att = list → MethInit = (V,Self)–State\Self([init_list(V)])–State,
      MethGet = (V,Self)–State\Self([get_list(V)])–State,
      MethSet = (V,Self)–State\Self([set_list(V)])–State).

domain(domain(X),Myself,Self)–State:=
    (true ? X = [list]).
insert_element(insert_element(El),Myself,Self)–State:=
    ( true ? Self([get(list,L0)])–State,
      Self([set(list,L)])–State,
      Self([insert(El,L0,L)])–State).

insert(insert(El,L0,L)Myself,Self)–State:=
    ( L0 = [] → Myself = ord_list, L = [El]
    ; L0 = [E|L1] → Myself = ord_list,
      Self(less(El,E,YesNo))–State,
      insert_aux(insert_aux(El,E,YesNo,L1,L),ord_list,Self)–State).

insert_aux(insert_aux(El,E,YesNo,L1,L),Myself,Self)–State:=
    ( YesNo = yes → Myself = ord_list,  L = [El,E|L1]
    ; YesNo = no → Myself = ord_list,  L = [E|L2],
      insert(insert(El,L1,L2),ord_list,Self)–State).

less(less(I1,I2,YesNo),Myself,Self)–State:=
    ( I1 < I2 → Myself = ord_list,  YesNo = yes
    ; true → Myself = ord_list,  YesNo = no).

dispatch(Msg,Myself,Self)–State:=
    ( Msg = insert_element(Arg1) → insert_element(Msg,Myself,Self)–State
    ; Msg = insert(Arg1,Arg2,Arg3) → insert(Msg,Myself,Self)–State
```

```
; Msg = less(Arg1,Arg2,Arg3) → less(Msg,Myslef,Self)–State
; Msg = typeof(Arg1) → typeof(Msg,Myslef,Self)–State
; Msg = init_list(Arg1) → init_list(Msg,Myself,Self)–State
; Msg = attribute_property(Arg1,Arg2,Arg3,Arg4) →
  attribute_property(Msg,Myself,Self)–State
; Msg = domain(Arg1) → domain(Msg,Myslef,Self)–State).
```

The *ord_list* class is expanded with additional definitions: *typeof/5, dispatch/5 init_list/5, domain/5, attribute_property*, and *public* definitions. The expansion of classes and their method codes will be discussed in the following section.

5.3 Expansion to AKL Code

The expansion of AKL+ definitions to AKL definitions is based on a source-to-source transformations. The operational and declarative semantics of AKL+ programs is given in terms of their translations to AKL[1]. The method of defining a language semantics by translating the language into another (formal) language has been used to give semantics to the object-oriented language called Vulcan [79] in terms of translations to the language Concurrent Prolog [126] and the object-oriented language Oz [61] in terms of translations to the language kernel Oz [131], among others.

The expansion of a class definition into AKL code is based on transformation rules. Each kind of transformation is defined by a rewrite rule, $D \Rightarrow D'$, that substitutes the definition D', on its right hand side, for the definition D, on its left hand side. The definition to which a method expand takes five arguments:

$$\boxed{\mathcal{M}ethod : Message \times Myself \times Self \times State \rightarrow State}$$

The first argument is the received message (a method atom), the second argument is the class name of the class being defined, the third argument is an abstraction of the base class, the fourth argument is the state at the time of message reception, and the fifth (output) argument is the state that results from the method activation.

Notations

In the transformation rules that follows, the following notations are used:

[1]The definition of AKL is given in [58, 74, 50, 51].

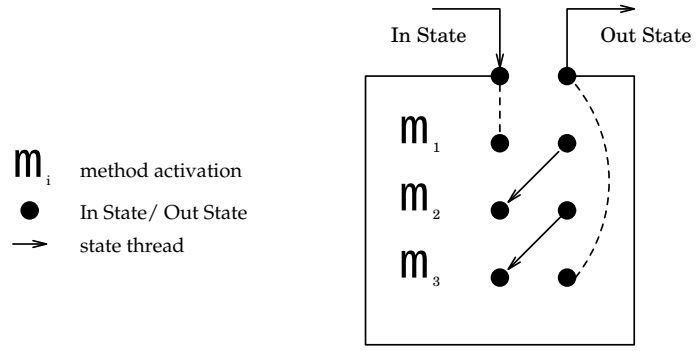

Figure 5.2: State threading through the body of a method

- *goal* ⇒ *goal(X, Y)*

  denotes the transformation produced by augmenting *goal* by the two arguments *X* and *Y*.

- *goal* ⇒ *goal–Z*

  denotes the transformation produced by augmenting *goal* by the accumulator pair *Z*. An accumulator pair is a syntactic variable that indicates a pair of arguments. The first argument is called the *input argument* and the second is the called the *output argument*. The concept of the accumulator pair is very useful in *state threading* where a pair of arguments represent the input and output version of some changing data that is passed along from one definition to another, see Fig. 5.2. For example, the definition

```
p(...)–State:=
    ( true →
      q(...)–State,
      r(...)–State).
```

  is equivalent to

```
p(...,State0,Staten):=
    ( true →
      q(...,State0,State1),
      r(...,State1,Staten)).
```

  Here *–State* represent a pair of variables which are inserted as arguments. This has the effect of reducing the proliferation of variable names and implicitly expressing the threading of the state arguments through the definition.

- the sequence of definitions $d_1, \cdots, d_n$ is denoted by: $\left\{ \begin{array}{l} d_1, \\ \vdots \\ d_n \end{array} \right\}$

Program transformation rule

program $\Rightarrow$ *program'*

Where *program* is an AKL+ program and *program'* is an AKL program.

Class transformation rule

Class definition $\Rightarrow$ *module definition*

Class heading transformation rule

Class declaration transformation rule

:- **class** *Class* $\Rightarrow \left\{ \begin{array}{l} :- \textbf{module}\ m \\ definition' \end{array} \right\}$

where *Class* is the class atom of the class being defined, and m is its functor. *definition'* is the expansion of the method definition:

typeof(X) := **true** *?* X = *Class*

Attributes declaration transformation rule

Let a_1,...,a_n represent the attributes defined in the class being defined. Let a_{n+1},...,a_m represent its inherited attributes.

:- **attributes** *[*a_1=$(X_1)\backslash\backslash b_1$,...,$a_n$=$(X_n)\backslash\backslash b_n$*]* $\Rightarrow \left\{ \begin{array}{l} definition_1' \\ definitiona_1' \\ \ldots \\ definitiona_n' \\ definition_2' \end{array} \right\}$

where a_i is an attribute name. $definition_1'$ is the expansion of the method definition:

$domain(X)$:= **true** ? $X = [a_1,\ldots,a_m]$

$definitiona_i'$ is the expansion of the method definition:

$inita_i\ (X_i)$:= **true** ? b_i

$definition_2'$ is the expansion of the method definition:

$property(A,Init,Get,Set)$:=

$(A = a_1 \rightarrow\ Init = (V)\backslash\backslash inita_1,$

$Get = (V)\backslash\backslash geta_1,$

$Set = (V)\backslash\backslash seta_1,$

; ...

; $A = a_m \rightarrow\ Init = (V)\backslash\backslash inita_n,$

$Get = (V)\backslash\backslash geta_m,$

$Set = (V)\backslash\backslash seta_m)$

Method definition transformation rule

$$head := body \Rightarrow \left\{ \begin{array}{l} head' := body' \\ :- \textbf{public}\ p/n \end{array} \right\}$$

where *head* translates to *head'* according to the *head transformation* rule and *body* translates to *body'* according to the *statement transformation* rule. p is the functor of *head'*, and n its arity that corresponds to the arguments: *Message, Myself, Self, State,* and (output) *State* of the expanded definition. The *public* definition is to be defined if the definition is visible so that definitions can call it from within other modules.

Head transformation rule

$head \Rightarrow head(Myself,Self)–State$

where *head* refers to a method head.

Statement transformation rules

Composition transformation rule

statement, statement $\Rightarrow$ *statement′,statement′*

Where *statement* translates to *statement′* according to the *statement transformation rules.*

Hiding transformation rule

$X_1, \ldots, X_n$: *statement* $\Rightarrow$ $X_1, \ldots, X_n$: *statement′*

where $X_1, \ldots, X_n$ are variables that have a scope limited to the statement being defined and *statement* translates to *statement′* according to the *statement transformation rules.*

Choice transformation rule

$$\left\{ \begin{array}{l} (\mathit{statement}\ \%\ \mathit{statement} \\ ; \ldots \\ ; \mathit{statement}\ \%\ \mathit{statement}) \end{array} \right\} \Rightarrow \left\{ \begin{array}{l} (\mathit{statement}'\ \%\ \mathit{statement}' \\ ; \ldots \\ ; \mathit{statement}'\ \%\ \mathit{statement}') \end{array} \right\}$$

Where % is a guard operator and *statement* translates to *statement′* according to the *statement transformation rules.*

Clause transformation rule

$$X_1, \ldots, X_n : \mathit{statement}\ \%\ \mathit{statement} \Rightarrow\ X_1, \ldots, X_n : \mathit{statement}'\ \%\ \mathit{Myself} = \mathit{Class},\ \mathit{statement}'$$

where % is a guard operator, $X_1, \ldots, X_n$ are variables that have a scope limited to the statement being defined, *Class* is the class atom of the class being defined, and *statement* translates to *statement′* according to the *statement transformation rules.*

Aggregate transformation rule

aggregate(X,statement,L) $\Rightarrow$ *aggregate(X,statement′,L)*

where *statement* translates to *statement′* according to the *statement transformation rules.*

Method abstraction transformation rule

$Z = (X_1, \ldots, X_n) \backslash\backslash statement \Rightarrow Z = (X_1, \ldots, X_n, Self)\text{–}State \backslash statement'$

where *statement* translates to *statement'* according to the *statement transformation rules*.

Procedure abstraction transformation rule

$Z = (X_1, \ldots, X_n) \backslash statement \Rightarrow Z = (X_1, \ldots, X_n) \backslash statement'$

where *statement* translates to *statement'* according to the *statement transformation rules*.

Method delegation transformation rules

- *goal* # *Class* $\Rightarrow$ *goal(Class,Self)–State*

  where *Class* refers to the class atom of the class being defined and *goal* is a method atom defined locally in *Class*.

- *goal* # *Class* $\Rightarrow$ *dispatch(goal,Class,Self)–State*

  where *Class* refers to the class atom of the class being defined and *goal* is a method atom not defined locally in *Class*.

- *goal* # *Class* $\Rightarrow$ *m.dispatch(goal,Class,Self)–State*

  where *Class* refers to a class atom other than the class being defined and *m* is its functor.

Method invocation transformation rules

- *goal* <# *Class* $\Rightarrow$ *Self'* = *(M)–State* \ *m.dispatch(M,Class,Self')–State*, *goal(Class,Self')–State*

  where *Class* refers to the class atom of the class being defined and *m* is its functor, and *goal* is a method atom defined locally in *Class*.

- *goal* <# *Class* $\Rightarrow$ *Self'* = *(M)–State* \ *m.dispatch(M,Class,Self')–State*, *dispatch(goal,Class,Self')–State*

  where *Class* refers to the class atom of the class being defined and *m* is its functor, and *goal* is a method atom not defined locally in *Class*.

- $goal <\# Class \Rightarrow Self' = (M)\text{–}State \backslash m.dispatch(M,Class,Self')\text{–}State,$
  $m.dispatch(goal,Class,Self')\text{–}State$

  where *Class* is a class atom other than the class being defined and m and is its functor.

Message send transformation rule

$goal\hat{}\, Object \Rightarrow meta.send(goal)\text{–}Object$

where *Object* is a variable, *goal* is a method atom, and *meta* is the run–time module, described in Section 6.5.

Method application transformation rule

$goal(X_1, \ldots, X_n) \Rightarrow akl.goal(X_1, \ldots, X_n, Self)\text{–}State$

where *goal* is a variable.

Class abstraction transformation rule

$Z = \#Class \Rightarrow Z = (M)\text{–}State \backslash m.dispatch(M,Class,Self)\text{–}State,$

where *Class* is a class atom and m is its functor.

Class application transformation rule

$goal \;\#\; Class \Rightarrow Class(goal)\text{–}State$

Where *Class* is a variable and *goal* is a method atom.

The Method Dispatcher

A definition of the method dispatcher is to be defined for each class definition. A clause of which is the entry point to the definition of a method atom. It is used for all method activations, except for invocations of local definitions. A method activation

is either a method invocation, a method delegation, a method application, or a class application. Like the method definition, a dispatcher definition takes five arguments:

$$\boxed{\mathcal{D}ispatch : Message \times Myself \times Self \times State \rightarrow State}$$

When the dispatcher is activated, the received message *Message* is matched with a method head pattern to select the target method definition. The ('→') control structure is used for selecting among messages which definition to execute. This definition is activated with the message itself, the class atom of the target method, the abstraction of the base class, and the incoming state, resulting in the outgoing state, respectively. The dispatcher will be described using the notational conventions that identifiers beginning with a lowercase letter are constants; those with an initial upper-case letters are variables. The dispatcher head has the following form:

dispatch(Msg,Myself,Self)–State

The dispatcher body consists of several statements of the following form, one for each method:

- if the method msg_i/n is defined locally, the statement is an expression of the form:

  $Msg = msg_i(Arg_1,\ldots,Arg_n) \rightarrow$
  $msg_i(Msg,Myself,Self)–State.$

- if the method msg_i/n is inherited from an ancestor class, say *Ancestor*, the statement is an expression of the form:

  $Msg = msg_i(Arg_1,\ldots,Arg_n) \rightarrow$
  $Myself = Class,$
  $Handler = Ancestor,$
  $ancestor.msg_i(Msg,Handler,Self)–State.$

  Where *Class* is the class atom of the class being defined, *Ancestor* is the class atom of the ancestor class that msg_i/n is inherited from, and *ancestor* is the functor of *Ancestor*. *Class* and *Ancestor* must be present to cater for sharing of class parameters.

Inheritance can be thought of as constructing a new definition of the method dispatcher from existing ones. Definitions are inherited along the inheritance graph, excluding differentially inherited and hidden definitions encountered, until redefined

in a class. If a class inherits definitions with the same selector from more than one superclass, a default (implicit) differential inheritance is applied; excluding all the method definitions with the same selector occurring further on the right. Inheriting a definition is not a strict depth-first traversal of the inheritance graph since an exclusion of a definition by the differential inheritance mechanism will affect the inheritance path.

5.4 Summary

We have presented the syntax of AKL+. The semantics of AKL+ is described in terms of translation to AKL. The translation is based on a source-to-source transformations. Each kind of transformation is defined by a rewrite rule. Inheritance is described by the incremental definition of the method dispatcher in such a way that the interface of a class is cleanly captured.

⋆ ⋆ ★ ⋆ ⋆

6

Implementation

AKL+ is an object-oriented language built on top of the AKL language. The AKL+ classes translate to AKL code at compile time in such a way that a method dispatches in a constant time. Fig. 6.1 illustrates the compiler and the run-time extensions of AKL.

A good programming language alone is not sufficient for economic software production. The programming environment has a significant influence on the productivity of software engineers [122]. We have designed and implemented an essential part of a software development environment, namely *incremental compilation facility* which keeps files up to date, à la *UNIX make* tool [49]. A primarily feature of the incremental compilation facility is that the work needed after a change is proportional to the "size" of the change rather than to the size of the program.

The AKL+ language and our incremental compilation facility have been implemented on Unix-based workstations and they are parts of the official release of the AKL system developed at SICS (Swedish Institute of Computer Science). The AKL system, AGENTS, is available from SICS for research and educational purposes (contact **agent-request@sics.se**).

In this chapter, we present the implementation aspects of the general incremental compilation facility and our language. In Section 6.1, we describe an algorithm for a general incremental compilation facility. The algorithm specifies dependencies as functions to (re)create the target files. In Section 6.2, we describe an algorithm for the dependency that computes multiple inheritance. In Section 6.3, we describe the dependency for expanding a class definition into AKL and show how it is based on the semantics of the preceding chapter. In Section 6.4, we briefly mention the dependency provided by AKL to expand the AKL code into the AKL abstract machine. In Section 6.5, we describe the AKL+ run-time module.

6.1 The Incremental Compilation Facility

The general incremental compilation facility is provided in order to bring a target file up to date with respect to those files on which it depends, which are called *dependencies*. All files reachable (directly or indirectly) are compiled and loaded. Since each dependency is a target, it may have dependencies of its own. The initial target file may be a source file or a project file. Source files are either akl files with

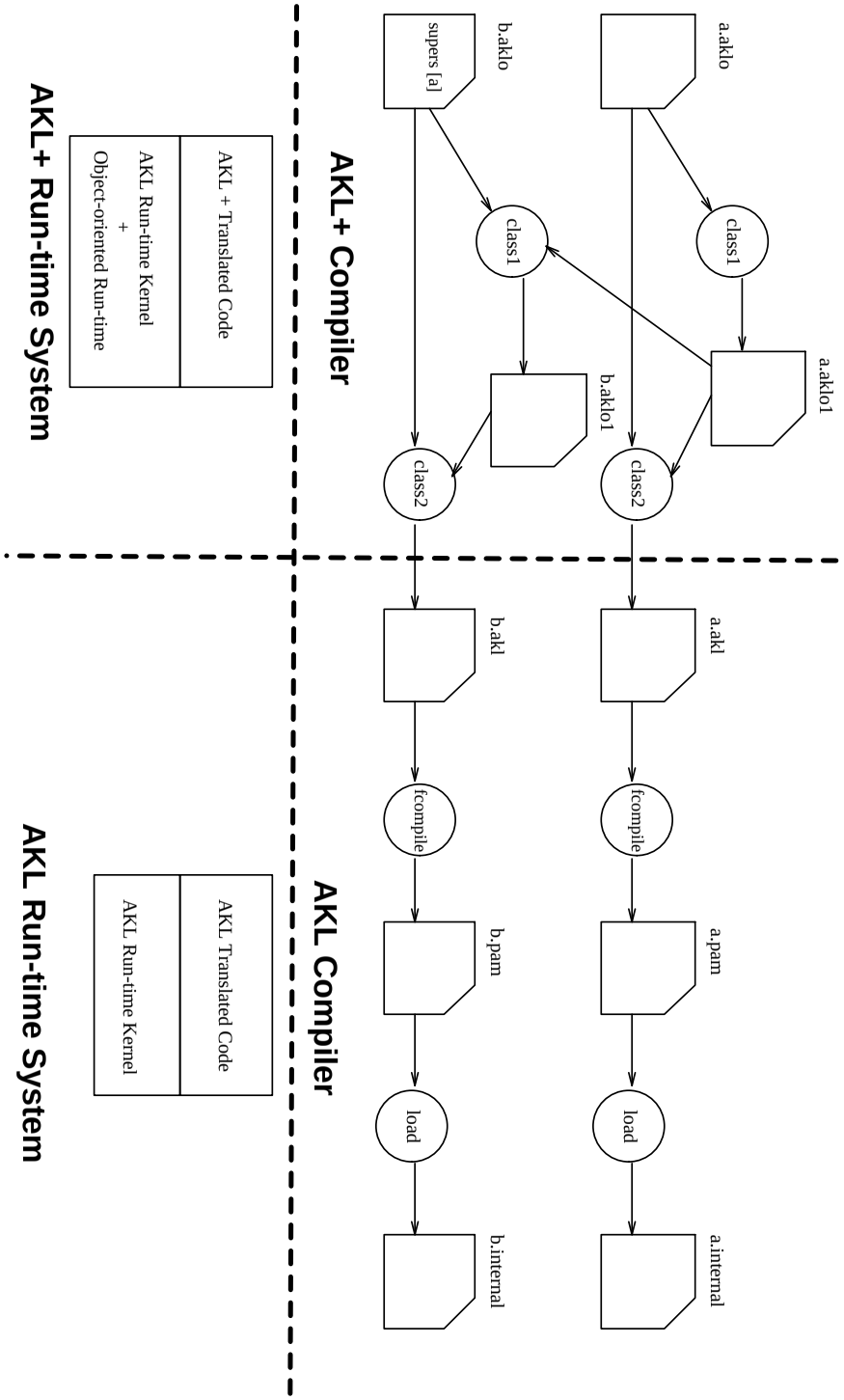

Figure 6.1: Compiler and run-time expansions of AKL

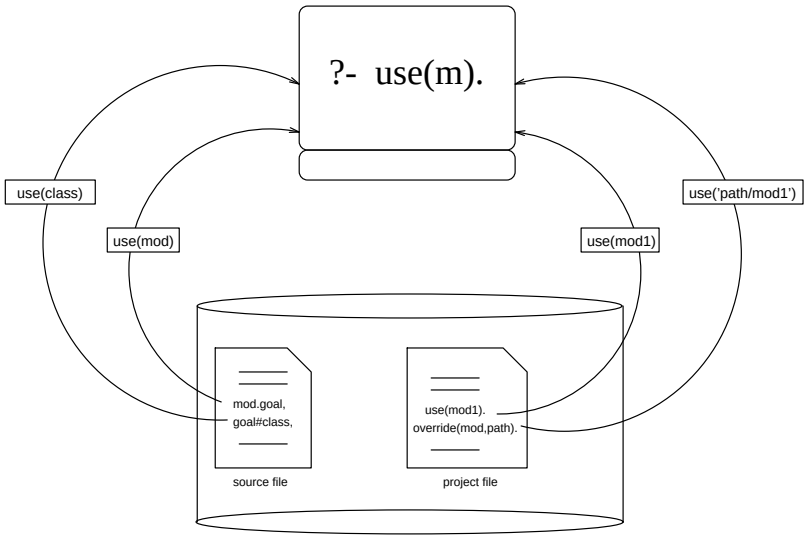

Figure 6.2: Finding the source files

the extension ".akl" or object-oriented files with the extension ".aklo"[1]. A project file specifies additional top level files and overrides the default placement of modules and classes. The project file has the extension ".proj". The entries in that file can have the following forms:

- *use(source)*: specifies that *source.akl (or source.aklo)* is used.
- *override(source,path)*: specifies that module (or class) *source* is to be found at *path*.

It should be noted that the order of the lines in the project file should not matter, e.g. the override is valid even if it mentioned after the use specification.

The Incremental Compilation Algorithm

When the user enters *agents* to start AGENTS, he may issue the command *use(RootFile)*, see Appendix A, and the system will automatically compile and load all files needed for *RootFile*. This is described by the following algorithm:

1. finding the source files:

   All reachable files from *RootFile* are to be found directly or indirectly as follows, see Fig. 6.2:

   - if a module is included it is to be searched for all other used modules and classes.

[1] We will identify files by their extensions.

- if a class is included it is also to be searched for all other used modules and classes, but it will also refer to its superclasses which are also to be included.

2. constructing the class dependency graph:

   Each node of this graph corresponds to a subclass or superclass. Each edge is a class dependency relationship (superclass–subclass). This graph is stored as a usual list of pairs (Node–Node). If any subgraph of this graph forms a cycle, the system will report an exception indicating those nodes, i.e. classes, that forms the cycle; then return to the top-level.

3. determining the files dependencies:

   There are a number of file dependencies. The possible dependencies are given below with the command that is to be used to (re)create the files

   - class1: (sub.aklo, super.aklo1) $\rightarrow$ sub.aklo1
   - class2: (x.aklo,x.aklo1) $\rightarrow$ x.akl
   - fcompile: x.akl $\rightarrow$ x.pam
   - load: x.pam $\rightarrow$ x.internal

   For $(A \rightarrow B)$ we say that B is *invalid* if A is newer than B. The age of the ".internal" file[2] is the time when it was loaded. A non-existing file is considered to be *very old*.

4. construct the file and the function graph, see Fig. 6.1:
   Each node of this graph is either:

   - a file: *file(FileName)*
   - a function: *function(FunctionName,InputFileList,OutputFileList)*

   The starting point for the graph is the list of source files.
   Each edge of this graph is a file dependency relationship:

   For an akl file named a.akl the following edges are added to the graph:

   file('a.akl')–function(fcompile,['a.akl'],['a.pam'])
   function(fcompile,['a.akl'],['a.pam'])–file('a.pam')
   file('a.pam')–function(load,['a.pam'],['a.internal'])
   function(load,['a.pam'],['a.internal'])–file('a.internal')

   For an aklo file named b.aklo the following edges are added to the graph:

[2]This file is not a physical file. We use it to indicate dependency of loading the executable AKL code into memory by the AKL run-time routines.

file('b.aklo')–function(class1,['b.aklo','super.aklo1'], ['b.aklo1'])
file('super.aklo1')–function(class1,['b.aklo','super.aklo1'], ['b.aklo1'])
(there might be several super.aklo1 files)
function(class1,['b.aklo','super.aklo1'], ['b.aklo1'])–file('b.aklo1')
file('b.aklo')–function(class2,['b.aklo','b.aklo1'],['b.akl'])
file('b.aklo1')–function(class2,['b.aklo','b.aklo1'],['b.akl'])
function(class2,['b.aklo','b.aklo1'],['b.akl'])–file('b.akl')

and then the nodes for the akl file 'b.akl' as above

This graph is stored as a usual list of pairs (Node–Node).

5. find the triggered functions:

   The function is triggered if at least one of its input files is newer than one of its output files.

6. extract the subgraph reachable from the triggered nodes.

7. do a topological sort of the remaining graph.

8. execute the functions in the order found by the topological sort.

In order to achieve a speed up and avoid redundancy in finding the source files, the source file is scanned and the following information is stored in a ".used" file:

- a list of all directly reachable modules and classes.
- a list of edges of the class dependency graph that represent the class dependency of the class being processed to its superclasses.

When finding source files, it suffices to read this file after checking its validity. This means that this file will be updated if its corresponding source file is update too.

The *class1* function is discussed in Section 6.2, The *class2* function is discussed in Section 6.3, and the *fcompile* function is briefly mentioned in Section 6.4.

6.2 Applying the Inheritance Mechanism

A method is executed if the object receives a message. Conceptually, when the method that services this message is not defined in the target object's class, this involves a bottom up (specialized to general) search for the appropriate method in the class hierarchy. AKL+ supports different strategies for resolving the conflicts that

may arise due to *multiple inheritance*. An effective strategy is to use the *differential inheritance* mechanism to selectively exclude any of the inherited methods such that the class dispatcher is cleanly captured. However, if such a conflict is encountered a default (implicit) differential inheritance is applied; excluding all the methods with the same name occurring further on the right. It is possible to control the access of the class methods and make them invisible to the instances of the class and its subclasses. The subclass-superclass relationship among classes forms a direct acyclic graph. Inheriting a definition is not a strict depth-first traversal of the inheritance graph since a definition of a differentially inherited operation will affect the inheritance path. Since efficiency is one of our prime motivations, the class dispatcher is computed at compile time and this will result in dispatching methods in a constant time rather than searching the class hierarchy each time a method is invoked. This run-time overhead is unpredictable and dependent on the depth of the inheritance path.

The result of applying the inheritance mechanism to a class is a file with the extension ".aklo1". This file will contain the class information, the inherited information, and the optimization information of the state access and update method invocations that are needed to construct the expansion of the class to AKL code and by its subclasses to construct their own ".aklo1" files. The structure of the ".aklo1" file is the term *table/9*. The following is the description of the elements of this term:

- *class-Id*: the class atom as given in the **class** declaration.
- *superclasses*: the superclasses as given in the **supers** declaration.
- *defined attributes*: the attributes as given in the **attributes** declaration.
- *private methods*: an ordered set of the private methods as given in the **private** declaration.
- *defined methods*: an ordered set of the defined and generated (implicit) methods of the class. An element of this set is the term *method_functor/method_arity*; where *method_functor* is the functor of the defined method and *method_arity* is its arity.
- *inherited methods*: a list of the methods inherited by the class. An element of this list is the term *ancestorclass-Id/ancestor methods*; where *ancestorclass-Id* is an ancestor class identifier and *ancestor methods* is an ordered set of the inherited methods from this ancestor. An element of this set is the term *method_functor/method_arity*; where *method_functor* is the functor of the inherited method and *method_arity* is its arity.
- *inherited attributes*: a list of the inherited attribute definitions.
- *state*: a tag to indicate which standard state representation class that is used.

- *flag*: a flag to indicate which state invocations is to be optimized.

Operations on these terms are based on ordered sets, a list with ordered elements, which provides efficiency in speeding up these operations rather than using the unordered representation [115].

The Multiple Inheritance Algorithm

Applying the inheritance mechanism involves the ".aklo1" files of the superclasses and the class source file (".aklo"). This means that each class will have a corresponding ".aklo1" file. This will have the effect of removing the redundancy in computing the inherited information each time a subclass is defined as well as facilitating the incremental compilation by working on separately compiled units. The following algorithm describes how to generate this file:

1. obtain *class-Id*, *superclasses*, *defined attributes*, *private methods*, and *defined methods* terms from the working class definition.

2. find the ".aklo1" files of the superclasses of the working class, i.e. obtain *table/9* for each superclass.

3. compute the working class information:

   (a) add entries of the generated methods to the *defined methods*:

   - add the *class membership method*.
   - add the *attribute methods*.

   (b) sort the *defined methods*.

   (c) sort the *private methods*.

   (d) compute the set difference between the *defined methods* and the *private methods* yielding the *defined methods* as the difference, i.e. after excluding the private methods.

4. compute the working class inherited information:

   A temporary ordered set is used, *combined methods*, for computing all the defined and inherited methods of the working class. Initialize this set to the *defined methods* of the working class. Initialize the *inherited methods* and the *inherited attributes* of the working class to the empty set, i.e. [].

   (a) accumulate the superclasses *inherited attributes* definitions to the working class *defined attributes* yielding the working class *inherited attributes*.

(b) compute the *inherited methods*:

For each superclass apply the following:

i. obtain the superclass *defined methods* and *inherited methods* from the superclass *table/9*.

ii. if the working superclass is associated with a differentially inherited methods, *excluded methods*, apply the following:

A. sort the set of the *excluded methods* of this superclass.

B. compute the set difference between the superclass *defined methods* and the *excluded methods* yielding the superclass *defined methods* as the difference.

iii. compute the set difference between the superclass *defined methods* and the *combined methods* yielding the superclass *defined methods* as the difference.

iv. compute the set union between the *combined methods* and superclass *defined methods* yielding the *combined methods* as the union.

v. obtain the *superclass-Id* from *superclasses*.

vi. add *superclass-Id* and the superclass *defined methods* to the working class *inherited methods*.

vii. obtain the superclass *class-Id* from the superclass *table/9*.

viii. unify the *superclass-Id* with the superclass *class-Id* yielding the most general unifier (*mgu*) of the two terms.

ix. For each element of the superclass *inherited methods*, apply the following:

A. apply the substitution of the *mgu* to the *ancestorclass-Id*.

B. if the working superclass is associated with differentially inherited methods, *excluded methods*, compute the set difference between the *ancestor methods* and the *excluded methods* yielding the *ancestor methods* as the difference.

C. compute the set difference between the *ancestor methods* and the *combined methods* yielding the *ancestor methods* as the difference.

D. add *ancestorclass-Id* and the *ancestor methods* to the working class *inherited methods*.

E. compute the set union between the *combined methods* and superclass *ancestor methods* yielding the *combined methods* as the union.

5. compute the optimization information of the state access and update method invocations:

If neither of the standard state representation is used, set *state* and *flag* to *none*. Otherwise, apply the following:

(a) set the *state* to the identifier of the standard state class.

(b) compute the intersection between the *combined methods* and the set *[get/2,set/2]* yielding *generic access and update* set.

(c) set the *flag* to the complement of the *generic access and update* set.

6.3 Expansion to AKL Code

The expansion of AKL+ definitions into AKL definitions is based on transformation rules. The definition of these rules is given in Section 5.3. Some of these rules are optimized for the purpose of implementation efficiency. Every defined class will translate to several AKL definitions belonging to a unique module with the same name as the functor of the class name. This means that there cannot be two classes having the same name even if they have different arities (number of arguments).

The incremental compiler uses the ".aklo1" file of the working class generated from applying the inheritance mechanism, and the source code of this class provided by the ".aklo" file to generate the module definition to which the class is expanded. This definition will be stored into the ".akl" file of the working class.

6.3.1 Examples of a Class Expansion

In the following we will use examples to show how classes are expanded into AKL.

Example 6.1 Consider the class *ord_list* that is defined in Section 5.2. Now, the following is its expansion.

```
:- module ord_list.
:- public dispatch/4.
:- public typeof/4.
:- public less/6.
:- public insert_element/4.
:- public insert/6.
:- public init_list/4.
:- public domain/4.
:- public attribute_property/7.

typeof(Class,Self)-State:=
    (true ? Class = ord_list).

init_list(V,Self)-State:=
```

```
    (true ? V = []).

attribute_property(Att,MethInit,MethGet,MethSet,Self)-State:=
    ( Att = list →
      MethInit = (V,Self)-State\method_apply(Self,[init_list(V)])-State,
      MethGet = (V,Self)-State\method_apply(Self,[get_list(V)])-State,
      MethSet = (V,Self)-State\method_apply(Self,[set_list(V)])-State).

domain(domain(X),Self)-State:=
    (true ? X = [list]).

insert_element(El,Self)-State:=
    ( true ? method_apply(Self,[get(list,L0)])-State,
      method_apply(Self,[set(list,L)])-State,
      method_apply(Self,[insert(El,L0,L)])-State.

insert(El,L0,L,Self)-State:=
    ( L0 = [] → L = [El]
    ; L0 = [E |L1] → method_apply(Self,[less(El,E,YesNo)])-State,
      insert_aux(El,E,YesNo,L1,L,Self)-State).

insert_aux(El,E,YesNo,L1,L,Self)-State:=
    ( YesNo = yes → L = [El,E |L1]
    ; YesNo = no → L = [E |L2],
      insert(El,L1,L2,Self)-State).

less(I1,I2,YesNo,Self)-State:=
    (I1 < I2 → YesNo = yes
    ;true → YesNo = no).

dispatch(Msg,Self)-State:=
    ( Msg = insert_element(Arg1) → insert_element(Arg1,Myself,Self)-State
    ; Msg = insert(Arg1,Arg2,Arg3) → insert(Arg1,Arg2,Arg3,Self)-State
    ; Msg = less(Arg1,Arg2,Arg3) → less(Arg1,Arg2,Arg3,Self)-State
    ; Msg = typeof(Arg1) → typeof(Arg1,Self)-State
    ; Msg = init_list(Arg1) → init_list(Arg1,Myself,Self)-State
    ; Msg = attribute_property(Arg1,Arg2,Arg3,Arg4) →
      attribute_property(Arg1,Arg2,Arg3,Arg4,Myself,Self)-State
    ; Msg = domain(Arg1) → domain(Arg1,Myself,Self)-State
    ; true → method_apply(Self,[typeof(Class)])-State,
      akl.stdout(S),
      io.format('~nMessage not understood: ~w ~nSelf: ~w ~nHandler: ~w~n',
      [MSG,Class,ord_list],S,_)).
```

□

Example 6.2 Consider the definition of class *ord_list_descending*, a subclass of *ord_list*, that follows. Note that sending the message *insert/3* to objects of this class will result in inserting the element in descending order. This is because the *less/3* is delegated to the base class which is *sort_descending* in this case.

```
:- class ord_list_descending.
:- supers [ord_list].
less(I1, I2, YesNo):=
    ( I1 > I2 → YesNo = yes
    ; true → YesNo = no).
```

The expansion of this class is as follows.

```
:- module ord_list_descending.
:- public dispatch/4.
:- public typeof/4.
:- public less/6.
:- public init_list/4.
:- public domain/4.
:- public attribute_property/7.

typeof(Class,Self)–State:=
    (true ? Class = ord_list_descending).

init_list(V,Self)–State:=
    (true ? V = []).

attribute_property(Att,MethInit,MethGet,MethSet,Self)–State:=
    ( Att = list →
      MethInit = (V,Self)–State\method_apply(Self,[init_list(V)])–State,
      MethGet = (V,Self)–State\method_apply(Self,[get_list(V)])–State,
      MethSet = (V,Self)–State\method_apply(Self,[set_list(V)])–State).

domain(domain(X),Self)–State:=
    (true ? X = [list]).

less(I1,I2,YesNo,Self)–State:=
    ( I1 > I2 → YesNo = yes
    ; true → YesNo = no).
```

```
dispatch(Msg,Self)–State:=
    ( Msg = less(Arg1,Arg2,Arg3) → less(Arg1,Arg2,Arg3,Self)–State
    ; Msg = insert(Arg1,Arg2,Arg3) → ord_list.insert(Arg1,Arg2,Arg3,Self)–State
    ; Msg = typeof(Arg1) → typeof(Arg1,Self)–State
    ; Msg = init_list(Arg1) → init_list(Arg1,Myself,Self)–State
    ; Msg = attribute_property(Arg1,Arg2,Arg3,Arg4) →
      attribute_property(Arg1,Arg2,Arg3,Arg4,Myself,Self)–State
    ; Msg = domain(Arg1) → domain(Arg1,Myself,Self)–State
    ; true → method_apply(Self,[typeof(Class)])–State,
      akl.stdout(S),
      io.format('~nMessage not understood: ~w ~nSelf: ~w ~nHandler: ~w~n',
      [MSG,Class,ord_list_descending],S,_)).
```

□

Example 6.3 Assume that the class *ord_list* is parameterized by a class abstraction that defines the method *less/3*, as follows.

```
:- class ord_list(Order).
:- private [insert_aux/5].
:- attributes [list=(V)\\(V=[])].

insert_element(El):=
    ( true ? get(list, L0),
      set(list, L),
      insert(El,L0,L)).

insert(El, L0, L):=
    ( L0 = [] → L = [El]
    ; L0 = [E|L1] → less(El,E,YesNo) # Order,
      insert_aux(El, E, YesNo, L1, L) # ord_list(Order)).

insert_aux(El, E, YesNo, L1, L):=
    ( YesNo = yes → L = [El,E|L1]
    ; YesNo = no → L = [E|L2],
      insert(El, L1, L2) # ord_list(Order)).

less(I1, I2, YesNo):=
    ( I1 < I2 → YesNo = yes
    ; true → YesNo = no).
```

The expansion of this class is as follows.

```
:- module ord_list.
:- public dispatch/5.
:- public typeof/5.
:- public less/7.
:- public insert_element/5.
:- public insert/7.
:- public init_list/5.
:- public domain/5.
:- public attribute_property/8.

typeof(Class,Myself,Self)–State:=
    (true ? Class = ord_list).

init_list(V,Myself,Self)–State:=
    (true ? V = []).

attribute_property(Att,MethInit,MethGet,MethSet,Myself,Self)–State:=
    ( Att = list →
      MethInit = (V,Self)–State\method_apply(Self,[init_list(V)])–State,
      MethGet = (V,Self)–State\method_apply(Self,[get_list(V)])–State,
      MethSet = (V,Self)–State\method_apply(Self,[set_list(V)])–State).

domain(domain(X),Myself,Self)–State:=
    (true ? X = [list]).

insert_element(El,Myself,Self)–State:=
    ( true ? method_apply(Self,[get(list,L0)])–State,
      method_apply(Self,[set(list,L)])–State,
      method_apply(Self,[insert(El,L0,L)])–State.

insert(El,L0,L,Myself,Self)–State:=
    ( L0 = [] → L = [El]
    ; L0 = [E |L1] →
      Myself = ord_list(Order),
      method_apply(Order,[less(El,E,YesNo)])–State,
      insert_aux(El,E,YesNo,L1,L,Myself,Self)–State).
```

```
insert_aux(El,E,YesNo,L1,L,Myself,Self)–State:=
    ( YesNo = yes → L = [El,E |L1]
    ; YesNo = no → L = [E |L2],
      insert(El,L1,L2,Myself,Myself,Self)–State).

dispatch(Msg,Myself,Self)–State:=
    ( Msg = insert_element(Arg1) → insert_element(Arg1,Myself,Myself,Self)–State
    ; Msg = insert(Arg1,Arg2,Arg3) → insert(Arg1,Arg2,Arg3,Myself,Self)–State
    ; Msg = less(Arg1,Arg2,Arg3) → less(Arg1,Arg2,Arg3,Myself,Self)–State
    ; Msg = typeof(Arg1) → typeof(Arg1,Myself,Self)–State
    ; Msg = init_list(Arg1) → init_list(Arg1,Myself,Myself,Self)–State
    ; Msg = attribute_property(Arg1,Arg2,Arg3,Arg4) →
      attribute_property(Arg1,Arg2,Arg3,Arg4,Myself,Myself,Self)–State
    ; Msg = domain(Arg1) → domain(Arg1,Myself,Myself,Self)–State
    ; true → method_apply(Self,[typeof(Class)])–State,
      akl.stdout(S),
      io.format('~nMessage not understood: ~w ~nSelf: ~w ~nHandler: ~w~n',
      [MSG,Class,ord_list],S,_)).
```

□

6.3.2 The Class Expansion

The transformation rules given in Section 5.3 provide the definitions of general rewrite rules for expanding definitions. These rules can be optimized for an efficient implementation. In the implementation, the definition to which a method expands depends on whether or not the class being defined is parameterized:

$$\mathcal{M}ethod : \begin{cases} \text{Message} \times \text{Myself} \times \text{Self} \times \text{State} \rightarrow \text{State} & if\ parameterized\ class \\ \text{Message} \times \text{Self} \times \text{State} \rightarrow \text{State} & otherwise \end{cases}$$

Where the argument *Message* is the received message (a method atom), the argument *Myself* is the class atom of the class being defined, the argument *Self* is an abstraction of the base class, the argument *State* is the state at the time of message reception, and the (output) argument *State* is the state that results from the method activation.

The difference arises from the need to expand the parameterized class methods with a parameter that will hold the class parameters which is not needed in case of a non-parameterized class. This definition is further optimized when considering the first argument indexing of the method code. In the following sections a variant

of the transformations rules and the method dispatcher that flavors efficiency are discussed.

In order to avoid redundancy, the transformation rules that follows consider only the rules that differs from the ones given in Section 5.3.

Method definition transformation rule

$$head := body \Rightarrow \left\{ \begin{array}{l} head' := body' \\ :- \textbf{public } p/n \end{array} \right\}$$

where *head* translates to *head′* according to the *head transformation* rule and *body* translates to *body′* according to the *statement transformation* rule. p and n are the functor and arity of *head′*, respectively. The **public** definition is to be defined if the definition is visible so that definitions can call it from within other modules. All methods of a class are visible from outside unless they are declared to be hidden (invisible) by a class **private** definition. Every visible method takes a **public** definition. The arity of *head′* is determined as follows:

$$n = \left\{ \begin{array}{ll} 4 + arity & if\ parameterized\ class \\ 3 + arity & otherwise \end{array} \right.$$

Where *arity* is the arity of *head*, *3* corresponds to the arguments: *Self, State,* and *(output) State* and *4* corresponds to the arguments: *Myself, Self, State,* and *(output) State.*

Head transformation rule

- $p(X_1, \ldots, X_n) \Rightarrow p(X_1, \ldots, X_n, Myself, Self)\text{–}State$
  If the class being defined is parameterized.
- $p(X_1, \ldots, X_n) \Rightarrow p(X_1, \ldots, X_n, Self)\text{–}State$
  If the class being defined is non-parameterized.

Clause transformation rule

The rule

$$X_1, \ldots, X_n : statement \ \%\ statement \Rightarrow X_1, \ldots, X_n : statement' \ \%\ Myself = Class,\ statement'$$

Where % is a guard operator, *Class* refers to the class atom of the class being defined, $X_1, \ldots, X_n$ are variables that have a scope limited to the statement being defined, and *statement* translates to *statement'* according to the *statement transformation rules.*

is augmented with the rules

- $X_1, \ldots, X_n$:*statement* % *statement* $\Rightarrow$ $X_1, \ldots, X_n$:*statement'*% *statement'*

  If the class being defined is non-parameterized.

- $X_1, \ldots, X_n$: *statement* % *statement* $\Rightarrow$ $X_1, \ldots, X_n$: *statement'*% *statement'*

  If the class being defined is parameterized and its parameters is a subset of $\{X_1, \ldots, X_n\}$.

- $X_1, \ldots, X_n$: $statement_1$%$statement_2$ $\Rightarrow$ $X_1, \ldots, X_n$: $statement_1$'%$statement_2$'

  If the class being defined is parameterized and its parameters is not used in $statement_1$ or $statement_2$.

The basic idea is that only clauses of methods that potentially uses or passes any of the class parameters will translate with the following (binding) statement to unfold the class parameters:

Myself = *Class*

Where *Class* refers to the class atom of the class being defined. *Class* has to be present to cater for sharing of class parameters.

Method delegation transformation rules

- $p(X_1, \ldots, X_n)$ # *Class* $\Rightarrow$ $p(X_1, \ldots, X_n$,*Myself,Self)–State*

  If *Class* refers to the class atom of the class being defined and the method atom $p(X_1, \ldots, X_n)$ is defined locally in the parameterized class *Class.*

- $p(X_1, \ldots, X_n)$ # *Class* $\Rightarrow$ $p(X_1, \ldots, X_n$,*Class,Self)–State*

  If *Class* refers to the class atom of the class being defined with different parameterers and the method atom $p(X_1, \ldots, X_n)$ is defined locally in the parameterized class *Class.*

- $p(X_1, \ldots, X_n)$ # *Class* $\Rightarrow$ $p(X_1, \ldots, X_n$,*Self)–State*

  If *Class* refers to the class atom of the class being defined and the method atom $p(X_1, \ldots, X_n)$ is defined locally in the non-parameterized class *Class.*

- *goal # Class ⇒ dispatch(goal,Myself,Self)–State*

  If *Class* refers to the class atom of the class being defined and *goal* is a method atom not defined locally in the parameterized class *Class*.

- *goal # Class ⇒ dispatch(goal,Class,Self)–State*

  If *Class* refers to the class atom of the class being defined with different parameters and *goal* is a method atom not defined locally in the parameterized class *Class*.

- *goal # Class ⇒ dispatch(goal,Self)–State*

  If *Class* refers to the class atom of the class being defined and *goal* is a method atom not defined locally in the non-parameterized class *Class*.

- *goal # Class ⇒ m.dispatch(goal,Class,Self)–State*

  If *Class* refers to a parameterized class other than the class being defined and m is its functor.

- *goal # Class ⇒ m.dispatch(goal,Self)–State*

  If *Class* refers to a non-parameterized class other than the class being defined and m is its functor.

Method invocation transformation rules

- $p(X_1,\ldots,X_n)$ *<# Class ⇒* $Self' = (M)$*–State\m.dispatch(M,Class,Self')–State,* $p(X_1,\ldots,X_n,Myself,Self')$*–State*

  If *Class* refers to the class atom of the class being defined and m is its functor, and the method atom $p(X_1,\ldots,X_n)$ is defined locally in the parameterized class *Class*.

- $p(X_1,\ldots,X_n)$ *<# Class ⇒* $Self' = (M)$*–State\m.dispatch(M,Class,Self')–State,* $p(X_1,\ldots,X_n,Class,Self')$*–State*

  If *Class* refers to the class atom of the class being defined with different parameters and m is its functor, and the method atom $p(X_1,\ldots,X_n)$ is defined locally in the parameterized class *Class*.

- $p(X_1,\ldots,X_n)$ *<# Class ⇒* $Self' = (M)$*–State\m.dispatch(M,Self')–State,* $p(X_1,\ldots,X_n,Self')$*–State*

  If *Class* refers to the class atom of the class being defined and m is its functor, and the method atom $p(X_1,\ldots,X_n)$ is defined locally in the non-parameterized class *Class*.

- $goal <\# Class \Rightarrow\ Self' = (M)\text{–}State \backslash m.dispatch(M,Class,Self')\text{–}State,$
  $dispatch(goal,Myself,Self')\text{–}State$

  If *Class* refers to the class atom of the class being defined and m is its functor, and *goal* is a method atom not defined locally in the parameterized class *Class*.

- $goal <\# Class \Rightarrow\ Self' = (M)\text{–}State \backslash m.dispatch(M,Class,Self')\text{–}State,$
  $dispatch(goal,Class,Self')\text{–}State$

  If *Class* refers to the class atom of the class being defined with different parameters and m is its functor, and *goal* is a method atom not defined locally in the parameterized class *Class*.

- $goal <\# Class \Rightarrow\ Self' = (M)\text{–}State \backslash m.dispatch(M,Self')\text{–}State,$
  $dispatch(goal,Self')\text{–}State$

  If *Class* refers to the class atom of the class being defined and m is its functor, and *goal* is a method atom not defined locally in the non-parameterized class *Class*.

- $goal <\# Class \Rightarrow\ Self' = (M)\text{–}State \backslash m.dispatch(M,Class,Self')\text{–}State,$
  $m.dispatch(goal,Class,Self')\text{–}State$

  If *Class* refers to a parameterized class other than the class being defined and m and is its functor.

- $goal <\# Class \Rightarrow\ Self' = (M)\text{–}State \backslash m.dispatch(M,Self')\text{–}State,$
  $m.dispatch(goal,Self')\text{–}State$

  If *Class* refers to a non-parameterized class other than the class being defined and m and is its functor.

Class abstraction transformation rule

- $Z = \# Class \Rightarrow Z = (M)\text{–}State \backslash m.dispatch(M,Class,Self)\text{–}State$

  If *Class* refers to a parameterized class and m is its functor.

- $Z = \# Class \Rightarrow Z = (M)\text{–}State \backslash m.dispatch(M,Self)\text{–}State$

  If *Class* refers to a non-parameterized class and m is its functor.

The Attributes Transformation

Attributes transformation result in several method definitions: attributes reference, attributes initializations, attributes properties besides attributes access and update

methods if any of the standard state classes is inherited by the class being defined. A user defined state is possible whereas the definition of attributes access and update methods should be provided by user. For simplicity, we will describe this transformation in terms of the AKL+ definitions. The expansion of these definitions to AKL code is straightforward according to the transformations rules discussed so far.

AKL+ supports two alternative representations of the object's state: hash table provided by the standard class *state_hash*, and array provided by the standard class *state_array*, see Section 4.3.1. Classes can choose to inherit any of them.

Attributes with their initialization methods are given in the **attributes** declaration of the class definition as follows:

:- **attributes** $[a_1{=}(X_1)\backslash\backslash b_1,\ldots,a_n{=}(X_n)\backslash\backslash b_n]$

where $a_i, 1 \leq i \leq n$, is an attribute name and b_i is its initialization method.

Assume $m - n \geq 0$ inherited attributes. Then, this definition expands to the following method definitions:

1. *domain(X)* := **true** *?* $X = [a_1,\ldots,a_m]$

   where $[a_1,\ldots,a_m]$, $n \leq m$, is a list which enumerates the defined and inherited attribute names. An activation of this method will return this list.

2. $init\_a_i\ (X_i)$:= **true** *?* b_i

   where $init\_a_i/1, 1 \leq i \leq m$, is the initialization method associated with the attribute a_i and X_i is its (output) argument. An activation of this method will return the initial value of this attribute.

3. *attribute_property(A,Init,Get,Set)* :=

   $(A = a_1 \rightarrow Init = (V)\backslash\backslash\ init\_a_1,$
   $\qquad Get = (V)\backslash\backslash\ get\_a_1,$
   $\qquad Set = (V)\backslash\backslash\ set\_a_1,$
   $; \ldots$
   $; A = a_m \rightarrow Init = (V)\backslash\backslash\ init\_a_m,$
   $\qquad Get = (V)\backslash\backslash\ get\_a_m,$
   $\qquad Set = (V)\backslash\backslash\ set\_a_m)$

   where the definition of the method *attribute_property/4* tabulates the method abstractions of initialize, access, and update methods of each attribute. An activation of this method given such an attribute will return these abstractions.

An application of any of these abstractions to its argument will execute the corresponding definition.

4. for a user defined state, the user is responsible to provide the access and update methods: *get_a_i*/1 and *set_a_i*/1, respectively, for each $a_i, 1 \leq i \leq m$, attribute.

5. for a standard state class the attributes access and update methods are added, one for each $a_i, 1 \leq i \leq m$, attribute:

   - for the hash table state representation, each attribute will be a key that is used to access its value. The definition of a class that inherits from the standard state class *state_hash* adds the definition of the following methods:
     - *get_a_i*(V):= **true** ? get(a_i,V)^State
     - *set_a_i*(V):= **true** ? set(a_i,V)^State

     where the methods *get_a_i*/1 and *set_a_i*/1 are the methods added for direct attribute (key) access and update of the attribute a_i's value.
   - for the array state representation, the compiler generates an index (numeric value) for each attribute and all attribute access are performed through this index. The definition of a class that inherits from the standard state class *state_array* adds the definition of the following methods:
     - *attribute_a_i(Index)* := **true** *? Index = array_index*
     - *get_a_i(V)*:= **true** *? get(array_index,V)^State*
     - *set_a_i(V)*:= **true** *? set(array_index,V)^State*

     where the methods *get_a_i*/1 and set_a_i/1 are the methods added for direct attribute (indexed) access and update of the attribute a_i's value. The *array_index* is the numeric value corresponding to the attribute a_i. *attribute_a_i*/1 refers to the array index of the attribute a_i.

Attribute access, update and initialization methods are like any other methods, they may be specialized in subclasses. The definition of each of which will be taken from the most specialized definition. It is not only possible to specialize the access and update methods for an attribute but also it is possible to specialize the *generic* access and update methods for all attributes. The *get/2* and *set/2* methods are chosen for specifying the generic specialization of the access and update methods of all attributes. As classes can be defined as *Mixins* where the access and update method activations may call the attribute access and update methods of each attribute or may call the generic attribute access and update methods, the compiler detects whether the definition of the generic method is absent. If so, the compiler optimizes the generic access and update method activations as follows:

- for the standard state class *state_array*, these activations are transformed according to the following rules:

- $set(a_i,\ V) \Rightarrow set\_a_i(V)$
- $get(a_i,\ V) \Rightarrow get\_a_i(V)$

- for the standard state class *state_hash*, these activations are transformed according to the following rules:
  - $set(a_i,\ V) \Rightarrow set(a_i,V)\hat{}State$
  - $get(a_i,\ V) \Rightarrow get(a_i,V)\hat{}State$

The Method Dispatcher

The method dispatcher is the entry point to the method handler. It is used for all method invocations, except for cases where it can be determined at compile time that the method call is to a method that is locally defined in the class. The dispatcher is implemented as a conditional choice statement that will select the appropriate handler of the method which is a class that will service the method call. The conditional statement is deterministic so that it enforces the inheritance mechanism. The dispatcher will be described using the notational conventions that identifiers beginning with a lowercase letter are constants; those with an initial upper-case letters are variables.

As a consequence of variant expansion of class methods according to whether or not the class is parameterized, the definition of the method dispatcher is as follows:

$$\mathcal{D}ispatch : \begin{cases} \text{Message} \times \text{Myself} \times \text{Self} \times \text{State} \rightarrow \text{State} & if\ parameterized\ class \\ \text{Message} \times \text{Self} \times \text{State} \rightarrow \text{State} & otherwise \end{cases}$$

Accordingly, the method head is one of the following forms:

- *dispatch(Msg,Myself,Self)–State*

  If the class is parameterized.

- *dispatch(Msg,Self)–State*

  If the class is non-parameterized.

The dispatcher body consists of several statements of the following form:

- if the method msg_i/n is defined locally in a parameterized class, the statement is an expression of the form:

$$Msg = msg_i(Arg_1,\ldots,Arg_n) \rightarrow$$
$$\quad msg_i(Msg,Myself,Self)\text{–}State.$$

- if the method msg_i/n is defined locally in a non-parameterized class, the statement is an expression of the form:

$$Msg = msg_i(Arg_1,\ldots,Arg_n) \rightarrow$$
$$\quad msg_i(Msg,Self)\text{–}State.$$

- if the method msg_i/n is inherited from a parameterized ancestor class, say *Ancestor*, to a parameterized class, the statement is an expression of the form:

$$Msg = msg_i(Arg_1,\ldots,Arg_n) \rightarrow$$
$$\quad Myself = Class,$$
$$\quad Handler = Ancestor,$$
$$\quad ancestor.msg_i(Msg,Handler,Self)\text{–}State.$$

  where *Class* is the class atom of the class being defined, *Ancestor* is the class atom of the ancestor class that msg_i/n is inherited from, and *ancestor* is the functor of *Ancestor*. *Class* and *Ancestor* must be present to cater for sharing of class parameters.

- if the method msg_i/n is inherited from a parameterized ancestor class, say *Ancestor*, to a non-parameterized class, the statement is an expression of the form:

$$Msg = msg_i(Arg_1,\ldots,Arg_n) \rightarrow$$
$$\quad ancestor.msg_i(Msg,Ancestor,Self)\text{–}State.$$

  where *Ancestor* is the class atom of the ancestor class that msg_i/n is inherited from, and *ancestor* is the functor of *Ancestor*.

- if the method msg_i/n is inherited from a non-parameterized ancestor class, say *Ancestor*, the statement is an expression of the form:

$$Msg = msg_i(Arg_1,\ldots,Arg_n) \rightarrow$$
$$\quad ancestor.msg_i(Msg,Self)\text{–}State.$$

  where *ancestor* is the functor of the ancestor class that msg_i/n is inherited from.

A final clause is the default method that will be invoked if no matching method is found. In the absence of a user-defined default method, *$default/1*, the following final clause is automatically added:

```
true →
        method_apply(Self,[typeof(Class)])–State,
        akl.stdout(S),
        io.format('~nMessage not understood: ~w ~nSelf: ~w ~nHandler:~w~n',
        [Msg,Class,ord_list],S,_)).
```

6.3.3 Efficiency

Object-Oriented languages have an undeserved reputation for inefficiency because some early languages (Smalltalk and Lisp-based languages) were interpreted rather than compiled [121]. The AKL+ language is a compiled language that expands classes into AKL code. The language is provided with mature standard classes. Two sets of standard classes are supported: object type classes and state classes. The standard class *object* defines port based objects which are active (heavy weight) objects communicated through ports. The standard class *cell* defines data objects which are very fine-grained (light weight) objects that provides a minimal form of encapsulated state. An association of the synchronization constraints on message acceptance protocol to any of the other types is possible. The standard class *sync_object* specifies an association between a port based object and any of the standard protocols. Two standard protocols are provided: *synchronizers* and *transitions*. The main advantage of these protocols is the clean separation of concurrency control and the method specification such that they can be inherited, overridden, or extended separately without affecting each other. Moreover, one scheme can be integrated and composed with other schemes. Two standard state representation classes are supported which provide efficiency in representing the object's state and its access and update operations on attributes. The hash table representation described by the standard state class *state_hash*, provides a direct attribute (key) access to the attribute's value. The array representation described by the standard state class *state_array*, provides a direct attribute (indexed) access to the attribute's value.

One aspect of object-oriented languages that seems inefficient is the use of *method resolution* at run-time (also known as dynamic binding) to invoke methods. Method resolution is the process of matching an operation on an object to a specific method. This would seem to require a search up the inheritance graph at run-time to find the class that implements the operation. AKL+ optimizes the look-up mechanism to make it more efficient; a method dispatches in a constant time once its target class becomes determinate regardless of the depth of the inheritance graph or the number of methods in the class. Moreover, the dispatch table is cleanly captured and will only contain the relevant information where all the excluded entries are removed.

The good programming styles that are employed on the AKL level can also be employed on the AKL+ level. This language efficiency is realized by:

1. class representation as a special light-weight AKL module. This representation facilitates efficient encapsulation of class primitives and enhances the code execution through direct accessing of the class primitives.

2. the method dispatcher exploits the first argument indexing of the AKL compiler, leading to direct access to the method clauses.

3. as a consequence to 1) & 2) the AKL first argument indexing for methods is preserved.

4. the unfolding of class parameters is only performed for the method clause that uses or passes any of these parameters.

5. preserving the last call optimization in recursive methods: the tail primitive is expanded into tail recursive code. A tail recursive primitive is a definition that invokes itself, i.e. calls a definition to itself in the class being defined. The expanded code will invoke the expanded code directly instead of calling the dispatcher. Hence, the tail recursive primitive definition is expanded into tail recursive AKL code.

6. enforce the override mechanism by applying the AKL conditional choice primitive to the class dispatcher.

7. a method invocation during the execution of a message to an object may be directly applied to the state without the need to schedule this message to the target object.

6.4 Expansion to AKL Abstract Machine(PAM)

The AKL implementation was based on a compiler. The AKL program is to be compiled to a high level abstract machine similar in spirit to the Warren Abstract Machine (WAM) [148, 59]. By keeping the abstract machine close to WAM, it was possible to adapt the Prototype Abstract Machine (PAM) with minor changes only, and is still written in Prolog. Since the target is not a real machine, AKL supports an emulator of the abstract machine. The AKL compiler takes an AKL program as input produces abstract machine instructions as output. We have omitted the description of PAM since it doesn't concern our work, for more details see [74, 72].

6.5 The AKL+ Run-time

In order to achieve a run-time uniform message sending for objects in AKL+, we have defined the module *meta*. This module consists of definitions that execute message sending no matter to whichever object type the message is sent. Messages to objects are always dynamically bound since objects are dynamically created and their types will not be known until run-time. The following is the description of this module.

```
:- module meta.
:- public send/3.
:- public send_cell/3.
:- public cell/1.
:- public create_cell_object/3.

send(Msg)–Obj:=
    ( akl.port(Obj) → akl.send(Msg)–Obj
    ; akl.abstraction(Obj) → send_cell(Msg)–Obj).

send_cell(Msg,Object,NewObject) :=
    (true → akl.Object(Msg,NewObject)).

cell(C) :=
    (true → akl.new_array(1,C)).

create_cell_object(C,Class,Object) :=
    (true → Object=(Msg,NewObject)\call_method(Msg,C,Class,Object,NewObject)).

call_method(Msg,C,CD,Object,NewObject) :=
    ( true → exchange(NewState,State,C,NewC),
      set_flag(NewC,Object,NewObject),
      akl.CD(Msg,State,NewState)).

exchange(New,Old,C,NewC) :=
    (true → akl.send(set(0,Old,New),C,NewC)).

set_flag(NewC,Object,NewObject) :=
    (akl.port(NewC) → NewObject = Object).
```

Where:

- *send/3*: is a generic definition to the message sending such that a message is automatically sent to an object according to the type of the target object.

For objects derived from the standard class *object*, the message is to be sent through the primitive *akl.send/3* of the underlying language. For objects derived from the standard class *cell*, the *send_cell/3* primitive is defined for this purpose.

- *send_cell/3*: is defined for applying the message directly to the cell object.
- *cell/1* is defined to create a cell object that will hold the state.
- *create_cell_object/3* is defined to create the cell construct as an abstraction. This abstraction calls *call_method/5* that holds the cell and the method dispatcher. Messages sent to the cell object are applied to this abstraction.
- *call_method/5*: is defined for dispatching methods of a cell object. The state is read and the new state is written simultaneously to the cell. The cell object is ready to accept messages just after the new cell is created.
- *exchange/3*: is defined to read the state and write the new state of the cell.
- *set_flag/3*: is defined to synchronize the accessing of the cell upon receipt of message

6.6 Summary

Concepts and implementation aspects of an incremental compilation facility have been presented. The logical structure of object-oriented software systems is very similar regardless of the programming language used. Only the notation is different in Eiffel, Smalltalk, C++, and AKL+ for example. However, programs written in these languages consist of classes and methods and use the inheritance mechanism. Therefore the concepts presented so far are applicable and advantageous to all of these languages. This means that the presented programming environment should be usable with most other object-oriented languages. In fact, it was designed with these considerations in mind to support various languages with minor changes.

Obviously each programming environment has to have a part specialized for the supported language. We tried to bundle all language-specific code into a separate dependency, the language-dependent translator, which may be replaced completely and easily by another language-dependent version. Efficiency is one of our prime goals. This is realized through the schemes developed for the optimization of the code to which a class is expanded, and the efficiency offered by the set of standard classes.

Programming Examples

This chapter presents examples for typical problems in object-oriented concurrent computing. The examples are based on message passing among concurrent objects. The structure and behavior of these objects are described in our programming language AKL+ by their class definitions. Several programming techniques in concurrent object-oriented programming, which include call-back messages, searching an object hierarchy, and concurrent objects synchronized by satisfying constraints on the acceptance of their messages, are employed in the examples.

The chapter is organized as follows. In Section 7.1 the activities of a car washer is taken as an example for demonstrating discrete-event simulation. In Section 7.2 a moving point in a two dimensional space is taken as an example for demonstrating region constraints. In Section 7.3 a car maintenance is taken as an example for demonstrating fault diagnosis. In Section 7.4 the dynamic locking of a bounded buffer is taken as an example for demonstrating the modification of acceptable states.

7.1 A Car Washer Simulation

Discrete event simulation is one of the major application areas of object-oriented concurrent programming. In general, each physical or conceptual entity in simulation is represented as an object. In this manner, a natural modeling for simulation can be obtained. In this section, we consider a typical discrete event simulation problem. We call it the *car washer* problem and its specification is given below.

Problem Description

There is a company where several workers wash cars. Everyday cars arrive at the company one by one at random intervals. Every worker washes cars according to the following rule:

1. if some workers are idle, the car will be washed.
2. otherwise (i.e. every worker is washing a car), the car will be washed by the worker who will finish the current work earliest.

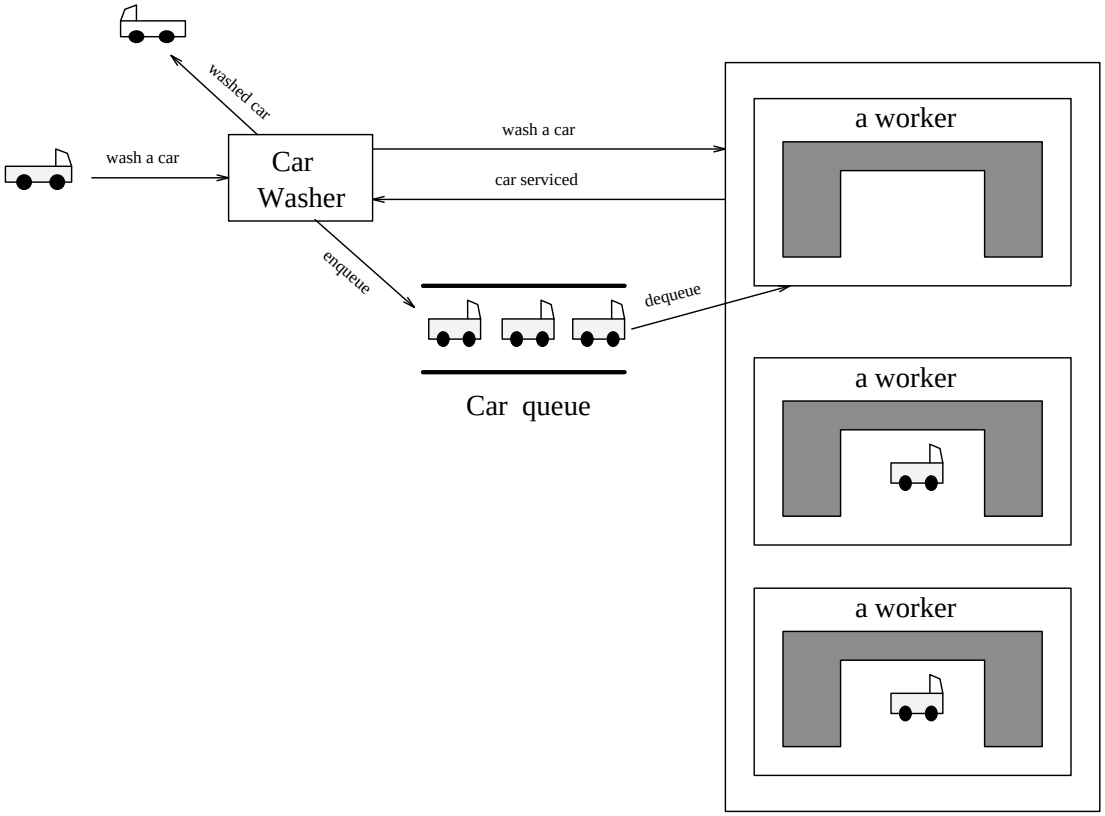

Figure 7.1: Existing interaction in the Car Washer problem

The classes

The design of our model for the car washer problem is characterized by being a general-purpose package which could be extended easily to handle other discrete-event simulation problems. Our simulation package consists of the classes *randomize, scheduler, queue, out,* and *simulation.* Entities in the problem domain are identified by the classes *car_washer* and *car_generator.* Fig. 7.1 illustrates the interactions among various entities in the problem domain. Within the car washing certain events can be recognized; the arrival of cars, their joining and leaving of a queue, being served by a car washer, and finally leaving the car washer. Arrivals are of random duration, and will be modeled accordingly. A single communal queue is used for all the workers. The input parameters for each simulation are the number of workers and the service time (a time taken by a worker to serve a car).

Note that we omit any description of a car since it only receives a message representing an interaction for washing and sends nothing. The car do not have any active role in the car wash problem. They are regarded as passive data (an AKL+ immutable object).

The following sections summarize the protocol (attributes and methods) for the various classes in the car washer simulation.

Randomize

The class *randomize* with class *ur_object* as its super is defined for generating random numbers. Its state contains the attribute *r*. Its behavior consists of the methods: *next_random/1* and *terminate/0*. To compute the initial value of the attribute *r*, the built-in procedure (agent definition) *random/2* defined in the standard module *random* is invoked. This agent generates a random number between 0 and 9. Then, the attribute *r* is initialized to a stream, i.e. setting a communication medium between the agent and the object. The *next_random/1* generates a successive random number. Method *terminate/0* will close the stream, and the communication will no longer exist.

```
:- class randomize.
:- supers [ur_object].
:- attributes [r=(Stream)\\random.random(10,Stream)].

next_random(R) :=
    ( true → get_r([R|Stream]),
      set_r(Stream)).

terminate :=
    (true → get_r([])).
```

Scheduler

In discrete-event simulation, there is an implicit sequencing between activities based on the simulated time for events. This is realized by limiting the inherent concurrency of events to ensure that next event will not start unless the activity of the current event reaches an end. The class *scheduler* with class *ur_object* as its super is defined for scheduling the events of the car washing. Its state contains the attributes: *counter* initialized with *0*, *ready_queue* initialized with an empty list, and *current_time* initialized with *0*. The attribute *counter* keeps information about the currently ongoing activities. The attribute *ready_queue*, a priority queue, stores a list of doublets ⟨Scheduled event time, Alarm⟩ ordered by increasing event time; where *Alarm* is a method abstraction of a call-back message that starts the next event. The *current_time* stores the simulation time. Its behavior consists of the methods:

- *initial_event/1*: schedules the initial event of the simulation.
- *hold/2*: suspends a call-back message for a *Duration* units of simulation time. This is implemented by scheduling *ready_queue* for the reactivation time of the call-back message.

- *current_time/1*: returns the current simulated time.
- *register/0*: records the start of an activity by incrementing *counter* by one.
- *unregister/[0-1]*: records the end of an activity by decrementing *counter* by one.
- *chk_next/1*: determines whether or not the next event is to start.
- *alarm/1*: picks the first element from *ready_queue*, i.e. selects the next event. Then, advances the simulation time to the scheduled time of this event and starts the next event by applying the method abstraction to this time.

```
:- class scheduler.
:- supers [ur_object].
:- attributes [counter = (N)\\(N=0), ready_queue=(Q)\\(Q=[]),
               current_time = (T)\\(T=0)].

initial_event(Alarm) :=
    ( true → register,
      hold(0,Alarm)).

hold(Duration,Alarm) :=
    ( true → get_current_time(T),
      NT is Duration+T,
      get_ready_queue(Q),
      sort.add_element((NT,Alarm))-Q,
      set_ready_queue(Q),
      unregister).

current_time(T) :=
    (true → get_current_time(T)).

register :=
    ( true → get_counter(N),
      N1 is N+1,
      set_counter(N1)).

unregister :=
    ( true → get_counter(N),
      unregister(N)).

unregister(N) :=
    ( N = 0 → true
```

```
    ; N > 0 → N1 is N-1,
      set_counter(N1),
      chk_next(N1)).

chk_next(N) :=
    ( N = 0 → get_ready_queue(Q),
      alarm(Q)
    ; N > 0 → true).

alarm(Q) :=
    ( Q = [] → true
    ; Q = [(T,Alarm)|QRest] → set_current_time(T),
      register,
      set_ready_queue(QRest),
      Alarm(T)).
```

Queue

The class *queue* with class *ur_object* as its super is defined for the abstract data type (ADT) queue operations, so-called FIFO queue. The principal properties of a FIFO queue are that elements join and leave at opposite ends, namely at the end of the queue and from the beginning of the queue, respectively. The most obvious way to represent a queue in AKL+ is through difference lists [136]. This is because we need fast access to both ends of the list, which this representation gives. The class *queue* has the attribute *contents* for storing elements, e.g. cars, in the queue. Its behavior consists of the methods: *enqueue/1* for inserting a new element at the end of the queue and the method *dequeue/[2-4]* for removing the oldest element from the front of the queue.

```
:- class queue(Scheduler).
:- supers [ur_object].
:- attributes [contents=(Q)\\(Q=(X,X))].

enqueue(E):=
    ( true → get_contents(Q), Q=(F,[E|R]),
      unregister^Scheduler,
      set_contents((F,R))).

dequeue(E,Flag):=
    ( true → get_contents(Q),
      dequeue(Q,E,Flag)).
```

```
dequeue(Q,E,Flag):=
    ( Q = (X,X) | Flag = empty
    ; Q = ([E1|F],R) | E1 = E,
      Flag = partial,
      set_contents((F,R)),
      register^Scheduler).
```

Out

The class *out* with class *ur_object* as its super is defined for displaying information. For simplicity, the class *out* has the method *out/1* that displays results as soon as it gets ready. In car washer simulation, this method displays the following information about the car: the arrival time, rescheduled time, and the time after it gets the service.

```
:- class out(Scheduler).
:- supers [ur_object].

out(E) :=
    ( true → unregister^Scheduler,
      write.gwriteln(E)).
```

Simulation

The class *simulation* is the functional interface to the simulation package. It is a generic class parameterized with a scheduler object. The power of the generic classes makes it possible to have simulation according to different scheduling algorithms. The default behavior of this class is to relay messages to the scheduler object.

```
:- class simulation(Scheduler).
:- supers [ur_object].

Msg :=
    (true → Msg ^ Scheduler).
```

Car Generator

The class *car_generator* with class *simulation* as its super is defined for generating cars at random duration. Its behavior consists of the methods: *init/0* for initiating a car arrival event of the simulation and *generate/2* for processing the arrival of a car and generating the arrival of the next car. Both *init/0* and *generate/2* methods use the power of *self-reference* to define a method abstraction that sends a call-back message. This method is scheduled for reactivation for a given duration. As soon as the car is generated, it is sent to the *CarWasher* object.

```
:- class car_generator(CarWasher,Random,Scheduler).
:- supers [simulation(Scheduler)].

init :=
    ( true → get_self(S),
      initial_event((T)\\(generate(T,1)^S))).

generate(CurrentTime, N) :=
    ( CurrentTime > 10000 → terminate^Random
    ; true → get_self(S),
      register,
      car([CurrentTime], N)^CarWasher,
      next_random(R)^Random,
      N1 is N+1,
      hold(R, (T)\\(generate(T, N1)^S))).
```

Car Washer

The class *car_washer* with class *simulation* as its super is defined for serving cars. Its state contains the attributes: *car_queue* initialized with an object of class *queue* for keeping the waiting cars, *avail_workers* initialized by the number of workers that was given as input parameter to the simulation. Its behavior consists of the methods:

- *car/[2-3]*: determines whether to serve the arrived car or insert it in *car_queue* according to the availability of the workers.
- *car_serviced/2*: terminates the current event, i.e. a car is to leave the car washer, and try to dequeue a waiting car from *car_queue*.
- *next_car/3*: schedules the dequeued car for arrival.

```
:- class car_washer(NoOfWorkers,ServiceTime,Out,Scheduler).
:- supers [simulation(Scheduler)].
:- attributes [car_queue = (Q)\\(new(Q)<#queue(Scheduler)),
               avail_workers = (N)\\(N=NoOfWorkers)].

car(TL, I) :=
    ( true → get_avail_workers(N),
      car(TL,I, N)).

car(TL, I, N) :=
    ( N = 0 → get_car_queue(Q),
      enqueue(car(TL, I))^Q,
      set_car_queue(Q)
    ; N > 0 → N1 is N-1,
      set_avail_workers(N1),
      get_self(S),
      hold(ServiceTime, (ST)\\car_serviced([ST|TL],I)^S)).

car_serviced(TL, I) :=
    ( true → register,
      get_avail_workers(N),
      N1 is N+1,
      set_avail_workers(N1),
      out(car(TL, I))^Out,
      get_car_queue(Q),
      dequeue(E,Flag)^Q,
      current_time(T),
      next_car(Flag,E,T),
      set_car_queue(Q),
      unregister).

next_car(Flag,Car,T) :=
    ( Flag = empty → true
    ; Flag = partial, Car = car(TL,N) → car([T|TL], N)).
```

Program Output

The following listing was cut from a typical run of the simulation. In this run the arrival of cars was used with a random number between 0 and 9 and the termination of the simulation was used with 10000 unit of time. The input parameters for this run of the simulation were as follows:

- No of Workers: 3
- Service Time: 5

```
?- NoOfWorkers = 3, ServiceTime = 5,
new(Scheduler) # scheduler,
new(Random) # randomize,
new(Out) # out(Scheduler),
new(CarWasher) # car_washer(NoOfWorkers,ServiceTime,Out,Scheduler),
new(CarGenerator) # car_generator(CarWasher,Random,Scheduler),
init ^ CarGenerator.
```

```
car([5,0],1)
car([7,2],2)
car([13,8],3)
car([20,15],4)
car([23,18],5)
car([25,20],6)
car([28,23],7)
car([36,31],8)
car([42,37],9)
car([44,39],10)
car([52,47],11)
car([54,49],12)
car([63,58],13)
car([65,60],14)
car([71,66],15)
car([71,66],16)
car([73,68],17)
car([79,74],18)
car([81,76],19)
car([85,80],20)
car([93,88],21)
car([95,90],22)
car([96,91],23)
car([100,95],24)
car([107,102],25)
car([116,111],26)
car([125,120],27)
car([131,126],28)
car([135,130],29)
car([143,138],30)
```

```
car([144,139],31)
car([145,140],32)
car([151,146],33)
car([154,149],34)
car([154,149],35)
car([162,157],36)
car([163,158],37)
car([168,163],38)
car([177,172],39)
car([178,173],40)
car([184,179],41)
car([187,182],42)
car([192,187],43)
car([192,187],44)
car([195,190],45)
car([197,192,190],46)
car([199,194],47)
car([203,198],48)
...
car([9865,9860],2167)
car([9866,9861],2168)
car([9870,9865,9861],2169)
car([9870,9865,9864],2170)
car([9871,9866,9865],2171)
car([9875,9870,9868],2172)
car([9877,9872],2173)
car([9882,9877],2174)
car([9883,9878],2175)
car([9886,9881],2176)
car([9887,9882],2177)
car([9890,9885],2178)
car([9898,9893],2179)
car([9898,9893],2180)
car([9903,9898],2181)
car([9911,9906],2182)
car([9911,9906],2183)
car([9918,9913],2184)
car([9921,9916],2185)
car([9929,9924],2186)
car([9938,9933],2187)
car([9940,9935],2188)
car([9949,9944],2189)
car([9958,9953],2190)
```

```
car([9967,9962],2191)
car([9972,9967],2192)
car([9981,9976],2193)
car([9986,9981],2194)
car([9993,9988],2195)
car([9999,9994],2196)
```

yes

7.2 A Moving Point Constraints

Constraints are functional relationships between entities of an object model. The term entity includes objects, classes, attributes. A constraint restricts the values that entities can assume. The specification of constraints in AKL+ describes the fact that the constraint must be satisfied in all objects of the class and its subclasses. The following problem is taken from [47]. It will investigate the application of AKL+ to specify objects with constraints.

Problem Description

We would like to represent the object constraints for a class hierarchy. The particular domain, that of a region constraint for a *moving point* class hierarchy, was chosen to illustrate this technique.

For the domain of moving point, a point can move horizontally or vertically without any specific constraints, keeps a record of its history, and obey a region constraints with or without keeping a history of its moves.

The classes

From the above discussion objects with the following incremental behavior can be identified:

1. a point that moves in a two dimensional space.
2. a movable point that records its moves.
3. a movable point that should satisfy a region constraints.
4. a movable point that both records its moves and should satisfy a region constraints.

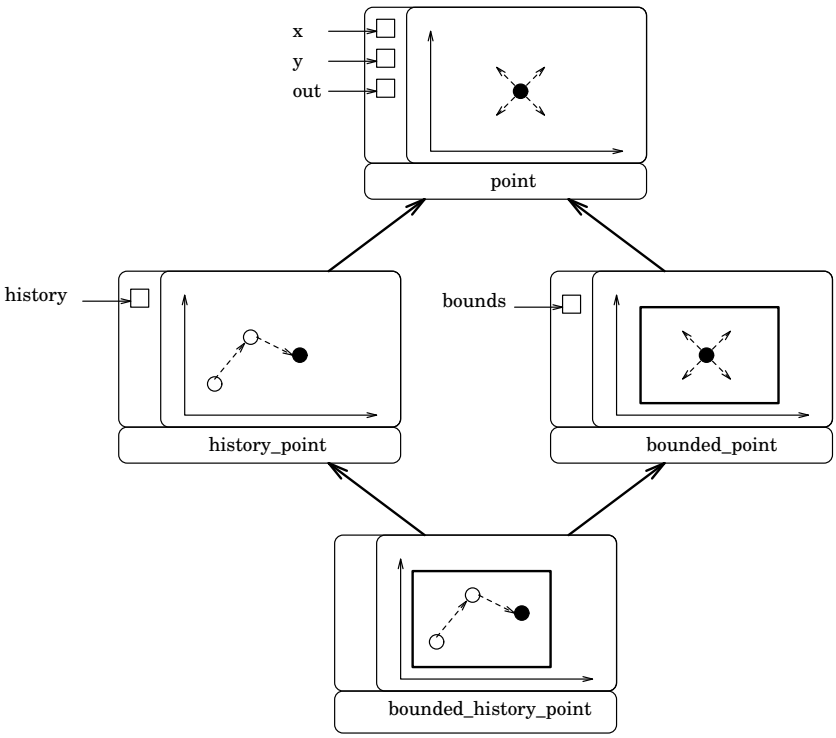

Figure 7.2: The moving point class hierarchy

Fig. 7.2 illustrates the moving point class hierarchy. The following sections summarize the protocol (attributes and methods) for the various classes in the moving point problem.

Point

The class *point* with class *ur_object* as its super defines a movable point in a two dimensional space. Its state contains the attributes: *x* initialized with *0*, *y* initialized with *0*, and *out* initialized with the standard output device. The attributes *x* and *y* together store the position of the point. The *out* attribute will be used to synchronize the display of output. Its behavior consists of the methods:

- *location/2*: returns the current location of the point.
- *move2*: changes the current location of the point.
- *display/0*: prints the current location on the given output device.

```
:- class point.
:- supers [ur_object].
:- attributes [x=(X)\\(X=0),y=(Y)\\(Y=0),out=(S)\\akl.stdout(S)].
```

```
location(X,Y):=
    ( true → get_x(X),
      get_y(Y)).

move(X,Y):=
    ( true → set_x(X),
      set_y(Y)).

display:=
    ( true → location(X,Y),
      get_out(S),
      io.format('Point at: (~w, ~w)~n',[X,Y])–S,
      set_out(S)).
```

History point

The class *history_point* with class *point* as its super defines a point with a changeable location which holds a list contains a record of all locations the point has had since its creation. Its state contains the attribute *h_item* initialized with an empty list. Its behavior consists of the methods:

- *move/2*: changes the current location of the point and updates the history record of point movements.
- *print_history/0*: prints the history record of point movement on the given output device.
- *display/0*: prints the current location and the history record of point movement on the given output device.

```
:- class history_point.
:- supers [point].
:- attributes [h_item=(L)\\(L=[])].

move(X,Y):=
    ( true → move(X,Y) # point,
      get_h_item(L),
      set_h_item([(X,Y)|L])).

print_history:=
    ( true → get_out(S),
      get_h_item(L),
```

```
    io.format('with location history: ~w~n',[L])–S,
    set_out(S)).

display:=
    ( true → display # point,
      print_history # history_point).
```

Note that the computations performed in response to the message *display/0* have been factored so that the message *display/0* is delegated to the superclass *point* such that the overridden method *display/0* is activated and the local processing is performed by activating the local method *print_history/0*. As the two activations are executed concurrently, the order of information displayed may vary. To display the information in a specific order the attribute *out*, inherited from *point*, is used to serialize the output displayed.

Bounded point

The class *bounded_point* with class *point* as its super defines also a point which has, in addition to a location, lower and upper bounds for that location. Its state contains the attribute *bounds* initialized with (0,0,0,0). Its behavior consists of the methods:

- *move/2*: changes the current location of the point if the given value lies within *bounds*.
- *check_constraint/4*: tries to satisfy the region constraints for each activation of *move/2*.
- *print_bounds/0*: prints the upper and lower bounds of the point on the given output device.
- *display/0*: prints the current location of the point and its upper and lower bounds on the given output device.

```
:- class bounded_point.
:- supers [point].
:- attributes [bounds=(B)\\(B=(0,0,0,0))].

move(X,Y):=
    ( true → get_bounds(Bounds),
      check_constraint(Bounds,X,Y,_YesNo)).

check_constraint(Bounds,X,Y,YesNo):=
    ( Bounds = (X0,X1,Y0,Y1),
```

```
      X >= X0,
      X =< X1,
      Y >= Y0,
      Y =< Y1 → move(X,Y) # point,
      YesNo = yes
    ; true → YesNo = no).

print_bounds:=
    ( true → get_out(S),
      get_bounds((X0,X1,Y0,Y1)),
      io.format('with bounds– min: (~w, ~w), max: (~w, ~w)~n', [X0,Y0,X1,Y1])–S,
      set_out(S)).

display:=
    ( true → display # point,
      print_bounds # bounded_point).
```

Bounded history point

The class *bounded_history_point* with classes *history_point* and *bounded_point* as its supers defines also a point which has a location, lower and upper bounds for that location, and a location history record. Its behavior consists of the methods:

- *move/2*: changes the current location of the point if the given value lies within *bounds*.
- *move_aux/3*: if the constraint is satisfied, *move/2* is delegated to *history_point*; otherwise nothing is to be done.
- *display/0*: prints the current location, its upper and lower bounds, and the history record of point movement on the given output device.

```
:- class bounded_history_point.
:- supers [history_point,bounded_point].
:- private [move_aux/3].

move(X,Y):=
    ( true → get_bounds(Bounds),
      check_constraint(Bounds,X,Y,YesNo),
      move_aux(X,Y,YesNo) # bounded_history_point).

move_aux(X,Y,YesNo):=
```

```
    ( YesNo = yes → move(X,Y) # history_point
    ; YesNo = no → true).

display:=
    ( true → display # bounded_point,
      print_history).
```

Program Output

The following output was taken from a typical run of a moving point.

```
?- new(BHPoint) # bounded_history_point,
set_bounds((5,15,5,15))^BHPoint,
move(9,12)^BHPoint,
display^BHPoint,
move(1,2)^BHPoint,
display^BHPoint,
move(8,12)^BHPoint,
display^BHPoint.


Point at: (9, 12)
with bounds– min: (5, 5), max: (15, 15)
with location history: [(9,12)]
Point at: (9, 12)
with bounds– min: (5, 5), max: (15, 15)
with location history: [(9,12)]
Point at: (8, 12)
with bounds– min: (5, 5), max: (15, 15)
with location history: [(8,12),(9,12)]

yes
```

7.3 A Car Maintenance Fault diagnosis

This example investigates the application of AKL+ to the diagnosis of faults. It is taken from LPA's Prolog++ manual [146]. It is intended to illustrate the technique of progressively sending messages through a hierarchy of objects, starting from objects at a general level down to objects at specific levels. It also shows that

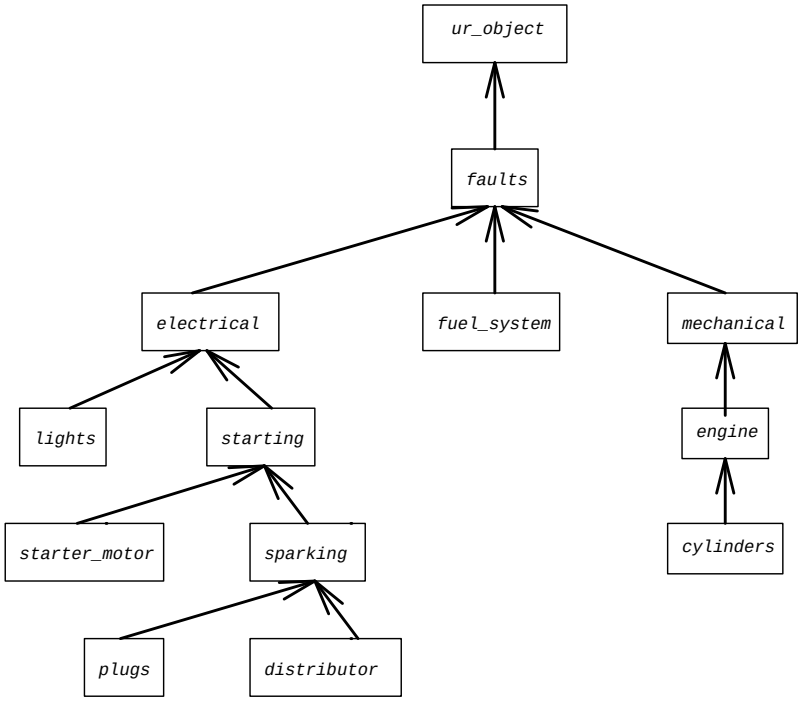

Figure 7.3: A car component hierarchy

it is easily to convert Prolog based object-oriented applications into AKL+. The particular domain, that of *car maintenance*, was chosen to illustrate this technique.

Problem Description

We would like to represent the causal-effect relationship between faults and symptoms. The crucial factor in fault diagnosis is being able to quickly pinpoint the general area of the problem, before focusing in on the root case. Identifying general aspects of the problem can avoid going down blind alleys, and more importantly avoids asking the user seemingly irrelevant questions.

For the domain of car maintenance, an automobile can be dissected into several problem areas such as the fuel system, mechanical faults and electrical faults. Associated with each area is a collection of faults which may occur, and the symptoms which they cause. Some of the symptoms may be contradictory, in the sense that two symptoms cannot possibly occur simultaneously.

The classes

From the above discussion two quite separate classes can be identified. The first concerns itself with fault diagnosis, including the ability to move from general objects down to specific objects. The second deals with the problem domain, in terms of the

causal-effect relationships between actual faults and exhibited symptoms. Fig. 7.3 illustrates a hierarchy for identifying faults in automobiles. At the top most level is the *faults* class containing the algorithm for finding faults, and which is inherited by all of the other classes in the hierarchy. At any point, however, a class has the option to override the default search algorithm with one that is specialized for its problem area.

An instance of the component hierarchy is created with a double link that represents the super-sub relationship. Thus, allowing to apply different search algorithms.

The following sections define the protocol (attributes and methods) for the *fault* class and for the domain specific classes.

Faults

The class *faults*, a subclass of *ur_object*, contains the search algorithm for identifying and reporting faults. The algorithm, since the class is at the top of the hierarchy, is inherited by all of the domain classes. Its state contains the attributes: *super* to specify its parents objects in the object hierarchy, and *sub* to specify its children objects, and *db* to specify the class specific information of the problem domain. The attributes *super* and *sub* is initialized with the empty list. The attribute *db* is initialized with a *hash table* object for fast access. Entries of this object will contain the following information:

- *faults*: takes on a list of unique fault numbers.
- *fault(Number)*: takes on a text of the fault for a unique fault *Number*.
- *symptoms*: takes on a list of unique symptom numbers.
- *symptom(Number)*: takes on a text of the symptom for a unique symptom *Number*.
- *contrary(Symptom)*: tabulates when one symptom is contrary to another, i.e. they cannot coexist as symptoms.
- *effect(Effect)*: tabulates when a fault causes an *Effect* which is exhibited as a symptom.

The behavior of *faults* consists of the methods:

- *new/3*: creates a newly initialized object. Given an object as super, updates the attribute *super* of the new object to refer to its super object and updates the attribute *sub* of the super object to refer to the new object.

- *set_db/2*: updates the value at the given key of the hash table object.
- *get_db/2*: returns the value at the given key of the hash table object.
- *init_db/2*: initializes the hash table with default values.
- *findall/0*: finds and reports all faults.
- *process_subs/1*: progressively moves through the object hierarchy from general problem areas down to specific problem areas.
- *find/[0-3]*: for each defined fault for this object, gets the effect, a list of symptoms, of this fault, and go through this list to verify its elements.
- *contrary_symptom/3*: accumulates the contrary symptoms of a particular fault.
- *check_contrary/5*: checks whether there are a contrary symptoms already exhibited. If this is the case, the previous reply is considered. Otherwise, asks about the contrary symptom and remembers the reply.
- *check_my_symptom/5*, *check_my_symptom_aux/5*: if the symptom was not previously asked nor the contrary symptom was found the symptoms will be checked to determine whether it is already exhibited. If this is the case, the previous reply is considered. otherwise, asks about the symptom and remembers the reply.
- *print/2*: prints the fault and the location (a class in the domain) where it was identified.
- *exhibited/[6-7]*: asks the user whether a particular symptom occurs and remembers the answers to questions in order to avoid them being repeatedly asked.
- *writeseqnl/4,write_seq/3*: displays questions on the standard output device.
- *member_told/3*: checks whether or not the question was previously asked. If so, the answer is returned.

```
:- class faults.
:- supers [ur_object].
:- attributes [super=(V)\\(V=[]),sub=(V)\\(V=[]),
               db=(DB)\\(akl.new_hash_table(HashDB),init_db(HashDB,DB))].

new(Obj)–Super:=
    ( true → new(Obj) # ur_object,
      get_super(A)^Obj,
      set_super([Super|A])^Obj,
```

```
      get_sub(B)^Super,
      set_sub([Obj|B])^Super).

set_db(A,V):=
    ( true → get_db(DB),
      set(A,V)^DB,
      set_db(DB)).

get_db(A,V):=
    ( true → get_db(DB),
      get(A,V)^DB).

init_db–DB:=
    ( true → set(faults,[])^DB,
      set(fault(nil),[])^DB,
      set(symptoms,[])^DB,
      set(symptom(nil),[])^DB,
      set(contrary(nil),[])^DB,
      set(effect(nil),[])^DB).

findall :=
    ( true → get_sub(Subs),
      process_subs(Subs)).

process_subs(Subs):=
    ( Subs = [Sub|Rest] → find^Sub,
      findall^Sub,
      process_subs(Rest)
    ; Subs = [] → true).

find :=
    ( true → get_db(faults,Faults),
      find(Faults,[],_R)).

find(Faults)–R :=
    ( Faults = [] → true
    ; Faults = [FaultNum|Fs] → get_db(effect(FaultNum),SL),
      contrary_symptom(SL,CONSL,[]),
      check_contrary_sym(CONSL,no,Abort)–R,
      check_my_sym(SL,Abort,Continue)–R,
      print(FaultNum,Continue),
      find(Fs)–R).
```

```
contrary_symptom(SL)–L :=
    ( SL = [] → true
    ; SL = [S|Ss] → get_db(contrary(S),S1),
      [S1]–L,
      contrary_symptom(Ss)–L
    ; SL = [_S|Ss] → contrary_symptom(Ss)–L).

check_contrary_sym(CONSL)–Abort–R :=
    ( CONSL = [] → true
    ; CONSL = [_S1|_Ss], Abort = yes → true
    ; CONSL = [nil|Ss] → check_contrary_sym(Ss)–Abort–R
    ; CONSL = [S1|Ss] → get_db(symptom(S1),Text),
      exhibited(S1,Text)–Abort–R ,
      check_contrary_sym(Ss)–Abort–R).

check_my_sym(SL,Abort,Continue,R0,RL) :=
    ( Abort = yes → R0 =RL,
      Continue = no
    ; true → check_my_sym_aux(SL,yes,Continue,R0,RL)).

check_my_sym_aux(SL)–Continue–R :=
    ( SL = [] → true
    ; SL = [_S1|Ss], Continue = no → true
    ; SL = [S1|Ss] → get_db(symptom(S1),Text),
      exhibited(S1,Text)–Continue–R,
      check_my_sym_aux(Ss)–Continue–R).

print(FaultNum,Continue):=
    ( Containue = yes → typeof(Where),
      get_db(fault(FaultNum),Fault),
      akl.stdout(S),
      writeseqnl('Location : ', [Where])–S ,
      writeseqnl('Possible Fault : ',[Fault])–S ,nl^S
    ; Continue = no → true).

exhibited(S,Text,_Flag,Reply)–R :=
    ( true → member_told(told(S,Ans),R,YesNo),
      exhibited(YesNo,S,Ans,Text,Reply)–R).

exhibited(YesNo,S,Ans,Text,Reply,R0,R) :=
    ( YesNo = yes → Reply = Ans, R0 = R
    ; YesNo = no → yesno(Text,Reply),
      Reply = Ans,
```

```
      R = [told(S,Ans)|R0]).

yesno(Text,R):=
    ( akl.stdout(S),
      akl.stdin(In),
      akl.data(Text),
      io.write(Text)–S,nl^S,
      io.read(yes)–In → R = yes
    ; true → R = no).

writeseqnl(Prompt,L)–S:=
    ( true → io.write(Prompt)–S,
      write_seq(L)–S).

write_seq(L)–S:=
    ( L=[] → true
    ; L =[H|T] → io.write(H)–S, io.write(' ')–S,
      write_seq(T)–S , nl^S).

member_told(Told,L, YesNo):=
    ( L=[] → YesNo = no
    ; L=[told(X,Y1)|_],
      Told = told(X,Y) → Y = Y1, YesNo = yes
    ; L=[_|Xs] → member_told(Told,Xs,YesNo)).
```

Domain Classes

As an illustration of one of the classes in the car maintenance system, the class distributor is given below.

Distributor

The class *distributor* with class *sparking* as its super is defined. Its behavior consists of the specialized method *new/3* which sets the domain specific knowledge for its newly created object.

```
:- class distributor.
:- supers [sparking].

new(Obj)–Super:=
    ( true → new(Distributor)–Super # sparking,
```

```
set_db(faults,['F1001','F1002','F1003'])^Distributor,
set_db(fault('F1001'), 'Condensation in distributor cap')^Distributor,
set_db(fault('F1002'), 'Faulty distributor arm')^Distributor,
set_db(fault('F1003'), 'Worn distributor brushes')^Distributor,
set_db(symptoms,['S1001','S1002','S1003','S1004'])^Distributor,
set_db(symptom('S1001'), 'Starter turns, but engine does not fire')^Distributor,
set_db(symptom('S1002'), 'Engine has difficulty starting')^Distributor,
set_db(symptom('S1003'), 'Engine cuts out shortly after starting')^Distributor,
set_db(symptom('S1004'), 'Engine cuts out at speed')^Distributor,
set_db(contrary('S1002'),'S1001')^Distributor,
set_db(contrary('S1003'),'S1001')^Distributor,
set_db(contrary('S1001'), nil)^Distributor,
set_db(contrary('S1004'), nil)^Distributor,
set_db(effect('F1001'), ['S1001'])^Distributor,
set_db(effect('F1002'), ['S1001','S1004'])^Distributor,
set_db(effect('F1003'), ['S1002','S1003'])^Distributor,
Obj = Distributor).
```

Program Output

The following outputs were taken from two fault diagnosis runs. In these runs, an instance of the car maintenance class hierarchy is created with an object of *distributer* set to the above domain specific knowledge and other domain classes with their default values.

```
?- new(Faults) # faults,
new(Electrical)–Faults # electrical,
new(Fuel_system)–Faults # fuel_system,
new(Mechanical)–Faults # mechanical,
new(Lights)–Electrical # lights,
new(Starting)–Electrical # starting,
new(Sparking)–Starting # sparking,
new(Starter_motor)–Starting # starter_motor,
new(Plugs)–Sparking # plugs,
new(Distributor)–Sparking # distributor,
new(Engine)–Mechanical # engine,
new(Cylinders)–Engine # cylinders,
findall^Faults.
```

Starter turns, but engine does not fire
yes.
Location : distributor
Possible Fault : Condensation in distributor cap

Engine cuts out at speed
yes.
Location : distributor
Possible Fault : Faulty distributor arm

yes

?- new(Faults) # faults,
new(Electrical)–Faults # electrical,
new(Fuel_system)–Faults # fuel_system,
new(Mechanical)–Faults # mechanical,
new(Lights)–Electrical # lights,
new(Starting)–Electrical # starting,
new(Sparking)–Starting # sparking,
new(Starter_motor)–Starting # starter_motor,
new(Plugs)–Sparking # plugs,
new(Distributor)–Sparking # distributor,
new(Engine)–Mechanical # engine,
new(Cylinders)–Engine # cylinders,
findall^Faults.

Starter turns, but engine does not fire
no.
Engine has difficulty starting
yes.
Engine cuts out shortly after starting
yes.
Location : distributor
Possible Fault : Worn distributor brushes

yes

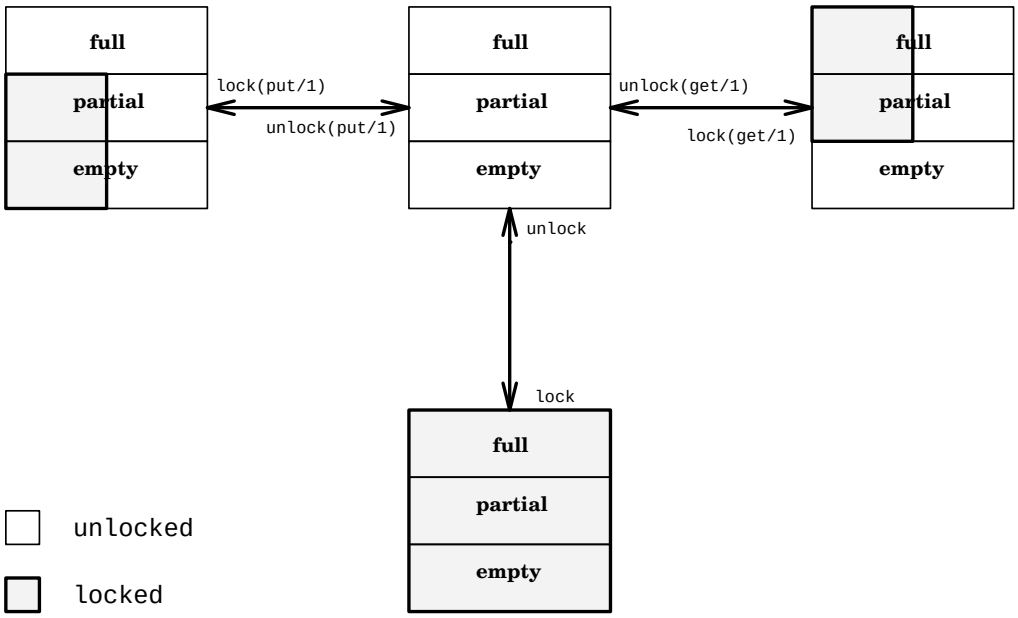

Figure 7.4: Conceptual illustration of the state modification of a bounded buffer object

7.4 A Bounded Buffer State Modification

This example investigates the application of AKL+ to the modification of acceptable states of a concurrent object. It is intended to illustrate the dynamic locking of a concurrent object operations which modifies these states. The particular domain, that of *bounded buffer*, was chosen to illustrate this technique. The bounded buffer is a classic example that is used in the literatures to illustrate the concurrency and synchronization of a concurrent object operations.

Problem Description

We would like to represent the mixing of lock operations with a bounded buffer operations such that the state modification anomaly, discussed in Section 4.5.3, is avoided. The example is used in [98, 99] to illustrate this type of anomaly. The problem is that in order to account for the locking of operations in a subclass of the bounded buffer, the bounded buffer operations is affected such that methods of the bounded buffer must be overridden in order to account for locked.

The classes

From the above discussion two quiet separate classes can be identified. The first concerns itself with the locking of operations. The second deals with the problem domain, a buffer that can lock its operations. The class *write_lock* defines a two-level

lock so that the whole object operations can be exclusively locked until they unlocked. The locking of operations is extended in its subclass *write_lock_op* to allow arbitrary locking of the bounded buffer operations. A bounded buffer *buffer_trans* class with operations to store and retrieve elements is identified; then the synchronization is that one cannot retrieve from a buffer whose state is *empty* and cannot store into a buffer whose state *full* is likewise prohibited. The satisfaction of these constraints is not achieved automatically; the user must somehow program the methods to implement the object behavior that satisfy the synchronization constraints. The AKL+ *transitions* approach is chosen to specify these constraints. The class *write_lock_op_buffer* is identified with the locking operations mixed from *write_lock_op*. Fig. 7.4 illustrates the possible state modification of an object of *write_lock_op_buffer* according to the locking of its operations. At any point in time, it is possible to send a message that locks or unlocks the bounded buffer operations.

According to the *transitions* approach, the application is programmed with methods: *mset/2* to specify the accept method set and their identifiers, *transition/3* to specifying the transitional behavior of the accept method set, and *enable/[3-4]* to determine the current accept method set.

Write lock

The class *write_lock* defines the methods: *lock/0* and *unlock/0*. *lock/0* locks the object exclusively so that no other methods can access it until it receives a corresponding *unlock/0* message. Its state contains the attributes: *lock_var* to indicate whether the current object is locked or not and *method_set* to store/restore the accept method set when the message *lock/0*/ *unclock/0* is received. The definition of the class *write_lock* is given in Example 4.11.

Write lock operation

The class *write_lock_op* defines the methods: *lock/1* and *unlock/0*. *lock/1* locks a given arbitrary set of object operations. *unlock/1* unlocks a given arbitrary set of object operations. Its state contains the attribute *lock_op_var* to store the current locked operations.

```
:- class write_lock_op.
:- supers [write_lock].
:- attributes [lock_op_var=(V)\\(V=[])].

lock(Op):=
    ( Op = [_|_] → get_lock_op_var(L),
      set_lock_op_var(L1),
```

```
      sets.set_union(Op,L,L1),
      io.format('-- lock(~w) -- ~n',[Op])
    ; true → get_lock_op_var(L),
      set_lock_op_var(L1),
      sets.set_union([Op],L,L1),
      io.format('-- lock(~w) -- ~n',[Op])).

unlock(Op):=
    ( Op = [_|_] → get_lock_op_var(L),
      set_lock_op_var(L1),
      sets.set_difference(L,Op,L1),
      io.format('-- unlock(~w) -- ~n',[Op])
    ; true → get_lock_op_var(L),
      set_lock_op_var(L1),
      sets.set_difference(L,[Op],L1),
      io.format('-- unlock(~w) -- ~n',[Op])).
```

Bounded Buffer

The class *buffer_trans* with the standard AKL+ classes *transitions* and *state_hash* as its superclasses defines a bounded buffer operations. Its state contains the attributes: *in, out,* and *array.* The attribute *buffer* is initialized with an array object for fast access. The attributes *in* and *out* is initialized with *0*, they are used as indices into the buffer. Its behavior contains the methods: *put/1* for storing an item in the buffer and *get/1* for retrieving an item from the buffer. The class is parameterized by its maximum size, allowing for creating buffers with variable length at run-time.

```
:- class buffer_trans(MaxSize).
:- supers [transitions,state_hash].
:- attributes [in=(V)\\(V=0),out=(V)\\(V=0),size=(V)\\(V=0),
               buffer=Array\\(akl.new_array(MaxSize,0,Array),
               write_array.write_array-Array)].

mset(SetId,Mset):=
    ( SetId = initially → mset(empty,Mset) # buffer_trans(MaxSize)
    ; SetId = empty → Mset= [put/1]
    ; SetId = full → Mset = [get/1]
    ; SetId = partial → mset(empty,Mset1) # buffer_trans(MaxSize),
      mset(full,Mset2) # buffer_trans(MaxSize),
      sets.set_union(Mset1, Mset2, Mset)).

transition(M,MethodSet,Enables):=
```

```
    ( M = default → get_size(Size),
      enable(Size,MaxSize,MethodSet,Enables) # buffer_trans(MaxSize)
    ; M = get/1 → transition(default,MethodSet,Enables)
    ; M = put/1 → transition(default,MethodSet,Enables)).

enable(Size,Max,_MethodSet,Enables):=
    ( Size > 0,Size < Max → Enables = partial
    ; Size = 0 → Enables = empty
    ; Size = Max → Enables = full).

put(Item) :=
    ( true → akl.stdout(S),
      get_in(In),
      get_size(Size),
      get_buffer(Array),
      In1 is (In+1) mod MaxSize,
      set(In,Item)^Array,
      Size1 is Size +1,
      io.write('—– put(')–S,
      io.write(Item)–S,
      io.write(')—–')–S,nl^S,
      write_array.write_array–Array,
      set_in(In1),
      set_size(Size1)).

get(Item) :=
    ( true → akl.stdout(S),
      get_out(Out),
      get_size(Size),
      get_buffer(Array),
      Out1 is (Out+1) mod MaxSize,
      get(Out,Item)^Array,
      Size1 is Size-1,
      set(Out,[])^Array,
      io.write('—– get(')–S,
      io.write(Item)–S,
      io.write(')—–')–S,nl^S,
      write_array.write_array–Array,
      set_out(Out1),
      set_size(Size1)).
```

Bounded Buffer with locks

The class *write_lock_op_buffer* with classes *buffer_trans* and *write_lock_op* as its super defines a bounded buffer which can lock its operations. To make reuse of code, the *buffer_trans* is defined as follows.

```
:- class write_lock_op_buffer(MaxSize).
:- supers [buffer_trans(MaxSize),write_lock_op].

mset(SetId,Mset) :=
    ( SetId = initially → mset(initially, Mset1) # buffer_trans(MaxSize),
      mset(always, Mset2) # write_lock_buffer(MaxSize),
      sets.set_union(Mset1,Mset2, Mset)
    ; SetId = locked → mset(locked,Mset) # write_lock_op
    ; SetId = unlocked → mset(unlocked,Mset) # write_lock_op
    ; SetId = always → Mset = [lock/1,unlock/1,lock/0]
    ; true → mset(SetId,Mset1) # buffer_trans(MaxSize),
      mset(always, Mset2),
      sets.set_union(Mset1, Mset2, Mset3),
      get_lock_op_var(Op),
      sets.set_difference(Mset3,Op,Mset)).

transition(M,MethodSet,Enables):=
    ( M = lock/0 → transition(lock/0,MethodSet,Enables) # write_lock
    ; M = unlock/0 → transition(M,MethodSet,Enables) # write_lock
    ; true → transition(default,MethodSet,Enables) # buffer_trans(MaxSize)).
```

Note that methods *put/1* and *get/1* is reused and the synchronization code specifies the locking and unlocking of the bounded buffer operations, (cf. [98, 99]).

Program Output

The following outputs were taken from three runs.

```
?- new(Obj) # write_lock_op_buffer(3),
get(_)^Obj,
put(a)^Obj,
lock(get/1)^Obj,
put(b)^Obj,
lock(put/1)^Obj,
get(_)^Obj,
put(c)^Obj,
```

```
unlock(get/1)^Obj,
put(d)^Obj,
unlock(put/1)^Obj,
put(e)^Obj.


[] [] []
—– put(a)—–
a [] []
—– get(a)—–
[] [] []
—– lock(get/1) ——
—– put(b)—–
[] b []
—– lock(put/1) ——
—– unlock(get/1) ——
—– get(b)—–
[] [] []
—– unlock(put/1) ——
—– put(c)—–
[] [] c
—– put(d)—–
d [] c
—– put(e)—–
d e c
** Object Terminated successfully **
```

yes

```
?- new(Obj) # write_lock_op_buffer(3),
get(_)^Obj,
put(a)^Obj,
lock([get/1,put/1])^Obj,
put(b)^Obj,
get(_)^Obj,
put(c)^Obj,
unlock(put/1)^Obj,
put(d)^Obj,
unlock(get/1)^Obj,
put(e)^Obj.
```

```
[] [] []
—– put(a)—–
a [] []
—– get(a)—–
[] [] []
—– lock([get/1,put/1]) ——
—– unlock(put/1) ——
—– put(b)—–
[] b []
—– put(c)—–
[] b c
—– put(d)—–
d b c
—– unlock(get/1) ——
—– get(b)—–
d [] c
—– put(e)—–
d e c
** Object Terminated successfully **
```

yes

```
?- new(Obj) # write_lock_op_buffer(3),
get(_)^Obj,
put(a)^Obj,
lock([get/1,put/1])^Obj,
put(b)^Obj,
get(_)^Obj,
put(c)^Obj,
unlock(put/1)^Obj,
put(d)^Obj,
unlock(get/1)^Obj,
put(e)^Obj,
get(_)^Obj,
put(f)^Obj,
lock^Obj,
put(g)^Obj,
put(h)^Obj,
unlock^Obj,
get(_)^Obj,
get(_)^Obj.
```

```
[] [] []
—– put(a)—–
a [] []
—– get(a)—–
[] [] []
—– lock([get/1,put/1]) ——
—– unlock(put/1) ——
—– put(b)—–
[] b []
—– put(c)—–
[] b c
—– put(d)—–
d b c
—– unlock(get/1) ——
—– get(b)—–
d [] c
—– put(e)—–
d e c
—– get(c)—–
d e []
—– put(f)—–
d e f
—– lock ——
—- unlock —–
—– get(d)—–
[] e f
—– put(g)—–
g e f
—– get(e)—–
g [] f
—– put(h)—–
g h f
** Object Terminated successfully **
```

yes

7.5 Summary

We have described programming examples that cover a range of problems, each of which illustrates different aspects of the language. They are:

- a discrete-event simulation of the activities of a car washer.
- a region constraints for a moving point in a two dimensional space.
- a fault diagnosis for an automobile according to exhibited symptoms and counter symptoms.
- a state modification for a bounded buffer with respect to dynamic locking of its operations.

These problems, which are small but not trivial, are general enough for the purpose of showing the applicability of AKL+ both as a modeling language and as a programming language. In particular, simulation is the most general concurrent computing problem in the sense that any concurrent problem can be formulated as a simulation. The bounded buffer example demonstrates that the AKL+ standard protocols can successfully solve the state modification anomaly as specified in the literatures.

* * ★ * *

Conclusion and Future Work

8.1 Conclusion

This thesis has been concentrated on issues in the design and implementation of two essential parts of a software development environment, namely *incremental compilation facility* and a new concurrent object-oriented programming language called *AKL+*. The language was built on top of AKL supporting all features that are needed for any object-oriented application. The incremental compilation facility is a general tool providing flexible support to automatic incremental compilation and loading of object-oriented programs which is essential for large object-oriented applications.

Chapter 2 has given a survey to a wide range of object-oriented programming languages. We have classified these languages as conventional languages, actor based languages, and logic based languages. The logic based languages are of our main interest. In particular, those languages which are either logic based languages extended with object-oriented constructs or higher object-oriented languages built on top of logic based languages. These languages have been discussed by examining the design and implementation issues concerning classes, objects, state, and inheritance. These issues have been examined because of their central roles in object-oriented programming. At the end of Chapter 2, we have presented the main features of our language and compared it with the most related ones. We have showed that AKL+ supports a more mature features which subsume features of these languages.

Chapter 3 has given a brief and informal summary of AKL and reviewed the basic techniques that allows us to do object-oriented programming in AKL. AKL was chosen as concurrent constraint logic programming element of our language. The reasons for this include its use of deep guards, its use of don't know nondeterministic capabilities of Prolog and the constraint logic programming languages with the process-describing capabilities of concurrent logic languages such as GHC, and the simplicity and flexibility in its support of multiple programming paradigms, such as concurrent, object, functional, logic, and constraint programming. In addition, AKL offers a large potential for automatic parallel execution. From object-oriented view AKL greatly facilitates communication between objects through ports. AKL is a programming language *kernel* which has enabled us to design and efficiently implement a complete language on top of it.

Chapter 4 has informally presented AKL+ and its computation model. AKL+ is a concurrent object-oriented language based on the concepts of classes, generic classes,

metaclasses, multiple inheritance, delegation and abstractions of classes and methods. Several simple examples have been used to illustrate the main features of the language and various programming techniques such as higher-order and data-driven programming techniques. Classes can be defined with attributes, methods, access control of methods, and superclasses. Classes, methods and instances of classes can be expressed as first-class values. Method definitions can be called in two ways: "method delegation" and "method invocation". The method delegation preserves the base class reference. The target object is available under the special attribute "self". Objects can be allocated or destroyed dynamically. Objects can share a common object. AKL+ has achieved a uniform message sending. A set of built-in standard classes has been provided to supply programmers with the minimum set of efficient and effective built-in standard classes (library).

The language has supported the basic synchronization schemes that achieved the concurrency control for a concurrent object. AKL+ has provided two standard protocols, "transitions" and "synchronizers", to support synchronization schemes for resolving the inheritance anomaly. The main advantage to the synchronization constraints schemes in AKL+ is the clean separation of concurrency control and the method specification such that they can be inherited, overridden, or extended separately without affecting each other. Furthermore, one scheme can be integrated and composed with other schemes.

AKL+ has supported multiple inheritance and modeled the inheritance graph directly. The designer is able to resolve any name conflicts due to multiple inheritance in different ways. One way is to redefine the operation in the subclass. An alternative way is to use the differential inheritance feature of the language to exclude the conflicting operation.

Chapter 5 has presented the syntax and semantics of AKL+. The semantics of AKL+ has been described in terms of translation to AKL. The translation has been based on a source-to-source transformations. Each kind of transformation has been defined by a rewrite rule.

Chapter 6 has presented the implementation aspects of the general incremental compilation facility and our language. The AKL+ run-time module that allows us to achieve a run-time uniform message sending has been also described. We have given an algorithm for a general incremental compilation facility. The algorithm specifies dependencies as functions to bring a target file up to date with respect to those files on which it depends. This incremental compilation facility should be applicable and advantageous to all object-oriented software systems. A primarily feature of the incremental compilation facility is that the work needed after a change is proportional to the "size" of the change rather than to the size of the program. We have given an algorithm for the dependency that computes multiple inheritance. We have described the dependency for translating a class definition into AKL. We

have briefly mention the dependency provided by AKL to translate the AKL code into the AKL abstract machine.

Efficiency is one of our prime goals. This has been realized through the schemes developed for the optimization of the code to which a class is expanded, and the efficiency offered by the set of standard classes.

AKL+ is a compiled language that translates classes into AKL code. Efficiency of the translated code has been realized through:(1) class representation as a special light-weight AKL module, (2) the method dispatcher exploits the first argument indexing of the AKL compiler, leading to direct access to the method clauses, (3) the unfolding of class parameters is only performed for the method clause that uses or passes any of these parameters, (4) preserving the last call optimization in recursive methods, (5) enforce the override mechanism by applying the conditional choice primitive to the class dispatcher, and (6) a method invocation during the execution of a message to an object may be directly applied to the state without the need to schedule this message to the target object.

Inheritance has been implemented by the incremental definition of the method dispatcher in such a way that the interface of a class is cleanly captured. The class dispatcher is computed at compile time and this will result in dispatching methods in a constant time rather than searching the class hierarchy each time a method is invoked. This run-time overhead is unpredictable and dependent on the depth of the inheritance path.

The standard class *object* has defined port based objects which are active (heavy weight) objects communicated through ports. The standard class *cell* has defined data objects which are very fine-grained (light weight) objects that provides a minimal form of encapsulated state. The standard class *state_hash* has provided a direct attribute (key) access to the attribute's value. The standard class *state_array* has provided a direct attribute (indexed) access to the attribute's value.

The AKL+ language and our incremental compilation facility have been implemented on Unix-based workstations and they are parts of the official release of the AKL system developed at SICS (Swedish Institute of Computer Science).

Chapter 7 has presented small but complete AKL+ examples that cover a range of problems in object-oriented concurrent computing, from its conception to realization. These problems has showed the applicability of AKL+ both as a modeling language and as a programming language. They are: a discrete-event simulation of the activities of a car washer, a region constraints for a moving point in a two dimensional space, a fault diagnosis for an automobile according to exhibited symptoms and counter symptoms, and a state modification for a bounded buffer with respect to dynamic locking of its operations.

Appendix A has described the commands of the interactive incremental compilation facility.

Finally, the thesis **contributed** with the first concurrent object-oriented, logic-based, system that has supported all needed features for any object-oriented application, a number of implementation techniques for efficient implementation, and the general incremental compilation facility which is essential for developing large object-oriented application.

8.2 Future Work

We have several ongoing activities, all concerned with extending our thesis work to be more powerful and applicable. In what follows we present some of these activities.

As has already mentioned, AKL+ definitions are translated to AKL definitions. This expansion is usually transparent to the user. While debugging an AKL+ program, however, the expanded representation may become exposed. Debugging an AKL+ in terms of the AKL code generated by the compiler is much more complicated and unfriendly than debugging at a level which is more like AKL+ itself. It is far more preferable to debug AKL+ queries in terms of AKL+ programs rather than in terms of AKL programs. Unfortunately, since we compile AKL+ programs into AKL, any AKL+ debugger would have to be able to "de-compile" the AKL code into the AKL+ original. This is difficult to do for all cases; especially since our aim in building the compiler is to produce fast AKL programs rather than easily debugable programs. We may borrow some of the techniques used in the debugging of languages like "C" and compile in debugging information to the AKL+ program as we convert it into an AKL program.

Large software systems comprise many thousands of lines of source code. Sequential reading and writing is inadequate for mastering this complexity. In particular, the structure of object-oriented software systems makes it necessary to inspect many locations in order to get the picture of the class hierarchy, of overridden methods, and so on. Presenting structural information about complex object-oriented software systems requires obtaining this information from different source files. In fact, the environment was designed with these considerations in mind. As was mentioned before, a separate file will store the class information and its inherited information. A browsing tool that takes the advantage of this file is considered in the next release.

A promising avenue of research is being pursued in the logic programming context is to provide the language with partial evaluator to remove the cost of some extra code. A partial evaluator takes a partially specified query and program and evaluates only some parts of the program as not everything is fully specified. The result is a new program specialized to handle all instances of the original query. The intention is

that the generated program will execute more efficiently than the original one for those instances. We need to investigate the design and implementation of a partial evaluator for AKL+.

Research on object-oriented reflective concurrent computation has just started and most research issues have not been investigated yet. A reflective computational system is a computational system which exhibits reflective behavior. In a conventional system, computation is performed on data that represent (or model) entities which are external to the computational system. In contrast, a reflective computational system must contain some data that represent or model the structural and computational aspects of the system itself [150]. And such data must be maintainable within the system itself, and more importantly, changes made to such data must be causally reflected/connected to the actual computation being performed. One of our future goals is to investigate the design of a reflective AKL+.

AKL+ supports all features that are needed for any object-oriented application. Although, there is no experience yet in using AKL+ for a serious application. This experience is necessary to evaluate the system and to discover its strengths and weakness.

One of the most important future design goals for AKL+ is support for distributed programming. object-oriented programming is a very suitable abstraction for distributed programming, since it naturally contains computationally independent entries which communicate by message passing. However, extensions are required for dealing with such things as the explicit mapping of objects to processors, transferring objects from one cite to achieve load balancing, and recovery from failure.

* * ★ * *

A

Interactive Incremental Compilation Commands

We give a summary of the available commands of the incremental compilation facility discussed in Section 6.1.

- *use(Root) :* Invokes the incremental compilation facility.
- *use :* (Re)invokes the incremental compilation facility.
- *clean(Root) :* Removes all the dependent/reachable files of Root.
- *clean :* Removes all the dependent/reachable files of the current Root.
- *automake(Status) :*

  If Status is "on", the incremental compilation facility is activated such that system makes sure that all changed files are recompiled and loaded before each new top-level query.

  If Status is "off", only *use* command can activate the incremental compilation facility.
- *automake :* Displays the automake status.
- *used :* Displays the used files.
- *edit(File) :* Invokes Emacs editor for the given file.
- *editb(File) :* Invokes Emacs editor for the given file as a background process.
- *halt :* Terminates the session.
- *h :* Displays the history list.
- *hist. No. :* Executes a history command.
- *help :* Displays this list.

⋆ ⋆ ★ ⋆ ⋆

References

[1] Agha, G., Actors: A model of Concurrent Computation in Distributed Systems, *MIT Press*, Cambridge, 1986.

[2] Agha, G., Hewitt C., Concurrent Programming Using Actors, in Yonezawa A., Tokoro M. (eds.), Object-Oriented Concurrent Programming, *MIT Press*, Cambridge, 1987.

[3] Agha G., Foundational Issues in Concurrent Computing, ACM SIGPLAN Workshop on Object-Based Concurrent Programming, Proceeding, *SIGPLAN Notices*, 24(4):60–65, 1988.

[4] Ait-kaci H., Nasr R., Login: A Logic Programming Language with built-in Inheritance, *Journal of Logic Programming*, 3(3):185–215, 1986.

[5] Ait-kaci H., Towards a meaning of LIFE, *Journal of Logic Programming*, 16(3):195–234, 1991.

[6] Alexiev V., Mutable Object State for Object-Oriented Logic Programming: A Survey, *Technical Report*, TR 93–15, Department of Computing Science, University of Alberta, 1993.

[7] Almgren J., Andersson S., Carlsson M., Flood L., Haridi S., Frisk C., Nilsson H., Sundberg J., SICStus Prolog Library Manual, *SICS Technical Report*, T93:02A, Swedish Institute of Computer Science, 1993.

[8] America P., Inheritance and Subtyping in a Parallel Object-Oriented Language, *ECOOP*, Proceeding, 1987.

[9] America P., POOL–T: A Parallel Object-Oriented Language, in Yonezawa, A., Tokoro M (eds.), Object-Oriented Concurrent Programming, Cambridge, *MIT Press*, 1987.

[10] Andersen B., Ellie Language Definition Report, *ACM SIGPLAN Notices*, 25(11):45–64, 1990.

[11] Andersen, B., Ellie: a General, Fine-grained, First-class, Object-based Languages. *Journal of Object-Oriented Programming*, 5(2):35–42, May, 1992.

[12] Andreoli J., Parechi R., LO and Behold! Concurrent Structured Processes, *ECOOP/OOPSLA*, Proceeding, 1990.

[13] Andreoli J., Parechi R., Linear Objects: Logical processes with Built-in Inheritance, *OOPSLA*, Proceeding, 1991.

[14] Annot J., Haan P., POOL and DOOM: The Object-Oriented approach, in Treleaven P. (ed.), Parallel Computers: Object-Oriented, Functional, Logic, *Jonh Wiely & Sons*, NY, 1990.

[15] APaepcke, A. (ed.), Object-Oriented Programming: The CLOS perspective. *MIT Press*, 1993.

[16] Athas W., Boden N., Cantor: An Actor Programming System for Scientific Computing, ACM SIGPLAN Workshop on Object-Based Concurrent Programming, Proceeding, *SIGPLAN Notices*, 24(4):66–68, 1988.

[17] Barnes J., Programming in Ada, *Addison-Wesley Publishing Co.*, Mass, 1989.

[18] Bartual R., LPA Prolog and Flex Expert, *Program Now*, 3(2):43-47, 1989.

[19] Bielak, R. Object-Oriented Programming: the Fundamentals, *ACM SIGPLAN Notices*, 28(9):13–14, Sept., 1993.

[20] Bobrow, D., CommonLoops: Merging Lisp and Object-Oriented Paradigm, *ACM Conference on Object-Oriented Systems, Languages, and Applications*, Proceeding, 17–29, Sept., 1986.

[21] Bobrow D., L. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, D. Moon, Common Lisp, Object System Specification, *SIGPLAN Notices*, Vol. 23, special issue, Sept., 1988.

[22] Booch G., Software Engineering with Ada, *Benjamin Cummings Publishing Co.*, Menlo Park, CA, 1987.

[23] Bos J., Laffra C., PROCOL: A Parallel Object Language with Protocols, *OOPSLA*, Proceeding, 1989.

[24] Briot J., From Objects to Actors: Study of a limited Symbiosis in Smalltalk–80, ACM SIGPLAN Workshop on Object-Based Concurrent Programming, Proceeding, *SIGPLAN Notices*, 24(4):69–72, 1988.

[25] Brooks, No Silver Bullet, Essence and Accidents of Software Engineering, *Computer*, April, 1987.

[26] Brogi A., Lamma E., Mello P., Objects in a Logic Framework, Lectures Notes in Artificial Intelligence, 592, *Springer Verlag*, 1992.

[27] Budd T., An Introduction to Object-Oriented Programming, *Addison-Wesley Publishing Co.*, 1991.

[28] Bugliesi M., Lamma E., Mello P., Modularity in Logic Programming, *Journal of Logic Programming*, 19(20):443–502, 1994.

[29] Carlson B. *Compiling and Executing Finite Domain Constraints*, Ph.D. thesis, Uppsala University, Sweden, 1995.

[30] Chen W., Warren D.H., Objects as intensions, *Fifth International Conference on Logic Programming*, Proceeding, 1988.

[31] Chikayama T., *ESP Reference Manual*, *Technical Report*, TR–044, ICOT, Tokyo, 1984.

[32] Chikayama T., Unique Features of ESP, *FGCS'84*, Proceeding, 1984.

[33] Chusho T., Haga H., A Multilingual Modular Programming System for Describing Knowledge Information Processing Systems, in Kugler J. (ed.), Information Processing 86, *Elsevier*, North Holland, 1986.

[34] Conery J., Logical Objects, *Fifth International Conference on Logic Programming*, Proceeding, 1988.

[35] Cointe, P. Metaclasses are first class: the ObjVlisp model. Proceeding of *OOPSLA*, December 1987.

[36] Corradi A., Leonardi L. PO Constraints as tools to synchronize active objects, *Journal of Object-Oriented Programming*, 4(6):41–53, Oct., 1991.

[37] Cox, B. Object-Oriented Programming: An Evolutionary Approach, *Addison-Wesley Publishing Co.*, 1986.

[38] Dahl O., Nygaard K. Simula: An Algol Based Simulation language. *Comm. ACM*, 671–678, Sept., 1966.

[39] Dally W., Chien A., Object-Oriented Concurrent Programming in CST", ACM SIGPLAN Workshop on Object-Based Concurrent Programming, Proceeding, *SIGPLAN Notices*, 24(4):28–31, 1988.

[40] Daniels J., Cook S. Strategies for Sharing Objects in Distributed Systems, *Journal of Object-Oriented Programming*, 5(8):27–36, Jan., 1993.

[41] Davison A., *Polka: A Parlog Object Oriented Language*, Ph. D. thesis, Department of Computing, Imperial College of Science, Technology and Medicine, 1989.

[42] Davison A., Design Issues for Logic Programming-based Object Oriented Languages, *Research Report*, Department of Computing, Imperial College, London, 1991.

[43] Davison A., Survey of Logic Programming–Based Object-Oriented Languages, in Agha G., Wegner P., Yonezawa A.(eds.), Research Directions in Concurrent Object-Oriented Programming, *MIT Press*, 1993.

[44] *Digitalk Co.*, Smalltalk/V manual, 1988.

[45] Elshiewy N., Time, Clocks and Committed Choice Parallelism for Logic Programming of Real Time Computations. *SICS Research Report*, R86:13, Swedish Institute of Computer Science, 1986.

[46] Elshiewy N., Modular and Communicating Objects in SICStus Prolog, *FGCS'88*, Proceeding, ICOT, Tokyo, 1988.

[47] Elshiewy N., *Robust Coordinated Reactive Computing in Sandra*, Ph.D. thesis, Royal Institute of Technology, 1990.

[48] Elshiewy N., Logic Programming for Real Time Control of Telecommunication Switching Systems, *The Journal of Logic Programming*, 8(1), 1990.

[49] Feldman, S., Make–A program for maintaining computer programs, *Software Practice and Experience*, 9(4):255–266, 1979.

[50] Franzén, T. Logical Aspects of the Andorra Kernel Language. *SICS Research Report*, R91:12, Swedish Institute of Computer Science, 1991.

[51] Franzén, T. Some Formal Aspects of the Andorra Kernel Language. *SICS Research Report*, R94:10, Swedish Institute of Computer Science, 1994.

[52] Frolund S. Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages, *ECOOP*, Proceeding, 1992.

[53] Fukunaga K., Hirose S., An Experience with a Prolog–Based Object-Oriented Language, *OOPSLA*, Proceeding, 1986.

[54] Bracha, G., Cook, W., Mixin-based Inheritance, *ECOOP/OOPSLA*, Proceeding 1990.

[55] Girard J., Linear Logic, *Theoretical Computer Science*, 50(1), 1987.

[56] Goldberg A, Robson D. Smalltalk–80: The Language and its Implementation, *Addison-Wesley Publishing company*, 1983.

[57] Goldberg Y., Silverman W., Shapiro E., Logic Programs with Inheritance, *FGCS'92*, Proceeding, ICOT, Tokyo, 1992.

[58] Haridi, S., Janson S., Kernel Andorra Prolog and its computation model. In the Seventh International Conference of Logic Programming, Proceeding, *MIT Press*, 1990.

[59] Hassan A., *The WAM: A (Real) Tutorial, 5*, Paris Research Laboratory, France, 1990.

[60] Henderson P., Functional Programming: Applications and Implications, *Prentice-Hall International*, NJ, 1980.

[61] Henz, M., The Oz notation, *DFKI Oz documentation series*, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D–66123 Saarbrücken, Germany, 1994.

[62] Henz M., Mehl M., Müller M., Müller T., Niehren J., Schiedhauer R., Schulte C., Smolka G., Treinen R., Würtz J., The Oz Handbook, *Research Report*, RR-94-09, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D–66123 Saarbrücken, Germany, 1994.

[63] Henz, M., Smolka G., Würtz J., Object-Oriented Concurrent Constraint Programming in Oz, in Saraswat V., Hentenryck V. (eds.), Principles and Practice of Constraint Programming, *MIT Press*, Mass.,1995.

[64] Hewitt, C., Bishop, B., and Steiger, R. A universal, modular actor formalism for artificial intelligence. Proceeding of *IJCAI*, August 1973.

[65] Hewitt C., Viewing Control Structures as Patterns of Message Passing, *Journal of Artificial Intelligence*, 8(3):323–364, 1977.

[66] Hewitt C., Agha G., Concurrent Systems for Knowledge Processing: An Actor Perspective, *MIT Press*, Cambridge, Mass, 1989.

[67] Hoare C., Communicating Sequential Processes, *Comm. ACM*, 21(8):666–677, 1978.

[68] Hur J., Chon K., Overview of a Parallel Object-Oriented Language CLIX, *ECOOP*, Proceeding, 1987.

[69] Ishikawa Y., Tokoro M., A Concurrent Object-Oriented Knowledge Representation Language Orient84/K: Its Features and Implementation, *OOPSLA*, Proceeding, 1986.

[70] Ishikawa Y., Tokoro M., Orient84/K: An Object-Oriented Concurrent Programming Language for Knowledge Representation, in Yonezawa, A., Tokoro M. (eds.), Object-Oriented Concurrent Programming. Cambridge, *MIT Press*, 1987.

[71] Janson S., Haridi S., Programming Paradigms of the Andorra Kernel Language, *SICS research report*, Swedish Institute of Computer Science, 1991.

[72] Janson S., Montelius J., Design of a sequential prototype implementation of AKL, *SICS research report*, Swedish Institute of Computer Science, 1992.

[73] Janson S., Montelius J., Haridi S., Ports for Objects in Concurrent Logic Programs, in Agha G., Wegner P., Yonezawa A.(eds.), Research Directions in Concurrent Object-Oriented Programming, *MIT Press*, 1993.

[74] Janson S. AKL: *a Multi-paradigm Language*, Ph.D. thesis, Uppsala University, Sweden, 1994.

[75] Janson S. et. al., AGENTS Library Manual, version 1.0, *SICS technical report*, Swedish Institute of Computer Science, 1995.

[76] Kafura D., Concurrent Object-Oriented Real–Time Systems Research, ACM SIGPLAN Workshop on Object-Based Concurrent Programming, Proceeding, *SIGPLAN Notices*, 24(4):203–204, 1988.

[77] Kafura, D., Inheritance in Actor Based Concurrent Object-Oriented Languages, *ECOOP*, Proceeding, 1989.

[78] Kahn K., Tribble, D., Miller M., Bobrow D., Objects in Concurrent Logic Programming Languages, *OOPSLA*, Proceeding, 1986.

[79] Kahn K., Tribble, D., Miller M., Bobrow D., Vulcan: Logical Concurrent Objects, in Shapiro, E. (ed.), Concurrent Prolog, *MIT Press*, 1987.

[80] Kahn, K., Objects–A Fresh Look, *ECOOP*, Proceeding, 1989.

[81] Keisu T.: *Tree Constraints*, Ph.D. thesis, Royal Institute of Technology, Sweden, 1994.

[82] Keene S., Object-Oriented Programming in Common Lisp, *Addison-Wesley Publishing Co.*, Reading, Mass, 1989.

[83] Kernighan B., Ritcheie D., The C Programming Language, *Prentice Hall*, 1978.

[84] Khoshafian, S., Abnous R., Object Orientation: Concepts, Languages, Databases, User Interfaces, *John Wiely & Sons*, 1990.

[85] Kiczales G, Rivières, Bobrow D., The Art of Metaobject Protocol. *MIT Press*, 1993.

[86] Knudsen J., Name Collision in Multiple Classification Hierarchies, *ECOOP*, Proceeding, 1988.

[87] Knudsen J., Löfgren, Madsen O., Magnusson B., Object-Oriented Environments: The Mjolner Approach, *Prentice Hall*, 1994.

[88] Koschmann T., Evens M., Bridging the Gab Between Object-Oriented and Logic Programming, *IEEE Software*, 5(5):36–42, 1988.

[89] Kristensen B., Madsen O., Moller-Pedersen B., Nygaard K., The BETA Programming Language, in Shriver, B., Wegner P., (eds.), Research Directions in Object-Oriented Programming, *MIT Press*, 1987.

[90] Kühne, T., Higher Order Objects in Pure Object-Oriented Languages, *ACM SIGPLAN Notices*, 29(7):15–20, July, 1994.

[91] Lalonde W., Pugh J., Inside Smalltalk, Volume I & II, *Prentice-Hall, Inc.*, NJ, 1990.

[92] Lapalme G., Salle P., Plasma–II: An Actor Approach to Concurrent Programming, ACM SIGPLAN Workshop on Object-Based Concurrent Programming, Proceeding, *SIGPLAN Notices*, 24(4):81–83, 1988.

[93] Lieberman H., Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems, *OOPSLA*, Proceeding, 1986.

[94] Liberman, H. Concurrent Object-Oriented Programming in Act 1, in Yonezawa A. , Tokoro M. (eds.), Object-Oriented Concurrent Programming, *MIT Press*, Cambridge, 1987.

[95] Madsen O., Moller-Pedersen B., Nygaard K., Object-Oriented Programming in the BETA Programming Language, *ACM press, Addison-Wesley Publishing Co.*, 1993.

[96] Manning C., A Peek at Acore, an Actor Core Language, ACM SIGPLAN Workshop on Object-Based Concurrent Programming, Proceeding, *SIGPLAN Notices*, 24(4):84–86, 1988.

[97] Matsuoka S., Wakita K., Yonezawa A., Synchronization Constraints with Inheritance: What is not possible–So what is?, *Technical Report 10*, Department of Computer Science, the university of Tokyo, 1990.

[98] Matsuoka S., Taura K., Yonezawa A., Highly Efficient and Encapsulated Reuse of Synchronization Code in Concurrent Object-Oriented Languages, *OOPSLA*, Proceeding, 1993.

[99] Matsuoka S., Yonezawa A., Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages, in Agha G., Wegner P., Yonezawa A.(eds.), Research Directions in Concurrent Object-Oriented Programming, *MIT Press*, 1993.

[100] Matsuoka S., Language Features for Re-use and Extensibility in Concurrent Object-Oriented Programming Languages, Ph. D. thesis, Department of Information Science, the University of Tokyo, 1993.

[101] McCabe, F., Logic and Objects, *Research Report*, Department of Computing, Imperial College, London, 1987 .

[102] McCabe, F., Logic and Objects, *Prentice Hall International Series in Computer Science*, 1992.

[103] Mehrotra P., Concurrent Object Access in Blaze2, ACM SIGPLAN Workshop on Object-Based Concurrent Programming, Proceeding, *SIGPLAN Notices*, 24(4):55–58, 1988.

[104] Mello P., Matali A., Objects as communicating Prolog Units, *ECOOP*, Proceeding, 1987.

[105] Mello P., Concurrent Objects in Logic Programming Framework, ACM SIGPLAN Workshop on Object-Based Concurrent Programming, Proceeding, *SIGPLAN Notices*, 24(4):37–39.

[106] Meyer B., Object-Oriented Software Construction, *Prentice Hall*, NJ, 1988.

[107] Meyer, B. Harnessing Multiple Inheritance, *Journal of Object-Oriented Programming*, 1(4):48–51, Nov./Dec., 1989.

[108] Moon, D., Object-Oriented Programming with Flavors, *ACM Conference on Object-Oriented Systems, Languages, and Applications*, Proceeding, 9–16, Sept., 1986.

[109] Moss C., An introduction to Prolog++, *Technical Report*, Department of Computing, Imperial College, London, 1990.

[110] Moss C., Prolog++: The Power of Object-Oriented and Logic Programming Language, *Addison-Wesley Publishing Co.*, 1994.

[111] Murphy T., Software Review: LPA 386 Prolog, *AI Expert*, September 1993.

[112] Nelson, M., Concurrency and Object-Oriented Programming, *SIGPLAN Notices*, 26(10):63–72, Oct., 1991.

[113] Newton M., Watkins J., The Combination of Logic and Objects for Knowledge Representation, *Journal of Object-Oriented Programming*, November/December, 7–10, 1988.

[114] Oliveira, J., Suggestion for a Parameterized Class Model, *OOPS MESSENGER*, 5(3):27–36, July 1994.

[115] O'keef R., The Craft of Prolog, *MIT Press*, 1990.

[116] Ciancarini P., Levi G., What is Logic Programming good for in Software Engineering?, *Technical Report*, UBLCS–93–9, Laboratory for Computer Science, University of Bologna, Italy, 1993.

[117] Pereira L., Nasr R., Delta Prolog: A Distributed Logic Programming Language, *FGCS'84*, Proceeding, ICOT, Tokyo, 1984.

[118] Peyton S., The Implementation of Functional Programming Languages, *Prentice Hall*, New York, 1987.

[119] Pinson L., Wiener R., An Introduction to Object-Oriented Programming and Smalltalk, *Addison-Wesley Publishing Co.*, Reading, Mass, 1988.

[120] Pinson L., Wiener R., Objective–C, Object-Oriented Programming Techniques, *Addison-Wesley Publishing Co.*, 1991.

[121] Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W., Object-Oriented Modeling and Design, *Prentice Hall Inc.*, 1991.

[122] Sametinger J., chiffer S., Design and Implementation Aspects of an Experimental C++ Programming Environment, *Software Practice and Experience*, 25(2):111–128, 1995.

[123] Saraswat V., *Concurrent Constraint Programming Languages*, Ph. D. thesis, MIT Press, 1993.

[124] Schachte P., Saab G., Efficient Object-Oriented Programming in Prolog, *The Practical Application of Prolog*, Proceeding, 1994.

[125] Schaffert, C, Cooper T., Bullis B, Kilina, M., and Wilpolt, C. An Introduction to Trellis/Owl, *OOPSLA*, Proceeding, 9–16, 1986.

[126] Shapiro, E., A Subset of Concurrent Prolog and Its Interpreter, *ICOT Technical Report*, TR–003, 1983.

[127] Shapiro, E., Takeuchi A., Object-Oriented Programming in Concurrent Prolog, *Journal of New Generation Computing*, 1(1):25–49, 1983.

[128] Shapiro, E. (ed.), Concurrent Prolog, *MIT Press*, 1987.

[129] Silberschatz A., Peterson J., Galvin P., Operating Systems Concepts, *Addison-Wesley Publishing Co.*, 1991.

[130] Singh, G., Single Versus Multiple Inheritance in Object-Oriented Programming, *OOPS MESSENGER*, 5(1), January 1994.

[131] Smolka, G., The definition of Kernel Oz, *DFKI Oz documentation series*, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D–66123 Saarbrücken, Germany, 1994.

[132] Snyder A., Object-Oriented Programming for Common Lisp, *Report ATC-85-1*, Software Technology Laboratory, Hewlett-Packard Laboratories, Palo Alto, California, 1985.

[133] Snyder, A., Encapsulation and Inheritance in Object-Oriented Programming Languages, *OOPSLA*, Proceeding, 1986.

[134] Steel G, Common Lisp: The language, *Digital Press*, Digital Equipment Group, Bedford, Mass, 1984.

[135] Steigerwald R., Nelson M., Concurrent Programming in Smalltalk–80, *SIGPLAN Notices*, 25(8):27–36, 1990.

[136] Sterling K., Shapiro E., The Art of Prolog, *MIT Press*, 1994.

[137] Stroustrup, B., The C++ Programming Language, *Addison-Wesley Publishing Co.*, Reading, Mass., 1986.

[138] Stroustrup, B., C++ Reference Manual, *AT&T Bell Laboratories*, Murry Hill, NJ, 1989.

[139] Stroustrup B., A History of C++:1979–1991, History of Programming Languages Conference, Proceeding, *ACM SIGPLAN Notices*, 28(3):271–297, 1993.

[140] Teitelman W., Masinter L., The InterLisp Programming Environment, *Computer*, 14(4), 1981.

[141] Thomas D., LaLonde W., Duimovich J, Wilson M., Actra-Multitasking/ Multiprocessing Smalltalk, ACM SIGPLAN Workshop on Object-Based Concurrent Programming, Proceeding, *SIGPLAN Notices*, 24(4):87–90, 1988.

[142] Tripathi A., Berge E., An Implementation of the Object-Oriented Concurrent Programming Language SINA, *Software Practice and Experience*, 19(3):235–256, 1989.

[143] Ueda K., Guarded Horn Clauses, in Shapiro, E. (ed.), Concurrent Prolog, *MIT Press*, 1987.

[144] US Department of Defense, *Ada Reference Manual*, 1980.

[145] Uustalu T., Combining Object-Oriented and Logic Paradigms: A Modal Logic Programming Approach, *ECOOP*, Proceeding, 1992.

[146] Vasey P., Spenser C., Westwood D., Westwood A., *Prolog++ Programming Reference Manual*, Logic Programming Associates Ltd., London, England, 1990.

[147] Vaucher J., Lapalme G., Malenfant J. SCOOP: Structured Concurrent Object-Oriented Prolog, *ECOOP*, Proceeding, 1988.

[148] Warren D., An Abstract Prolog Instruction Set, *Technical Note #309*, Artificial Intelligence Center, SRI International, 1983.

[149] Warren D., Database Updates in Pure Prolog, *FGCS'84*, Proceeding, ICOT, 1984.

[150] Watanabe T., Yonezawa A., Reflection in an Object-Oriented Concurrent Language, *OOPSLA*, Proceeding, 1988.

[151] Wegner, P., The Object-Oriented Classification Paradigm, in Shriver, B., Wegner P., (eds.), Research Directions in Object-Oriented Programming, *MIT Press*, 1987.

[152] Wegner, P., Dimensions of Object-Based Language Design, *OOPSLA*, Proceeding, 1987.

[153] Wegner, P., Technical Contributions: Concepts and Paradigms of Object-Oriented Programming. *OOPS MESSENGER*, 1(1):7–87, August 1990.

[154] Wiener R., Pinson L., An Introduction to Object-Oriented Programming and C++, *Addison-Wesley Publishing Co.*, Reading, Mass., 1988.

[155] Wu S., Integrating Logic and Object-Oriented Programming, *OOPS MESSENGER*, 2(1):28–37, July 1991.

[156] Yonezawa, A., Briot, J., Shibayama E., Object-Oriented Concurrent Programming in ABCL/1, *OOPSLA*, Proceeding, 1986.

[157] Yonezawa, A., Tokoro M., Object-Oriented Concurrent Computing: An Introduction, in Yonezawa, A., Tokoro M. (eds.), Object-Oriented Concurrent Programming, Cambridge, *MIT Press*, 1987.

[158] Yonezawa, A., Shibayama E., Takada T., Honda Y., Modeling and Programming in an Object-Oriented Concurrent Language ABCL/1, in Yonezawa, A., Tokoro M. (eds.), Object-Oriented Concurrent Programming, Cambridge, *MIT Press*, 1987.

[159] Yokote, Y., Tokoro M., The Design and Implementation of ConcurrentSmalltalk, *OOPSLA*, Proceeding, 1986.

[160] Yokote, Y., Tokoro M., Concurrent Programming in ConcurrentSmalltalk, in Object-Oriented Concurrent Programming, Cambridge, *MIT Press*, 1987.

[161] Yokote, Y., Tokoro M., Experience and Evolution of ConcurrentSmalltalk, *OOPSLA*, Proceeding, 1987.

[162] Yoshida K., Chikayama T., A'UM–A Stream Based Concurrent Object-Oriented Language, *FGCS'88*, Proceeding, ICOT, Tokyo, 1988.

[163] Yoshida K., Chikayama T., $\mathcal{A}'\mathcal{UM}$ = Stream + Object + Relation, ACM SIGPLAN Workshop on Object-Based Concurrent Programming, Proceeding, *SIGPLAN Notices*, 24(4):55–58, 1988.

[164] Zaniolo C., Object-Oriented Programming in Prolog, *IEEE Symposium on Logic Programming*, Proceeding, NJ, 1984.

⋆ ⋆ ★ ⋆ ⋆