AKL+: A Concurrent Language Based on Object-Oriented and Logic Programming Paradigms

Khaled Shaalan^ Seif Haridi* Salwa El-Gamal^ Ahmed Rafea^

* Swedish Institute of Computer Science (SICS)
^ Institute of Statisitical Studies and Research (ISSR)

Abstract

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+ is a natural development which is derived from the fusion of concurrent constraint logic 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.

This paper is going to describe the AKL+ language. The schemes for developing an efficient implementation is discussed. The AKL+ language has 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).

1. Introduction

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. It is built on top of the concurrent constraint language AKL [12]. 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.

The AKL+ language has been implemented on UNIX-based workstations and it is part 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**).

Section 2 compares our language with the most related ones. Section 3 gives a brief summary of AKL. Section 4 looks at the basic elements of the AKL+ language. Section

5 presents the implementation aspects of our language and shows how we can implement AKL+ as an efficient programming language. Section 6 concludes the paper.

2. Previous work

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.

Mergers with a logic based language are the most numerous. The historically earliest proposal by Shapiro and Takeuchi [20] and Zaniolo [24] have had strong influence on the development of this group, and may thus be also viewed as belonging here.

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 [13,14] 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 [3] 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 Polka but the state is held as unshared argument in the process of the object.

Sandra [4,5] 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 [8] 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.

Oz [10,11] 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 [21], 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.

Other differences which hold with 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.

3. AKL

In this section, we will present AKL because it 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.

3.1 Basic Concepts

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*. These two operations provide the necessary primitives for concurrent communication and synchronization. The information in the store is expressed in terms of *constraints*, which are statements in some constraint language, usually based on first-order logic. 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 $X = 1$.

The notion of constraints in AKL 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 [2]. For the purpose of this introduction, we will use a simple constraint system with a few obvious constraints, which is essentially that of Prolog [22] and GHC [23]. Thus, constraints in AKL will be formulas of the form

    <expression>=<expression>
    <expression>≠<expression>
    <expression><<expression>

and the like.

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 [9,12,6,7].

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

<name>(X_1,..., X_n)

is a defined agent. In a procedure atom, <*name*> is the functor, an alpha-numeric symbol, and *n* is the arity, the number of arguments, of the atom. The variables X_1, ..., X_n 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 *name/n* atoms, which uniquely identifies the corresponding procedure atom.The behavior of atoms is given by *procedure (agent) definitions* of the form

<name> (X_1, ..., X_n) :=<statement>

The variables X_1, ..., X_n 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

<statement>, ..., <statement>

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

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>)
```

where 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. The guards of a choice execute with corresponding constraint stores. If the guard fails, the corresponding clause is deleted. If all clauses are deleted, a method fails. Conditional choice corresponds to if-then-else in Prolog. If the first remaining guard succeeds, the goal is replaced with the body of this clause (The clause is *promoted*). Nondeterminate choice corresponds to disjunction in Prolog. If only one clause remains, and its guard is successfully reduced, the choice is said to be *determinate*. The clause is then promoted. Otherwise, if there is more than one clause left, the choice statement is said to be *nondeterminate*, and it will wait. Subsequent invocations 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 promoted in different copies of the state. The alternative computation paths are explored concurrently. Commit choice corresponds to guarded clauses in committed-choice languages. If any of the guards is successfully reduced, the corresponding clause is promoted.

AKL exploits the module system facilities. Procedures declared as “public” in a module declaration are *exported*, e.g.

```
:- module calc.
:- 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*.

3.3 Objects

Objects are realized as processes 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 preserves the *identity* of the object. 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.

Example: 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
```

Finally, there are a few things to note about these objects. 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 [12], objects (agents) that communicate by posting and checking constraints upon bags.

Sixth, no syntactic support for object-oriented programming was proposed.

As shown, this model did not deal with some of the fundamental issues involved in object-oriented programming, such as multiple inheritance conflicts, self communication, the accessing of state variables and clauses. Consequently, the concurrent logic based object-oriented programming languages developed since can and should provide solutions to these problems.

As its name suggests, AKL is a programming language *kerne*l. 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.

4. AKL+

The notion of a *class* is central to AKL+: every object is an instance of a class. An AKL+ class determines the structure and behavior of the objects that are its instances. In AKL+, a class is declared by writing it in the form

:- **class** <class name>.
:- **supers** [<super$_1$>,...,<super$_a$>].
:- **attributes** [<attribute$_1$>,...,<attribute$_b$>].
:- **private** [<selector$_1$>,...,<selector$_c$>].
<method$_1$>.
...
<method$_m$>.

The *<class name>* is an expression of the form *<identifier>(X$_1$,...,X$_n$), n>=0,* and *X$_1$,...,X$_n$* represent variables. The *<identifier>* is an alpha-numeric symbol. *<method$_1$>, ..., <method$_m$>, (m>=0),* are *method definitions*. The superclasses are the classes given in the **supers** declaration. It is possible to be selective about what is from a superclass, so-called *differential inheritance*, by writing the superclass in the following form

<super >-[<selector$_1$>,..., <selector$_d$>]

where <selector$_i$> consists of the functor and arity of the excluded method. A method that its selector is given in the **private** declaration is hidden. AKL+ makes it possible to declare a list of attributes with their initializations. The attribute declaration takes the form

<attribute$_i$ > =(X)\\<statement>

where *attribute$_i$*, an alpha-numeric symbol denotes an attribute name, will be assigned its initial value returned through the (output) argument *X* after applying *<statement>*.

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 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,...,X_n)\\<statement>

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 = # <class name>

such that the *<class name>* can be passed along to any method or class. Defining classes and methods as abstractions provides for all higher-order programming techniques. The behavior of an object is given by method definitions of the form

<identifier>(X_1, ...,X_n):= <statement>

The *<identifier>* is an alpha-numeric symbol denotes the functor of the methods and *n* denotes its arity. The variables X_1,..., 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). It also inherits the concurrent constraint programming of its host language. AKL+ maintains a "threaded" state such that only one of the state-using sections in a method body can at the same time be entered. 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. 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*.

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, ..., X_n)

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

p (X_1,...,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, i.e. delegation preserves *Self*.

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

$p(X_1,...,X_n)$ <# 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

$p(X_1,...,X_n)$ # 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*. Simply, the base-class reference is an application of a method to *Self* which takes the form

p $(X_1,...,,X_n)$ # Self

and may have the syntactic sugar

p (X*1*,...,X*n*)

Method application. An application of a method abstraction bound to a variable *Y*, with the actual parameters *X1,...,Xn* takes the form

$Y(X_1,...,X_n)$

Message sending. A class can have a create method, e.g. *new*, for generating instances. 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. The message send takes the form

$p(X_1,...,X_n)$^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.

$$\begin{tikzcd}
State_0 \arrow[r, "message_1"] & State_1 \arrow[r, "message_2"] & \cdots \arrow[r, "message_n"] & State_n
\end{tikzcd}$$

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.

Note that the difference between the message sending and the method call is that the method call is performed immediately on the current state, whereas the other messages ma y be taken and change the state before the message is received.

4.1 Standard Classes

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 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. 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 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" [15,16,17,18].

4.2 Implicit Behavior

A set of implicit behavior is defined for each class definition. These are attributes manipulation, class membership, method dispatcher, and default methods.

4.2.1 Attribute Methods

Attributes declaration implicitly define a set of methods which has a behavior describing how attributes is manipulated.

Example: 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

```
:- 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. It is very useful in defining generic attribute access and update methods where the attribute name may not known until run-time, e.g. the definition of the generic attribute access method may be as follows:

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

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. In this case the attribute access and update methods should be provided.

4.2.2 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. A main usage of this method, is the possibility of defining *class-specific methods* [1]. When a class-specific method is invoked the appropriate method is executed on the basis of the identity of the target classes.

Example: 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* that computes the volume 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).
```

4.2.3 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*.

4.2.4 Default Methods

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 by 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: 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 unknown messages to the target object

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

4.3 Generic Classes

A class gains a generic property by associating it with parameters. The scope of these parameters is the class methods. A parameter of a parameterized class lies in four categories: class abstraction, constant, object, or method abstraction.

Example: A good example for showing the usefulness of generic classes is a general sorting algorithm. 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 sorting as follows.

```
:- class sorter(Method).
:- supers [ur_object].

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

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

4.4 Synchronization

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 [15] and further defined in [16,17]. 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. 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. *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. The following example shows the definition of a bounded buffer with *synchronizers.*

Example: 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 nonfull) 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 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.

5. 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 constant time.

5.1 The Class Expansion

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.

```
:- class  ordlist.
:- 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) # ordlist).

insert_aux(El, E, YesNo, L1, L):=
        ( YesNo = yes → L = [El,E | L1]
        ; YesNo = → L = [E | L2],
           insert(El, L1, L2) # ordlist ).

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/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,_)).
```

The *ord_list* class is expanded with additional definitions: *typeof/4*, *dispatch/4*, *init_list/4*, *domain/4*, *attribute_property/7*, and *public* definitions.

The definition to which a method expands depends on whether or not the class being defined is parameterized:

$$\text{Method} : \begin{cases} \text{Message X Myself X Self X State} \rightarrow \text{State} \\ \text{Message X Self X State} \rightarrow \text{State} \end{cases}$$

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 optimized by considering the first argument indexing of the method code.

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.

5.2 Efficiency

Object-Oriented languages have an undeserved reputation for inefficiency because some early languages (Smalltalk and Lisp-based languages) were interpreted rather than compiled [19]. 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. 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. Conclusion

This paper has been concentrated on issues in the design and implementation of 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. It is 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.

References

1. APaepcke, A. (ed.), Object-Oriented Programming: The CLOS perspective. *MIT Press*, 1993.
2. Carlson B., *Compiling and Executing Finite DomainConstraints*, Ph.D. thesis, Uppsala University, Sweden, 1995.
3. Davison A., *Polka: A Parlog Object Oriented Language*, Ph. D. thesis, Department of Computing, Imperial College of Science, Technology and Medicine, 1989.
4. Elshiewy N., Modular and Communicating Objects in SICStus Prolog, *FGCS'88*, Proceeding, ICOT, Tokyo, 1988.
5. Elshiewy N., *Robust Coordinated Reactive Computing in Sandra*, Ph.D. thesis, Royal Institute of Technology, 1990.
6. Franzen, T. Logical Aspects of the Andorra Kernel Language. *SICS Research Report*, R91:12, Swedish Institute of Computer Science, 1991.
7. Franzen, T. Some Formal Aspects of the Andorra Kernel Language. *SICS Research Report*, R94:10, Swedish Institute of Computer Science, 1994.
8. Goldberg Y., Silverman W., Shapiro E., Logic Programs with Inheritance, *FGCS'92*, Proceeding, ICOT, Tokyo, 1992.

9. Haridi, S., Janson S., Kernel Andorra Prolog and its computation model. In the Seventh International Conference of Logic Programming, Proceeding, *MIT Press*, 1990.

10. Henz M., Mehl M., Muller M., Muller T., Niehren J., Schiedhauer R., Schulte C., Smolka G., Treinen R., Wurtz J., The Oz Handbook, *Research Report*, RR-94-09, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrucken, Germany, 1994.

11. Henz, M., Smolka G., Wurtz J., Object-Oriented Concurrent Constraint Programming in Oz, in Saraswat V., Hentenryck V. (eds.), Principles and Practice of Constraint Programming, *MIT Press*, Mass.,1995.

12. Janson S., *AKL: a Multi-paradigm Language*, Ph.D. thesis, Uppsala University, Sweden, 1994.

13. Kahn K., Tribble, D., Miller M., Bobrow D., Objects in Concurrent Logic Programming Languages, *OOPSLA*, Proceeding, 1986.

14. Kahn K., Tribble, D., Miller M., Bobrow D., Vulcan: Logical Concurrent Objects, in Shapiro, E. (ed.), Concurrent Prolog, *MIT Press*, 1987.

15. 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.

16. Matsuoka S., Taura K., Yonezawa A., Highly Efficient and Encapsulated Re-use of Synchronization Code in Concurrent Object-Oriented Languages, *OOPSLA*, Proceeding, 1993.

17. 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.

18. 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.

19. Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W., Object-Oriented Modeling and Design, *Prentice Hall Inc*., 1991.

20. Shapiro, E., Takeuchi A., Object-Oriented Programming in Concurrent Prolog, *Journal of New Generation Computing*, 1(1):25-49, 1983.

21. Snyder, A., Encapsulation and Inheritance in Object-Oriented Programming Languages, *OOPSLA*, Proceeding, 1986.

22. Sterling K., Shapiro E., The Art of Prolog, *MIT Press*, 1994.

23. Ueda K., Guarded Horn Clauses, in Shapiro, E. (ed.), Concurrent Prolog, *MIT Press*, 1987.

24. Zaniolo C., Object-Oriented Programming in Prolog, *IEEE Symposium on Logic Programming*, Proceeding, NJ, 1984.