University of Pretoria etd – Peer, E S (2005)

Chapter 6

Computational Intelligence Library

"Ah, well, I am a great and sublime fool. But then I am God's fool, and all His work must be contemplated with respect." — Mark Twain

CILib (Computational Intelligence Library) is a software framework designed to accommodate scientific research in Computational Intelligence, providing implementations for many CI algorithms, problems definitions and a simulator for conducting experiments. In order to maximise collaboration and solicit third party peer review, CILib is published under the GNU GPL (refer to Section 4.1.6) and is available for download from SourceForge[1]. The following high level project goals were identified:

- **Flexibility:** Design patterns should be exploited to create a reusable framework capable of supporting the complexity of the CI field. Whenever possible, hybrid algorithms and new functionality should be achieved by composing various existing classes in a pluggable fashion.
- **Experimentation:** The framework should facilitate scientific experimentation, making it possible to measure any property of an algorithmic simulation. Different simulations, in terms of various class compositions and algorithm parameters, should be configurable at run time without making changes to the source code.
- **Efficiency:** It is commonly accepted that developer time is more expensive than CPU time, however, CI algorithms can be very computationally intensive. Thus,

[1] http://cilib.sourceforge.net

University of Pretoria etd – Peer, E S (2005)

a scientific simulation framework may at times have to trade off clean OO design against improved performance.

- **Separability:** There should be a clean separation of algorithms and problems, so that any algorithm can be applied to any suitable problem. Further, algorithms should be independent of any scientific simulation and measurement components, so that algorithms can also be used in non-research applications.

- **Reliability:** The open source peer review process should increase the probability of any software errors being found and corrected. A clean OO design and extensive unit testing should be used to further reduce any chance of errors.

- **Collaboration:** The framework should maximise collaborative opportunities. By sharing a common open source code base, researchers may be more aware of what others are doing and can reuse parts of the framework developed by others without reinventing the wheel. Good documentation should be provided to keep the barrier to entry as low as possible.

Section 6.1 recommends some coding conventions for CILib developers. Following that, the implementation details of CILib are covered in Section 6.2. Collaborative contributions to CILib are mentioned in Section 6.3. Finally, some limitations of the framework are discussed in Section 6.4.

6.1 Coding Conventions

To date, no coding conventions have been enforced on contributions to CILib, however, it is the recommendation of this work that developers adopt the Java coding conventions published by Sun Microsystems [57], which reflect those presented in the Java Language Specification [59]. A single coding standard is necessary despite the fact that developers may have different stylistic preferences. Adopting a standard results in code that can be unambiguously understood and easily read, since developers know what to expect even though it may not be their personal preference. This is particularly important in an open source context, where the source code itself is a primary means of communication between developers.

University of Pretoria etd – Peer, E S (2005)

The specification outlines some guidelines pertaining to the commenting of code. Java supports two types of comments, namely implementation comments and doc comments. Implementation comments apply to the implementation details of the code itself, while doc comments can be extracted as separate documentation independent of the code using the Javadoc tool [33]. Doc comments should be used to describe the purpose and function of interfaces, classes and methods in an implementation independent way. Implementation comments should be kept to a minimum, the code should rather be made as self documenting as possible, since comments can easily fall out of synchronisation with the code. Good doc comments, design patterns, unit testing and careful consideration of the naming of methods and identifiers should be sufficient documentation for any developer to understand the implementation. If the implementation is not self documenting then there is probably something wrong with the design that needs to be fixed. In the case of implementations of research algorithms, a proper reference to any pertinent articles should be provided in the doc comments for the implementing class. JUnit tests (refer to Section 5.5) should be provided whenever possible. Unfortunately, the stochastic nature of many of the algorithms in CILib means that a researcher is not likely to know what its acceptable behaviour should be, which is typically what is being researched in the first place.

Further, the specification lists naming conventions. A convention of prefixing a package name with the reversed Internet domain of the package owner should be followed, to ensure there are no conflicts in the package namespace, hence CILib packages fall in a hierarchy under `net.sourceforge.cilib`. Interface and class names should be mixed case with the first letter of each word capitalised. Abbreviations should be avoided. Methods and variables follow the same convention except that the first character is lower case. Constants should be written in upper case with underscores as word separators.

Finally, the document specifies formatting conventions. A particularly contentious issue, particularly with C/C++ developers, is the Java convention of having opening braces for blocks at the end of the line that defines the block. Closing braces should be indented to align with the statement, method or class that forms the start of the block. A level of indentation is defined to be four spaces. Further, a space should occur between keywords and parentheses, after commas in an argument list, between binary operators, except the class membership operator, between expressions in a `for` statement and after a type cast. Blank lines should also be used liberally to group related sections of code,

University of Pretoria etd – Peer, E S (2005)

especially between blocks and methods. Lastly, parentheses should be used to group arguments in complicated expressions to make them easier to read, instead of relying on the reader's knowledge of operator precedence rules.

6.2 Implementation Details

CILib's implementation is heavily based on design patterns (refer to Chapter 3) to maximise its flexibility. The type system used for representing problem domains is discussed in Section 6.2.1. CILib's representation for problems and implementation of algorithms are discussed next in Sections 6.2.2 and 6.2.3 respectively. Section 6.2.4 demonstrates the framework's facilities using particle swarms as a specific example. Stopping criteria for iterative algorithms is handled in Section 6.2.5. Finally, scientific experimentation is supported by measurements, in Section 6.2.6, and a simulator, which is covered in Section 6.2.7.

6.2.1 Domains and Types

Domains define a type system based on a string representation of a data type. A partial grammar for describing types consisting of combinations of bits, integers and real values is provided in Figure 6.1. These domains are used to describe, amongst other things, the search domains of computational intelligence problems. For example, a multi-dimensional real valued optimisation problem, as described in Section 2.1.1, would have a domain representation of "R^N", where N is replaced with the actual dimension of the problem. A genetic program which searches a tree space (refer to Section 2.3.2) might operate on a domain characterised by a description of the valid non-terminal nodes, a list of terminal symbols and a maximum tree depth.

Vectors of any given type are represented by composite and compound domain components. A compound represents a repetition of a type, while a composite is used to represent a mixture of different types. Further, compound components can represent variable length vectors.

For example, the compound type "Z^5" represents 5 dimensional vectors of integers. Equivalently, the composite "[Z,Z,Z,Z,Z]" represents the same 5 dimensional vector type. Compound domains permit constructs such as "Z^3~2", which represents an integer

University of Pretoria etd – Peer, E S (2005)

domain	::=	type | compound | composite
composite	::=	'[' domain { ',' domain } ']'
compound	::=	domain '^' int ['~' int]
type	::=	'B' | 'Z' ['(' [int] ',' [int] ')'] | 'R' ['(' [real] ',' [real] ')']
real	::=	int ['.' digit_sequence] [('e' | 'E') int]
int	::=	['+' | '-'] digit_sequence
digit_sequence	::=	digit { digit }
digit	::=	'0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

Figure 6.1: Partial Domain Grammar

vector type of length ranging between 3 and 5 inclusive. That is, the second number which follows the tilde symbol, corresponds to the amount of slack permitted by the type. A composite type permits constructs such as "[R,R,R,Z,Z]", or equivalently "[R^3,Z^2]", which represents a mixed vector type of 3 real values followed by 2 integers. Note that compound and composite types can be arbitrarily nested.

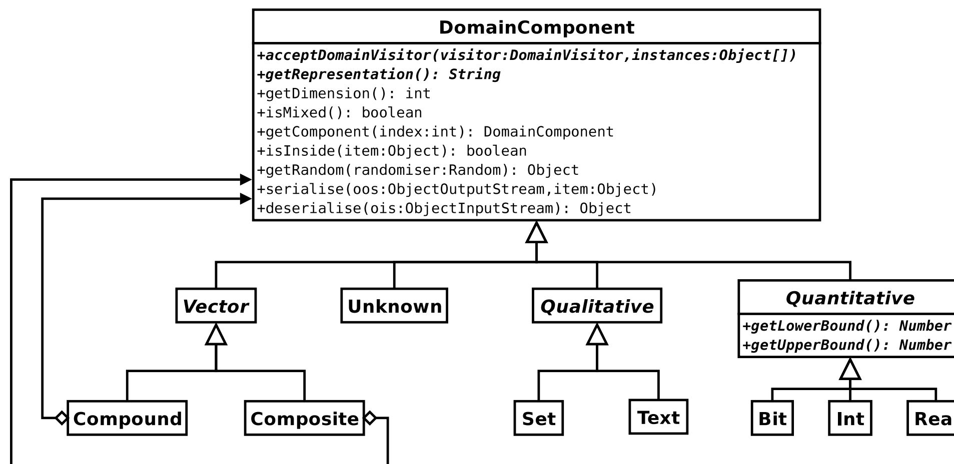

Figure 6.2: Domain Composite/Interpreter

Figure 6.2 illustrates how types are mapped into a *Composite* (refer to Section 3.2.2) object structure. The object structure can also be considered to be an instance of the *Interpreter* pattern, in Section 3.3.1, since the class hierarchy, although it has slightly more structure, to a certain extent mirrors the grammar. A *Singleton* (refer to Sec-

University of Pretoria etd – Peer, E S (2005)

tion 3.1.4) component factory is responsible for parsing domain strings and constructing their corresponding domain description, in terms of a hierarchy of domain components.

Types are divided into three categories: the composite and compound vector types which have already been discussed; qualitative types which represent ordinal or nominal data [106]; and quantitative types which represent numeric data. The quantitative types have the option of declaring bounds. In the grammar, these bounds are represented between parentheses. For example, a multi-dimensional search space bounded by $[-1, 1]$ in each dimension is represented by the string "R(-1,1)^N", where N is the dimension of the search space. Alternatively, a composite vector can be used to represent different bounds in each dimension. Lower and upper bounds are taken to be $-\infty$ and ∞ respectively if they are not specified.

The string representations for integer, real value, and vector types have already been discussed. Bits are represented by the string "B". String types are represented by the text component with representation "T". Sets are represented by the prefix "S" followed by a comma separated list of valid elements between braces. Graphs and trees might, in future, be represented by a prefix "G" followed by a list of terminal and non-terminal node descriptions. Any type which is not incorporated into the domain hierarchy is allocated an unknown type with representation "?".

The most important function of the domain hierarchy is producing random instances of a type, which are used as initial points in search spaces for optimisation algorithms. Care has been taken to return the most efficient concrete instance of any given domain. For example, a single bit returns a `java.lang.Byte` with a value of one or zero, but a vector of bits returns a `java.util.BitSet` instead of a memory inefficient array of bytes. Vectors of integers and real values return arrays of their respective `int` and `double` primitive types, which provide for the most efficient processing without polymorphic object overheads. Mixed composites return an array of generic objects containing as elements the largest possible groupings of more specific types. For example, the domain string "[R^30,B^20]" would result in a domain hierarchy that returns instances of the form `Object[] { double[30], BitSet }`, where the `size()` method of the bit set has been overridden to return the logical number of bits, in this case 20, as opposed to the actual number of bits used by the implementation. All a client of the domain hierarchy need do is cast the result into the type it expects. Domain validators are provided in the `net.sourceforge.cilib.Domain.Validator` package in order for a client to test those

University of Pretoria etd – Peer, E S (2005)

expectations before performing any casts. Clients that support multiple domains must query the domain hierarchy to determine what instances of the domain will look like and deal with them appropriately.

Beyond generating random instances inside a domain, a client may query: the dimension of a domain; whether a multi-dimensional domain contains mixed types; whether a given instance falls within the domain; and in the case of quantitative types, the bounds. The methods to get the dimension and the ith component of a vector present a flattened view of nested compound and composite vectors, so that indexing components does not need to take into account any effect of nesting. This means, equivalent domains, such as "[R^10,R^20]", "[R^20,R^10]" and "R^30", are identical from the client's perspective, even though they all have different hierarchical structures.

Measurements (refer to Section 6.2.6) are another aspect that require domain information, since they can be of any type and a common measurement interface is desired. The serialisation methods are provided so that instances of a domain, particularly measurements, can be stored and retrieved in a more space efficient fashion than the standard Java serialisation method.

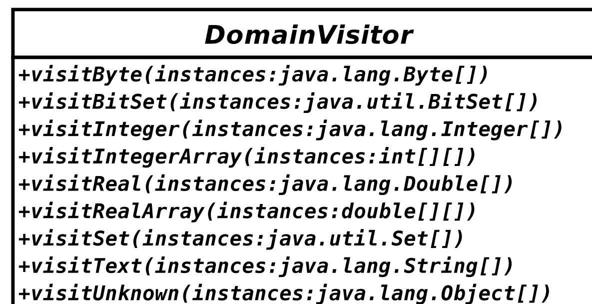

Figure 6.3: Domain Visitor Interface

Unfortunately, there are some design flaws in the domain strategy presented here. The most important being that clients cannot treat type instances in a uniform way, because the types described by a domain do not share a useful polymorphic interface. That is, a client needs to explicitly know how to deal with every type of domain that it supports. For example, an algorithm capable of dealing with both real valued and bit vectors needs to query the domain, directly or using validators, and conditionally execute one of two branches, one for each type, even though both branches probably

University of Pretoria etd – Peer, E S (2005)

contain similar logic. The domain *Visitor* (refer to Section 3.3.6) interface presented in Figure 6.3 alleviates this problem slightly by providing a cleaner interface for clients, but it is still clumsy and confusing, since an array of instances on which the visitor operates needs to be passed around, and its implementation is currently not very speed efficient.

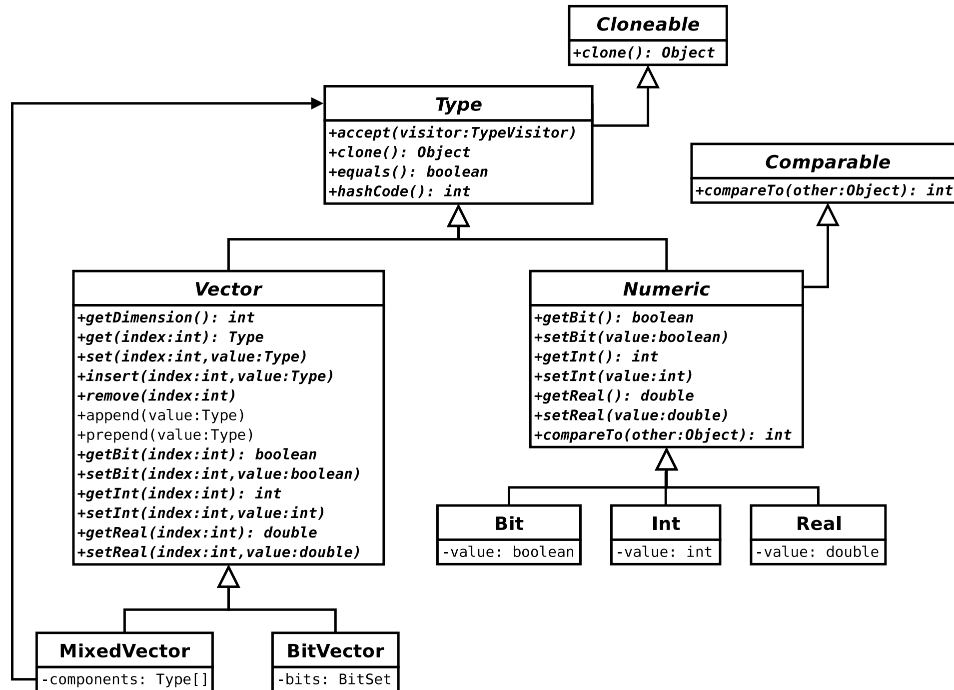

Figure 6.4: Partial Type System

The proper solution, assuming the object in-lining technology mentioned in Section 5.2 gets incorporated into future compilers, is to implement a polymorphic type system. The JFC already provide for numeric types using the `java.lang.Number` hierarchy. Unfortunately, this hierarchy consists of immutable numeric types, requiring object creation and collection overheads for even simple arithmetic operations, which are likely to be executed in tight loops by many algorithms. Thus, work has begun on the polymorphic type system presented in Figure 6.4.

Note that a client need only care whether it works on a vector or non-vector type, which is fine, since, for the most part, it will be one or the other exclusively. A client that does not care about the specific numeric type with which it works can simply utilise whichever units are most convenient. Those clients that do need to differentiate them, can make use of a more traditional *Visitor* (refer to Section 3.3.6) interface which does

University of Pretoria etd – Peer, E S (2005)

not require instances to be passed around as an additional parameter. Further, bit vectors and other arbitrary vectors present a uniform interface, meaning clients will not need to treat vectors of bits as a special case, while still benefiting from the storage efficiency of a bit set.

The problem with the type system presented in Figure 6.4 is that domain information cannot safely or efficiently be incorporated into the hierarchy. Bounds on numeric types and constraints on vectors can be cleanly implemented using *Decorators* (refer to Section 3.2.3), however, the extra level of indirection will have a severe performance penalty for types used in tight loops. In addition, bounds information which would be shared by a compound domain must be inefficiently stored for each individual vector component along with an additional memory reference. Further, although it may seem like a good idea to store the domain information implicitly in the type system, because clients have the freedom to modify the type, the integrity of the domain information may be compromised. For example, if the type system keeps track of the fact that it is an instance of “R^N” simply by virtue of the fact that it is a vector of real values, then a client which changes a component into an integer would alter the domain as a side effect. Finally, while serialisation can be supported in the type system relatively cleanly, deserialisation and generating random instances within a specified domain become very clumsy, since the type instance which would contain the necessary information does not yet exist.

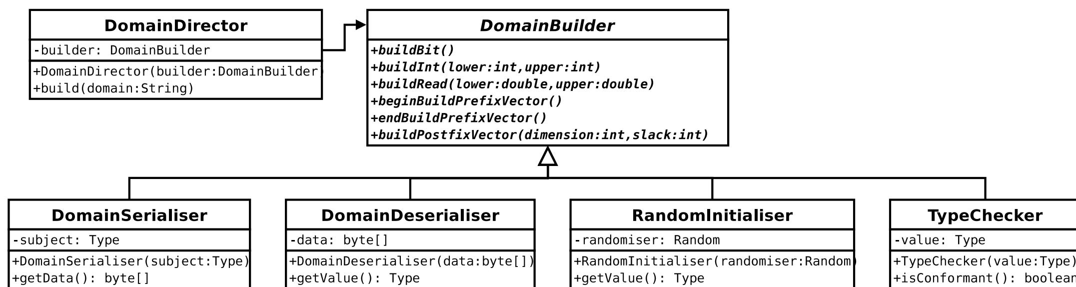

Figure 6.5: Domain Builder

The limitations of the type system just described seem to indicate that a parallel domain hierarchy still needs to be maintained, however, another possibility that is currently being investigated is the use of the *Builder* pattern (refer to Section 3.1.2) as illustrated in Figure 6.5. Instead of storing a domain hierarchy explicitly, only the original domain

University of Pretoria etd – Peer, E S (2005)

string is stored and different concrete builders are used to realise the same functionality. For example, a type checker can be used to determine whether a given type instance conforms to the domain string passed to the builder.

6.2.2 Problem Classes

Figure 6.6 demonstrates how the broad problem classes defined in Section 2.1 can be represented in software. The optimisation problem interface is characterised by: a domain, which defines the search space; and a fitness function, which evaluates the goodness of a given solution. Route optimisation problems, such as the TSP (refer to Section 2.1.2, are simply characterised by the graphs that define their routing networks. Both supervised and unsupervised learning problems are characterised by their data sets. In the case of supervised problems, patterns consist of an input part and a target part, which is encapsulated by the `Pattern` type. Both provide traversals of the data set using an *Iterator* (refer to Section 3.3.2). Patterns may conform to different domains, which are accessible via the respective problem interfaces. Additionally, unsupervised problems provide information about the number of clusters inherent in the data set, or alternatively, the constant `UNKNOWN_CLUSTERS` if such information is unknown.

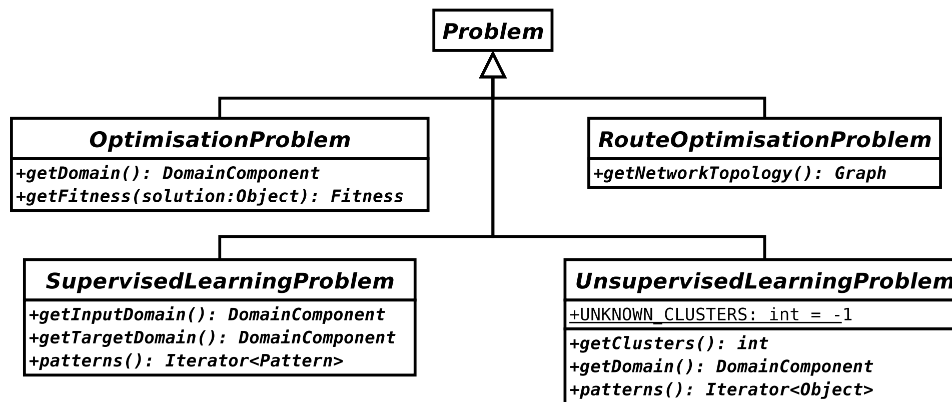

Figure 6.6: Problem Interfaces

These problem interfaces need to be implemented by concrete problem classes that take into account any context specific to a given situation. Concrete problems that are defined in terms of data sets, which can be true of any type of problem, can access their data via the `net.sourceforge.cilib.Problem.DataSet` interface. The data set

University of Pretoria etd – Peer, E S (2005)

interface does not enforce any structure on data. It simply provides input stream and byte array views of the raw data. The responsibility of interpreting the data falls upon the concrete problem implementation. Some problems may have their data represented as a structured XML document, while others may be constrained to operate on less structured data defined by the context of the problem. For example, a clustering problem defined for banking data may be constrained to the data format utilised by the bank's database. Each new application may require another concrete problem description, which encapsulates the characteristics of the application domain, presenting itself in terms of one of the general problem interfaces. The general framework will need to be extended as new problems arise which cannot fit into the model presented in Figure 6.6.

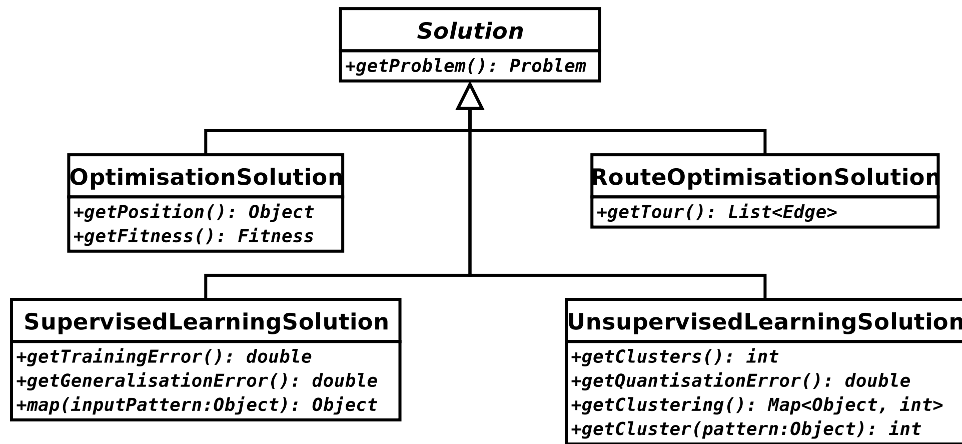

Figure 6.7: Solution Classes

Figure 6.7 shows the solutions corresponding to the given problem interfaces. First and foremost, solutions must exist within the context of some problem, hence there is a method providing access to their problems. The solution to an optimisation problem is characterised by a position and its fitness. Route optimisation solutions consist of an ordered list of the edges of the graph that form the optimal tour. The learning problems have solutions that are characterised by a model that fits the data. In the case of supervised problems, the model provides a method to determine the mapping for unseen input patterns, while an unsupervised model provides a method to determine the cluster index for an unseen pattern and access to the clustered training data. Both provide methods for determining the accuracy of the learned model.

Figure 6.8 illustrates some further specialisations of optimisation problems. Multi-

University of Pretoria etd – Peer, E S (2005)

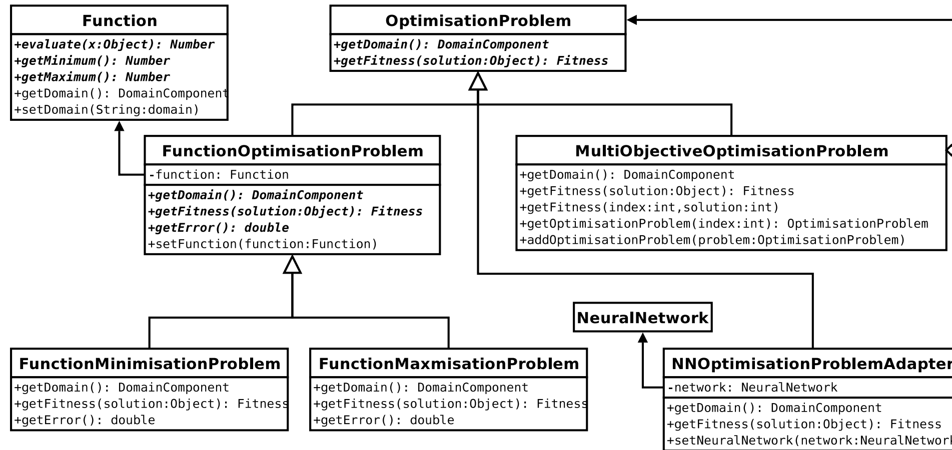

Figure 6.8: Optimisation Problems

objective optimisation problems turn the hierarchy into a *Composite* (refer to Section 3.2.2) so that a multi-objective problem still presents a single objective view, while permitting access to individual objectives for algorithms that support multi-objective optimisation. While the neural network code is currently in an incomplete state, it is easy to imagine a problem *Adapter* (refer to Section 3.2.1) that enables neural network training by means of an optimisation algorithm. In a research context, it is desirable to test optimisation algorithms on various benchmark functions. For this reason, an extensive set of benchmark functions is provided in the `net.sourceforge.cilib.Functions` package. Another *Adapter*, the `FunctionOptimisationProblem` class provides the glue between the optimisation problem interface and a benchmark function. Function optimisation is further specialised into minimisation and maximisation problems, which respectively minimise and maximise a benchmark function.

Earlier versions of CILib treated fitness as a single `double` value, which was negated in the case of function minimisation problems, so that larger values of fitness always indicated a more optimal solution. This simplistic approach had limitations when working with constrained optimisation problems, since constraint handling code needs access to the unaltered function surface. The fitness hierarchy in Figure 6.9 was introduced to solve this problem. Fitnesses now implement the comparable interface so that a fitness, when compared, performs the necessary transformation for minimisation problems, while still leaving the original function value accessible. Thus, fitness is always maximised, even for

University of Pretoria etd – Peer, E S (2005)

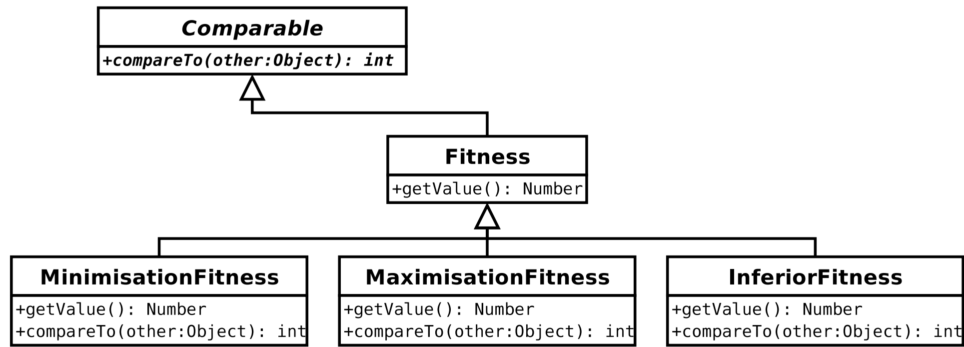

Figure 6.9: Fitness Classes

minimisation problems. The inferior fitness class always compares worse than other fitnesses, and is ideal for initialising the fitness of individuals in a population based search algorithm that have not yet been evaluated. Switching to a fitness type hierarchy also added the flexibility to handle discrete optimisation problems in a uniform way.

6.2.3 Algorithms

The `Algorithm` class, depicted in Figure 6.10, implements behaviour common to all iterative CI algorithms. These responsibilities include handling stopping criteria, notification of algorithm events, presenting an interface for threads and any other common house-keeping tasks.

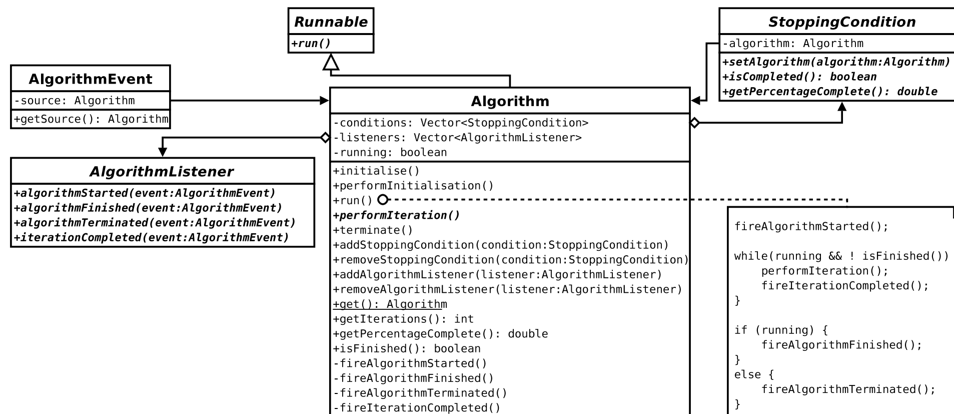

Figure 6.10: Algorithm, Stopping Conditions and Events

University of Pretoria etd – Peer, E S (2005)

The `run()` method is an example of a *Template Method* (refer to Section 3.3.5), which delegates the responsibility for executing a single iteration of the algorithm to a subclass that must override the abstract `performIteration()` method. The `initialise()` method is also a *Template Method*, performing initialisation tasks common to all algorithms before deferring to the `performInitialisation()` method, which is responsible for any algorithm specific initialisation, if necessary.

Stopping conditions monitor the progress of an algorithm, providing two methods to measure this progress. Firstly, the `isCompleted()` method is called for every iteration to determine when execution of the `run()` method should finish. Second, the `getPercentageCompleted()` method, which is typically more expensive to calculate, is primarily intended for updating progress indicators in a user interface, but can also be used as a value that increases linearly (depending on the particular stopping condition being used) over the execution duration for those algorithms that need it. Multiple conditions are accommodated simultaneously by maintaining them in a list, so that `isFinished()` returns true as soon as any one of the stopping conditions fires and `getPercentageComplete()` returns the average over all the conditions.

The event interface, which is an extension of the *Observer* pattern (refer to Section 3.3.3), is used to notify a list of observers, or listeners, whenever an algorithm, is started, finished, terminates early or completes an iteration. Unlike the basic *Observer*, which provides a listener with very little information about the subject, the event interface provides information about the kind of event that occurred as well as the source of the event, enabling many-to-many relationships between algorithms and listeners.

The class scope `get()` method returns a thread local instance of the algorithm which is currently executing. This provides a global method for objects lower down in the object reference graph to access the root algorithm class, so that they can navigate from that point to any required object. This contributes to keeping many interfaces simpler, reducing the need to pass additional objects around that are only used in rare circumstances. Also, it enables objects to access parts of the reference graph that were unforeseen in the design of certain interfaces. Unfortunately, there is a major problem with this approach, which is yet to be resolved, an object lower in the hierarchy may not know how to navigate the reference graph, since classes may be composed differently at run time.

An interface for accepting problems is not specified by the `Algorithm` class, since

University of Pretoria etd – Peer, E S (2005)

only its subclasses know what kind of problems they can be applied to. Figure 6.11 illustrates how optimisation problems fit into the CILib framework, showing that any algorithm implementing the `OptimisationAlgorithm` interface can be applied to an `OptimisationProblem`. For example, since PSO implements `OptimisationAlgorithm`, it can be applied to solve optimisation problems. Algorithm interfaces for other types of problems, such as routing or learning, can be implemented in a similar fashion. Having an algorithm interface for each type of problem enables an algorithm to be selective about the problems it can be applied to. Also, an algorithm may implement any number of these interfaces simultaneously, one for each type of problem that it can be applied to. For example, a feed forward neural network (refer to Section 2.2.1) would accept a `SupervisedLearningProblem`, while a SOFM (refer to Section 2.2.4) would accept unsupervised learning problems in addition to supervised learning problems.

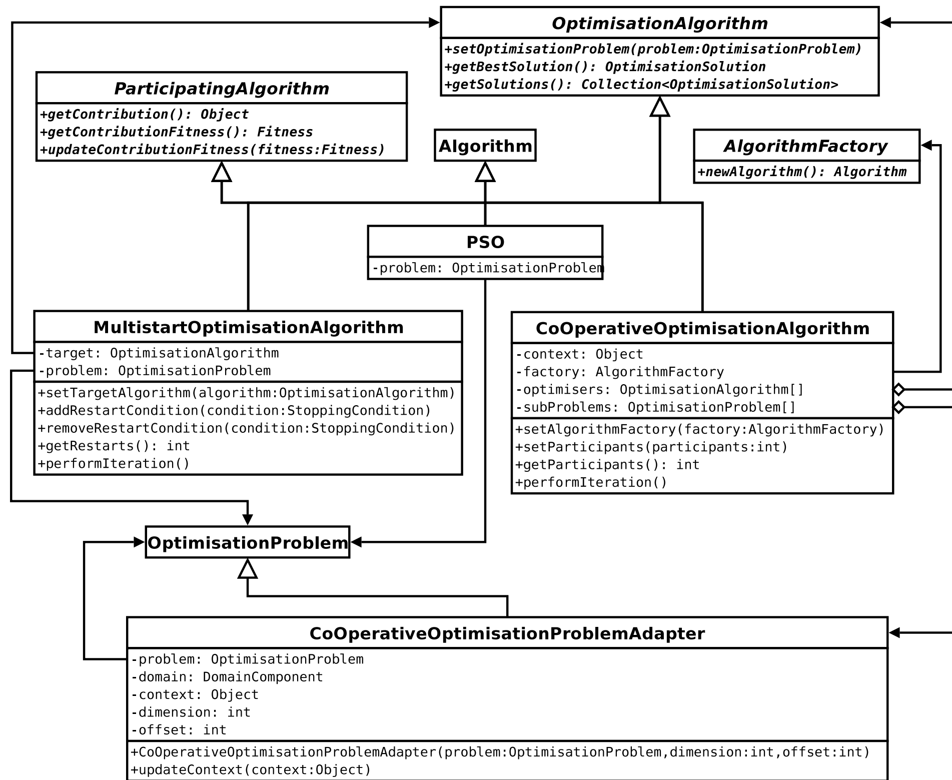

Figure 6.11: Optimisation Algorithms

Focusing again on optimisation problems, it is clear that any optimisation algorithm, including EC algorithms such as GAs, and not only PSOs can be implemented within

University of Pretoria etd – Peer, E S (2005)

the CILib framework by simply implementing the `OptimisationAlgorithm` interface, however, care should be taken to factor out any commonalities so that they can be reused and composed in various ways.

For example, the multi-start PSO (MPSO) [113] calls for restarting a PSO multiple times in order to find better solutions, since a PSO may prematurely converge onto suboptimal local extrema. By realising that this behaviour is generally applicable to all optimisation algorithms and not only PSOs, it can be factored out into a generic multi-start optimisation algorithm. The multi-start optimisation algorithm re-initialises a target algorithm whenever a restart condition is satisfied. For example, in the case of a PSO it may be appropriate to restart the algorithm whenever the average distance between particles drops below a certain threshold. This threshold would need to be captured in a stopping condition and applied to the multi-start algorithm as a restart criterion. Thus, any optimisation algorithm can have multi-start behaviour, provided a suitable restart condition can be defined. Indeed, it may be sensible to make this behaviour more general still, so that it can be applied to any algorithm as opposed to only optimisation algorithms. Such refactoring will be performed when it becomes evident how best to achieve it, bearing in mind that the multi-start optimisation algorithm needs to keep track of the best optimisation solution found during all the runs.

Coevolutionary techniques (refer to Section 2.3.6) also apply more generally than only to EC. As examples, consider the use of particle swarm optimisation instead of EC for Blondie 24 (refer to Section 2.7) or the cooperative PSO (CPSO) [113] which applies a technique used for cooperative coevolutionary GAs [91] to PSOs. The cooperative optimisation algorithm implemented in CILib, which factors this common behaviour into a more generic algorithm, only caters for optimisation algorithms that cooperate by splitting the solution vector up into smaller components. This is accomplished by a problem *Adapter* (refer to Section 3.2.1), which calculates the fitness of a smaller component of the vector in the context of the other cooperating algorithms. The cooperating algorithms, or participants, are created by the cooperative optimisation algorithm using an *Abstract Factory* (refer to Section 3.1.1), so that the type of the participants can be specified externally. Any algorithm used as a participant must implement the `ParticipatingAlgorithm` interface, which provides a mechanism for the cooperative algorithm to access the individual parts of the solution worked on by each participant. Thus, by implementing the `ParticipatingAlgorithm` interface, any optimisation al-

University of Pretoria etd – Peer, E S (2005)

gorithm, PSO, GA or otherwise (including combinations of different algorithms at the same time), can participate in a coevolutionary strategy that splits up the solution vector amongst multiple cooperating algorithms. Other coevolutionary approaches, such as sharing solutions using blackboard or having competing populations, are currently being worked on by another contributor (refer to Section 6.3). Competing populations could conceivably be implemented relatively transparently using a `Fitness` *Adapter* (refer to Section 3.2.1), which evaluates fitness relative to individuals in other populations.

6.2.4 Particle Swarm Optimisers

This section explores CILib's PSO (refer to Section 2.4.1) architecture in more detail as a demonstration of the framework's support for the implementation of an optimisation algorithm. Implementations of other algorithms, optimisation or otherwise, were not provided by the author and as such are not discussed (refer to Section 6.3 for information about other contributions).

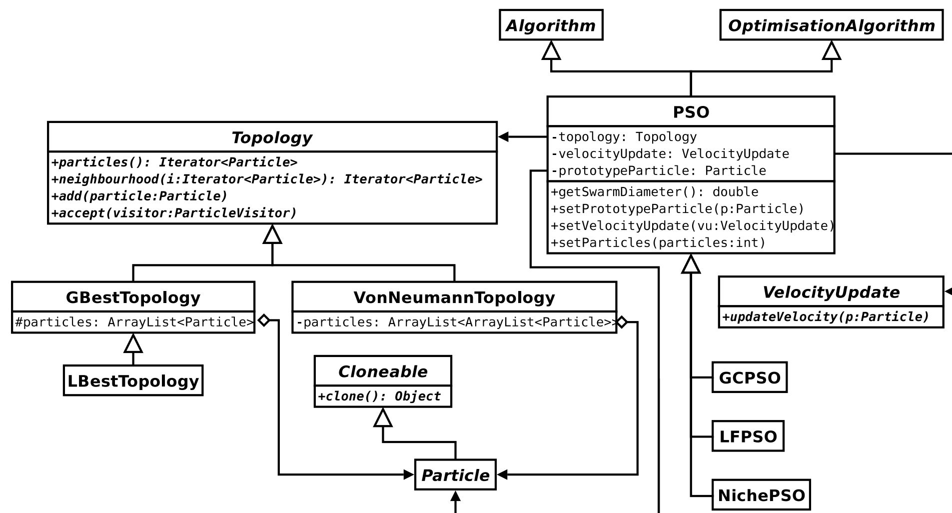

Figure 6.12: Overview of PSO Architecture

An overview of the PSO architecture implemented in CILib is provided in Figure 6.12. Particle swarms differ in terms of the neighbourhood topology of the particles and velocity update equation used to govern their trajectories. These two aspects are implemented as *Strategies* (refer to Section 3.3.4) which can be varied independently. Thus, any velocity update can be used in combination with any neighbourhood topology and *vice*

University of Pretoria etd – Peer, E S (2005)

versa.

The algorithm interface for `VelocityUpdate` is characterised by a single method, which is passed to the particle that it must update. The topology interface is more complex, exposing *Iterators* (refer to Section 3.3.2) based on the standard `java.util.Iterator` interface provided by the JFC. The PSO can use iterators to traverse all particles in the topology or only those particles within the neighbourhood of another particle, for which it must provide a pointer in the form of another iterator. Topologies in CILib are dynamic, particles can be added and removed at will. Removal of particles is achieved using the `remove()` method which is available through the iterator interface. Recently, *Visitor* (see Section 3.3.6) support was also added to topologies.

The fact that the LBest topology inherits from GBest requires some explanation, since GBest is a special case of LBest with the neighbourhood being equivalent to the entire swarm (refer to Section 2.4.1). To see why this is the case, consider that the LBest topology must implement a special *Iterator* with the ability to handle wrap-around in order to traverse the neighbourhood of any given particle. The GBest topology, however, does not require this specialised behaviour, since it can use an *Iterator* that simply traverses the whole array of particles for both the swarm and neighbourhood cases. Thus, LBest is the more specific case in terms of the implementation. The Von Neumann topology (refer to Section 2.4.1) is implemented as a two dimensional matrix, with a special neighbourhood *Iterator* that traverses the immediate particles in each compass direction.

Certain PSO algorithms require particles to store additional state or have special behaviour, an ideal opportunity to apply the *Decorator* pattern (Section 3.2.3), as illustrated in Figure 6.13. Particles may be configured differently depending on the particular type of PSO being used, but the `PSO` class is responsible for creating and initialising particles within the search space. For this reason, `Particle` implements the *Prototype* pattern (refer to Section 3.1.3), enabling the PSO to clone additional particles as necessary from a run time configured prototype. The particle positions are then initialised using the `DomainComponent` provided by the optimisation problem, by overriding the `performInitialisation()` hook provided by `Algorithm`. The inheritance depth weakness of the *Template Method* pattern (refer to Section 3.3.5) is clearly illustrated by this architecture. For example, both `PSO` and `GCPSO` may need to perform additional initialisation tasks, but only one can override the hook provided by the template method.

University of Pretoria etd – Peer, E S (2005)

Fortunately, in this case, the `GCPSO` class does not need to override it, but it is conceivable that some algorithm eventually will need to. In future, it may become necessary to store a list of initialisers in the base `Algorithm` class that must be executed in turn during initialisation, each initialiser performing the initialisation tasks specific to its algorithm.

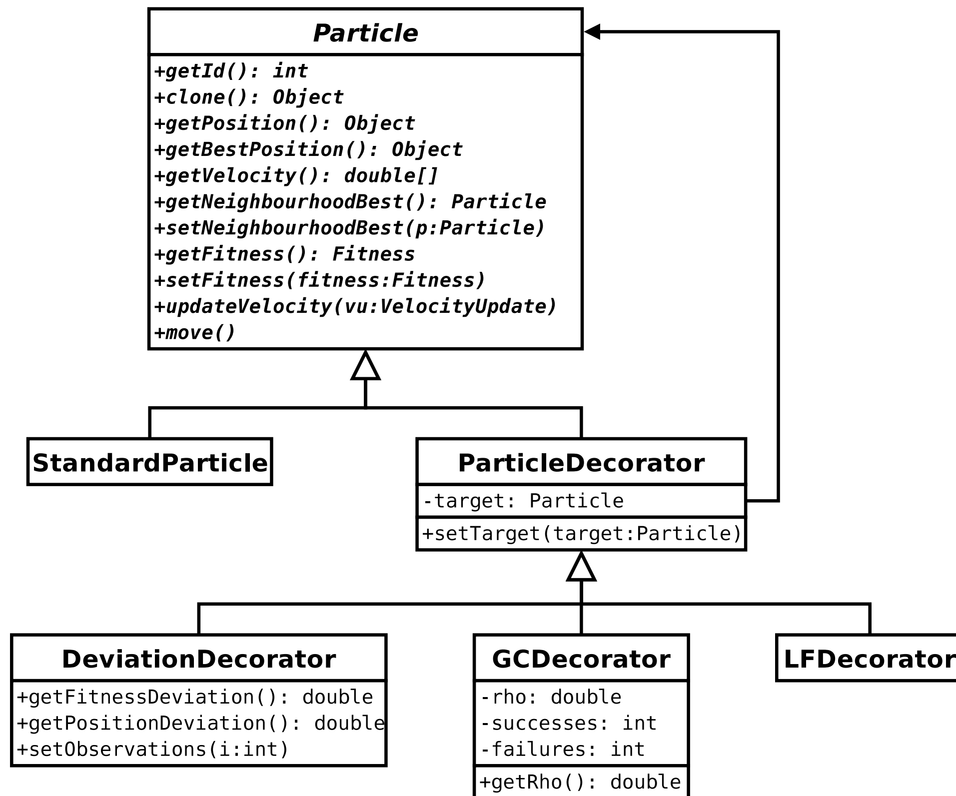

Figure 6.13: Particle Decorators

Figure 6.13 further illustrates the responsibilities of particles, each having to store its position, velocity, fitness and a reference to the best particle within its neighbourhood. In addition, each particle must be allocated a unique identifier, as a side effect of the *Decorator* pattern (refer to Section 3.2.3), so that they can be compared without regard to the dynamic nature of decorators that may be added and removed during the execution of an algorithm.

The deviation decorator, currently only used by the NichePSO [17], is used to track the standard deviations of the position and fitness of particles over time. This is an expensive operation. In terms of space, requiring a number of observations of position and fitness to be stored for each particle, and in terms of time, since these observations

University of Pretoria etd – Peer, E S (2005)

need to be updated every time a particle is moved. Thus, it makes sense to separate this functionality into a decorator that can be dynamically applied only when needed.

Both the GCPSO [114, 113] (refer to Section 2.7) and LFPSO (LeapFrog PSO, also refer to Section 2.7) algorithms implement a different velocity update equation for the neighbourhood best particles, each requiring additional state to be stored for these particles. The `GCDecorator` and `LFDecorator` decorators are used to store this additional state for their respective algorithms.

Specifically, the GCPSO velocity update performs a directed random search for the neighbourhood best particles. The step size of this search is controlled by a value, ρ (rho), which is dynamically updated based on the particle's past history. Particles which repeatedly improve their positions have their step size increased while particles that repeatedly fail to find better positions have their step size reduced.

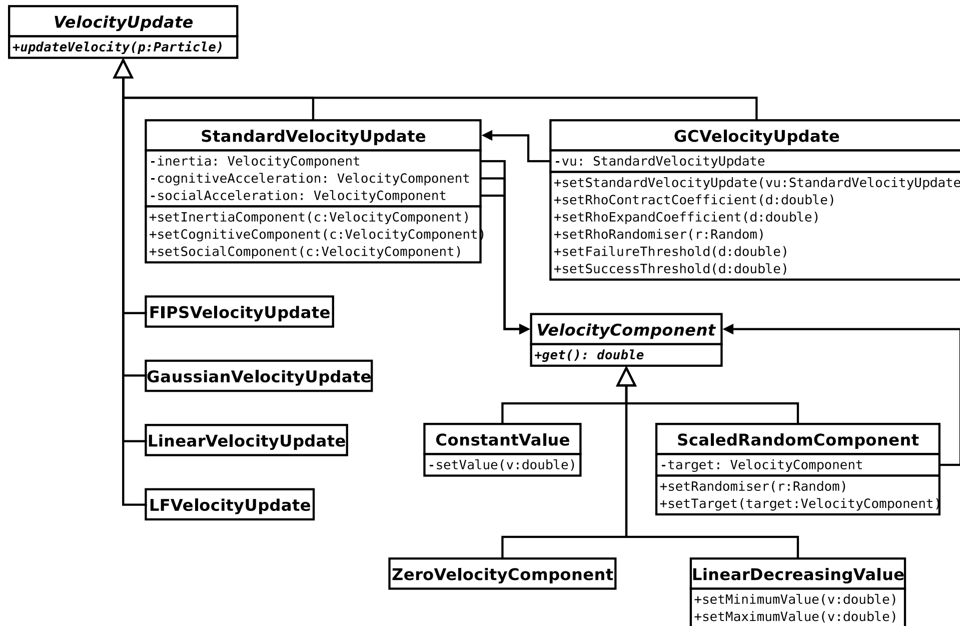

Figure 6.14: Velocity Updates

Figure 6.14 illustrates a number of velocity update *Strategies* (refer to Section 3.3.4), including the `GCVelocityUpdate` class, which implements the velocity update for the GCPSO. For non-neighbourhood best particles, it simply defers the velocity update to a standard velocity update instance. Thus, it only performs the directed random search for the best particle in each neighbourhood.

University of Pretoria etd – Peer, E S (2005)

The `StandardVelocityUpdate` class implements Equation 2.41, where the values for w, $c_1 r_1$ and $c_2 r_2$ are each delegated to a velocity component *Strategy* (refer to Section 3.3.4), giving a user a great deal of control over the velocity update. For instance, a linear decreasing inertia can be accomplished by simply replacing the default constant inertia component with a `LinearDecreasingValue`. By default, accelerations are implemented using a `ScaledRandomComponent` with a `ConstantValue` target, but they could be replaced with any velocity components, including a `ZeroVelocityComponent` to disable their influence, which is the equivalent of a `ConstantValue` with a value of zero.

The `LinearDecreasingValue` class is a good illustration of the usefulness of the global `Algorithm.get()` method described earlier, since it needs access to a value that scales linearly over the execution of the algorithm. A suitable value for this is available using the `getPercentageComplete()` method in `Algorithm`, however, it does not make sense to clutter the `VelocityUpdate` interface with this value, since it is not used by most velocity updates.

The remaining velocity update *Strategies* (refer to Section 3.3.4) implement a number of further PSO variants. The `LinearVelocityUpdate` class implements a variant suited for linearly constrained optimisation problems [87].

A bare bones PSO [62], which discards the notion of particle velocities and simply mutates their positions by sampling from a Gaussian distribution, is implemented by the `GaussianVelocityUpdate` class.

LFPSO is implemented by the `LFVelocityUpdate` class by following a similar approach to the `GCVelocityUpdate` class. The commonalities between the two approaches suggest that there may be merit in implementing a generic `OptimiserVelocityUpdate` which implements the `OptimisationProblem` interface, so as to replace the motion of neighbourhood best particles with the results of any `OptimisationAlgorithm` as suggested in Section 2.7.

The `FIPSVelocityUpdate` (for the Fully Informed Particle Swarm [78]) requires access to the entire neighbourhood of particles for the particle which is being updated. Since this was not foreseen when the `VelocityUpdate` or `Particle` interfaces were created, the current implementation is forced to make use of the global `Algorithm.get()` method. Unfortunately, it has to perform a linear search for the particle to obtain an iterator that can be used to access the neighbourhood, since particles do not know anything

University of Pretoria etd – Peer, E S (2005)

about the topology. This will be fixed in a later version of CILib, either by extending the `Particle` interface to make the entire neighbourhood accessible or by making particles aware of their position within a topology, by means of a *Decorator* (refer to Section 3.2.3), so that they can be located efficiently.

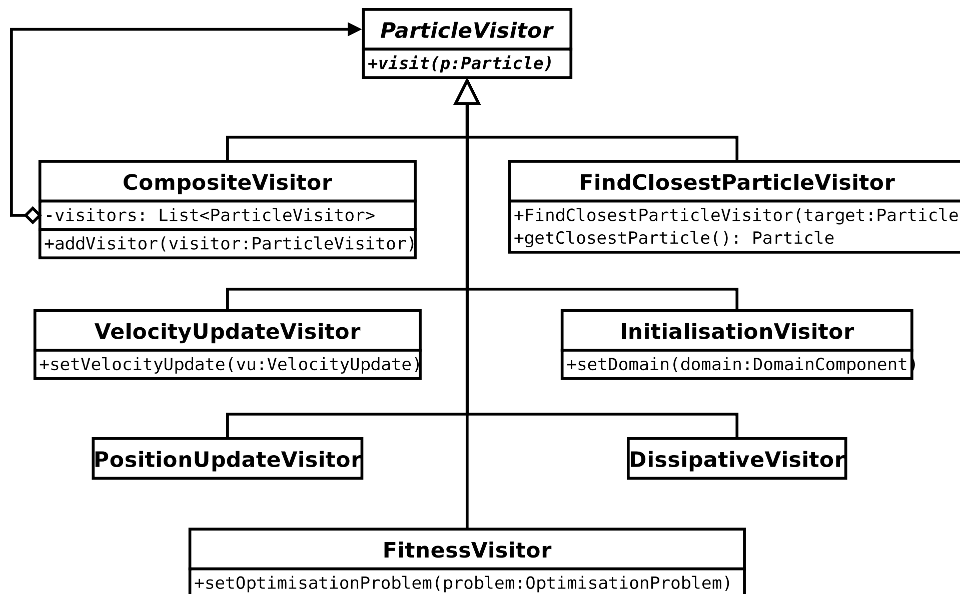

Figure 6.15: Particle Visitors

Most of the control logic for a PSO is currently in a monolithic `performItertion()` method. This is inflexible because that logic cannot be changed by simply composing different classes, but only by sub-classing the `PSO` class. Figure 6.15 represents the proposed next step in the evolution of the PSO code in CILib, the moving of parts of the internal PSO logic into external *Visitors* (refer to Section 3.3.6) which can be composed and reused in various ways. Of course, treating everything as visitors has the obvious danger that an inappropriate visitor will be used when something else is expected. Time will tell if this proposed design is a good idea or not.

The `VelocityUpdateVisitor` class is an *Adapter* (refer to Section 3.2.1) which makes any existing `VelocityUpdate` conform to the visitor interface. Perhaps velocity updates should have been implemented as visitors from the start, however, implementing velocity updates as visitors does restrict the `VelocityUpdate` interface to only accepting particles with no easy way to extend it. New velocity updates would not even need to implement the `VelocityUpdate` interface at all, but could implement `ParticleVisitor`

University of Pretoria etd – Peer, E S (2005)

directly. At this time, the global `Algorithm.get()` method appears to be a general enough mechanism for obtaining information not provided by the visitor interface.

The `PositionUpdateVisitor` class is analogous to the velocity update except that it moves the particle by altering its position instead of changing its velocity. This will have the side effect of cleaning up the `Particle` interface by removing the need for a separate `move()` method. In addition, the `GaussianVelocityUpdate` should rather be implemented as a position update, since it doesn't affect a particle's velocity at all.

The `InitialisationVisitor` class will be used to initialise particle positions based on a given domain. Delegating initialisation to a visitor enables a PSO to use an alternate means of initialisation, perhaps not even making use of the domain information, which is currently not possible.

The *Composite* (refer to Section 3.2.2) visitor is intended to allow multiple visitors to be used where only one is expected, with each visit method being called sequentially for each particle. For example, a position update visitor could be replaced by a composite containing both the position update and a dissipative visitor, which implements the logic required for the DPSO [122] (refer to Section 2.7).

Ultimately, subclasses of `PSO` will have to do far less work, perhaps as little as changing one of the visitors. This leads to the next improvement, an *Abstract Factory*, say `PSOComponentFactory`, with methods defined for creating particle, initialisation, velocity update and position update products. Thus, different particle swarm variants can be realised by merely supplying a different factory to the `PSO` class, negating the need for subclasses of `PSO` for every variant, only those that have radically different algorithms.

6.2.5 Stopping Conditions

Figure 6.16 shows some specific stopping conditions, which were discussed only generally in Section 6.2.3. Some conditions may be applied to any algorithm, while others are specific to certain types of algorithms.

For example, the maximum iterations condition can be applied to any algorithm, causing the algorithm to finish execution when the configured number of iterations has been reached. It makes use of the `getIterations()` method in `Algorithm` to determine when to fire. The condition for fitness evaluations, as another example, only applies to optimisation algorithms, which can be stopped when the objective function has been

University of Pretoria etd – Peer, E S (2005)

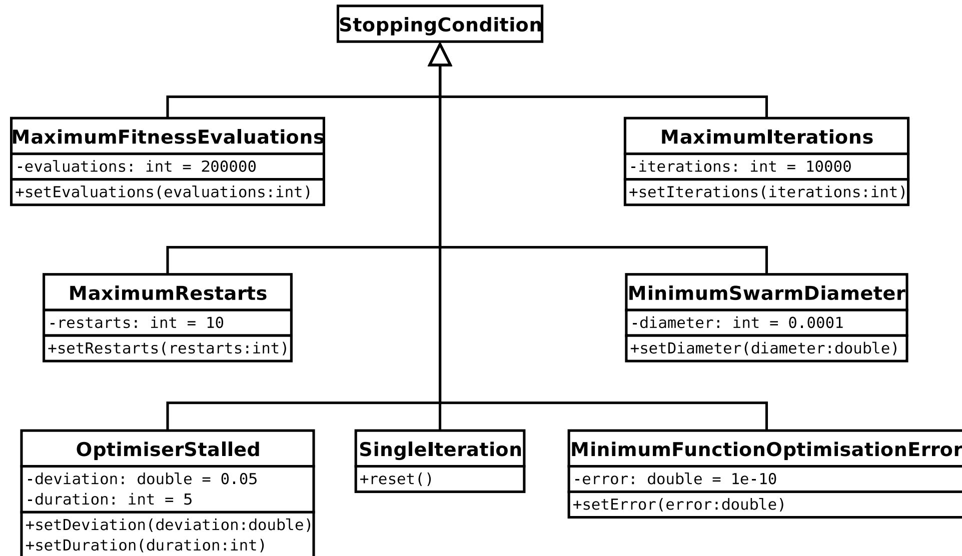

Figure 6.16: Stopping Conditions

tested a predetermined number of times. Implementations of conditions that apply to more specific algorithms must cast the algorithm they are passed into the type they expect it to be, throwing a `ClassCastException` if the user attempts to apply an unsuitable stopping condition to an algorithm. Table 6.1 lists the legal types of algorithm for each stopping condition.

The minimum swarm diameter condition fires when the average distance between particles and the global best drops below a threshold. Similarly, the minimum function optimisation error condition fires when the optimisation error, given by $|f(\mathbf{x}^*) - f(\mathbf{x})|$ for an objective function f with global extremum $\mathbf{x}^*$ and solution $\mathbf{x}$, drops below a threshold. Further, the `OptimiserStalled` condition fires when the standard deviation of an optimisation solution over a configurable number of iterations is less than a threshold. The single iteration condition is a special case condition, which fires after one iteration and does not permit execution again until it is reset. Finally, the maximum restarts condition fires whenever the number of restarts of a multi-start optimisation algorithm exceeds a threshold.

Wherever possible, an implementation should return a linearly increasing value in the range $[0, 1]$ for the `getPercentageComplete()` method (refer to Figure 6.10). For example, the maximum iterations condition returns the fraction $(\frac{\text{current iteration}}{\text{maximum iterations}})$. Con-

University of Pretoria etd – Peer, E S (2005)

Table 6.1: Legal Algorithms for Stopping Conditions

Stopping Condition	Legal Algorithms
`MaximumFitnessEvaluations`	Any optimisation algorithm
`MaximumIterations`	Any algorithm
`MaximumRestarts`	Only the multi-start optimisation algorithm
`MinimumSwarmDiameter`	Any particle swarm optimiser
`OptimiserStalled`	Any optimisation algorithm
`SingleIteration`	Any algorithm
`MinimumFunctionOptimisationError`	Only optimisation algorithms applied to function optimisation problems

ditions such as those based on the swarm diameter or optimisation error cannot make this guarantee, since they are dependent on the non-linear behaviour of the algorithm. However, they should still ensure to return a value in the correct range, even if it is only a binary 0 or 1 based on the output of `isFinished()`.

6.2.6 Measurements

Any platform designed for scientific research must be able to perform proper measurements during an experiment. The framework should enable a researcher to choose any property to measure and not dictate its type.

The CILib simulator, discussed in the next section, makes use of measurements to evaluate such properties during the execution of an algorithm. No restrictions are placed on the type of property, measurements return a `java.lang.Object`, with each measurement specifying its own domain, as a domain string which can be used to generate a domain description (refer to Section 6.2.1). Thus, irrespective of the property being measured, a measurement presents a uniform interface to a client, usually the simulator, as shown in Figure 6.17.

New measurements can be crafted to access any property in an algorithm's publicly accessible object reference graph. That is, measurements access the currently executing algorithm using the global `Algorithm.get()` method (refer to Section 6.2.3). Like stopping conditions, they need to cast the algorithm into the type they are expecting

University of Pretoria etd – Peer, E S (2005)

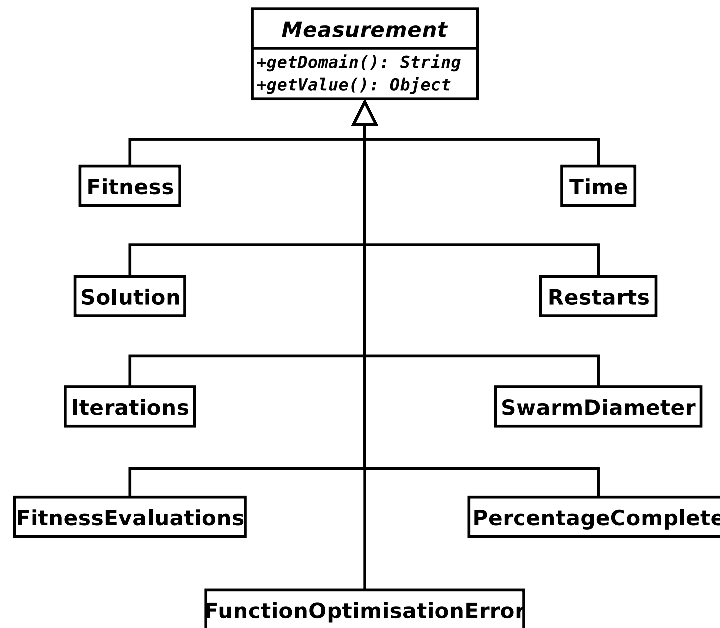

Figure 6.17: Measurements

and navigate to the property they are interested in. The implementation, however, may have difficulty locating properties if objects are composed in unexpected ways, particularly if they are deep in the graph. Using the global algorithm accessor enables a single measurement instance to be shared, provided they do not store any non-sharable state, since they do not maintain a reference to the algorithm (in future measurements may be implemented as *Singletons*, refer to Section 3.1.4).

Figure 6.17 shows a number of reusable measurements, so a researcher only needs to create new measurements if they are measuring something unusual. As was the case for stopping conditions, some measurements are specific to certain types of algorithms. Measurements have been defined for monitoring the solution and its fitness (for optimisation algorithms), number of fitness evaluations, current time, number of restarts (for the multi-start optimisation algorithm), number of iterations, percentage complete, swarm diameter (for particle swarms) and function optimisation error (for algorithms optimising functions). In fact, many of these are precisely the same properties which are monitored by stopping conditions.

Implementing stopping conditions using measurements has been considered as a means to reduce these parallel class hierarchies. That way, only two stopping con-

University of Pretoria etd – Peer, E S (2005)

ditions would be necessary, a maximum threshold condition which fires whenever the measured value exceeds a threshold and a complementary minimum version. For example, the maximum iterations stopping condition could be implemented using a maximum threshold condition and the `Iterations` measurement. The problem with this approach stems from the fact that measurements can have any type, numeric or otherwise. Thus, even for simple numeric types, which are handled very efficiently by stopping conditions, a measurement needs to perform an expensive object instantiation, creating a new `java.lang.Number`. Since measurements used by the simulator are typically only executed every k^{th} iteration for fairly large values of k, they can afford this inefficiency for the benefit of being able to deal with any type of property. Further, the measurement interface would require the stopping condition to perform an additional down cast before it can use the value. If measurements are to be used in stopping conditions, then the performance implications of the extra work performed after every iteration needs to be considered and properly bench-marked first.

Algorithm implementations are not aware of any clients which are performing measurements, since the client simply needs to declare itself as an *Observer* (refer to Section 3.3.3) and can execute any measurements, by calling their `getValue()` method, as it sees fit. Thus, all scientific measurement code is kept out of the implementations of algorithms, which do not need to concern themselves with how their behaviour will be monitored beyond providing sufficient public accessors for any interesting properties. This ensures that algorithm implementations do not become polluted with measurement code, which may not required in all circumstances. For example, if an algorithm is implemented in a non-research context, as part of another application.

6.2.7 Simulator

The simulator is CILib's mechanism for configuring and executing experiments. The heart of the simulator is an XML object factory, which enables algorithms, problems and measurements to be constructed, configured and composed at run time according to a simple XML document. The `XMLObjectFactory` class, which accepts a DOM element (refer to Section 5.1.3) describing its configuration, can be used over and over again to construct objects with the same configuration. Further, it can be trivially *Adapted* (refer to Section 3.2.1) to be the implementation for any *Abstract Factory* (refer to

University of Pretoria etd – Peer, E S (2005)

Section 3.1.1) interface, as shown in Figure 6.18.

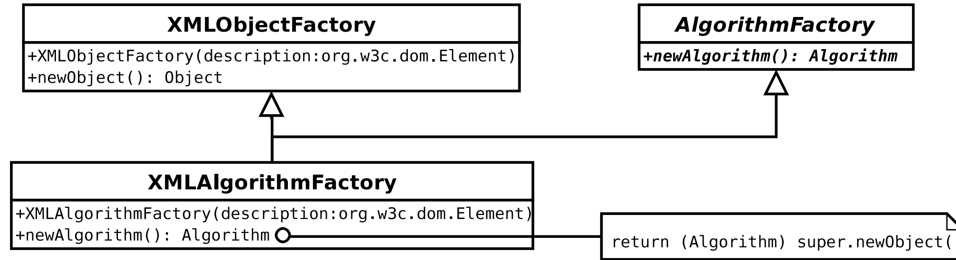

Figure 6.18: XML Object Factory

Figure 6.19 is an example configuration for the CILib simulator, using a standard PSO with a linear decreasing inertia component to find the minimum of the spherical function on its default domain of "R(-100,100)^30", given by:

$$f(\mathbf{x}) = \sum_{i=1}^{30} x_i^2, \text{ with } x_i \in \{\mathbb{R} \mid -100 \leq x_i \leq 100\} \tag{6.1}$$

while measuring the number of iterations and function optimisation error, by default every 100 iterations, and outputting the results to a file named "inertia.txt". By default, the simulator repeats the experiment 30 times, actually it runs them in parallel threads, outputting all the results to the same file, where they can be later analysed.

The simulation engine searches the document for `<simulation/>` elements, each containing the configuration for running a single algorithm on a given problem while measuring certain properties. All objects must have a default constructor and should provide sensible defaults for all of their properties. Any publicly accessible property can be set by specifying a corresponding tag name in the configuration. The document's legal tag names are dictated by the properties available in the source code at run time, using the Java reflection API, so it is impossible to construct a rigid schema that describes valid simulator documents (refer to Section 5.1.2).

For example, because the `PSO` exposes a public velocity update property, via the `setVelocityUpdate(VelocityUpdate vu)` method, it can be set using a tag corresponding to that property name. A `class` attribute specifies the name of a class that should be instantiated by the factory and passed to the property specified in its element. Class names are specified relative to the `net.sourceforge.cilib` package, however, fully qualified class names are also permitted.

University of Pretoria etd – Peer, E S (2005)


```
<simulator>
  <simulation>
    <algorithm class="PSO.PSO">
      <addStoppingCondition class="StoppingCondition.MaximumIterations"/>
      <velocityUpdate class="PSO.StandardVelocityUpdate">
        <inertiaComponent class="PSO.LinearDecreasingValue">
          <minimumValue>0.25</minimumValue>
          <maximumValue>1.0</maximumValue>
        </inertiaComponent>
      </velocityUpdate>
    </algorithm>
    <problem class="Problem.FunctionMinimisationProblem">
      <function class="Functions.Spherical"/>
    </problem>
    <measurements class="Simulator.MeasurementSuite">
      <file>inertia.txt</file>
      <addMeasurement class="Measurement.Iterations"/>
      <addMeasurement class="Measurement.FunctionOptimisationError"/>
    </measurements>
  </simulation>
</simulator>
```

Figure 6.19: Simple Simulator Configuration

Strings and primitive typed properties can be set by simply enclosing their value within the element body. Thus, in the sample, the minimum and maximum values for an instance of LinearDecreasingValue are set to 0.25 and 1.0 respectively. Similarly, the name of the file into which the measurement suite will output its results is specified within a <file/> element, which corresponds to the setFile(String fileName) method in the MeasurementSuite class.

Arbitrary methods can be called by using the method name as the tag name, the XML object factory simply provides a short hand for properties (indicated by a method with the prefix “set”). Thus, multiple stopping conditions and measurements can be added

University of Pretoria etd – Peer, E S (2005)

using the `addStoppingCondition()` method in `Algorithm` and the `addMeasurement()` method in `MeasurementSuite` respectively. Methods with an arbitrary number of parameters are also supported by nesting each parameter as a separate element (their names do not matter) within the method element in the order they appear in the method signature.

Figure 6.20, in turn, illustrates another slightly more complex configuration file. This example demonstrates how portions of the document can be reused by making use of ID references (refer to Section 5.1.1). Typically, more descriptive identifiers than "A", "B", "M" and "S" would be used, they were shorted here purely for formatting reasons. Note that the fact that multiple algorithms and simulations are specified within `<algorithms/>` and `<simulations/>` elements is immaterial. The simulator merely searches for simulation elements and follows any identity links to their targets, irrespective of where they are defined in the document. Further, the sample demonstrates two short hand ways to set properties. Primitive and string valued properties can be specified directly as attributes in the parent element instead of nesting them as separate elements. Alternatively, they can be specified using the `value` attribute of their own property tags instead of placing the value in the body of the element. Properties in reused portions of the document can be overridden where they are referenced. For example, the same measurement suite configuration is used to output to two different file names. In addition, the measurement suite has two additional properties: i) the number of repetitions of the experiment, or samples; and ii) the resolution, which specifies how often results are written to file. Finally, the cooperative optimisation algorithm uses the `XMLAlgorithmFactory` *Adapter* demonstrated in Figure 6.18. An XML algorithm factory expects its configuration to be specified in a nested `<algorithm/>` element and from there on down functions in exactly the same manner as the XML object factory.

Further examples of configuration files are distributed with the CILib source code. Additional examples which demonstrate all the features of the XML object factory are also available for download from the CILib project page.

6.3 Collaborations

To date, CILib has relatively mature implementations of particle swarm and ant colony frameworks. An early EC framework which is in need of some refactoring, to take into account improvements to the core framework since it was contributed, has also been

University of Pretoria etd – Peer, E S (2005)


```
<simulator>
  <algorithms>
    <algorithm id="A" class="Algorithm.CoOperativeOptimisationAlgorithm">
      <algorithmFactory class="XML.XMLAlgorithmFactory">
        <algorithm idref="B"/>
      </algorithmFactory>
      <participants value="10"/>
    </algorithm>
    <algorithm id="B" class="PSO.PSO">
      <topology class="PSO.VonNeumannTopology"/>
      <addStoppingCondition class="StoppingCondition.MaximumIterations"/>
    </algorithm>
  </algorithms>
  <problem id="S" class="Problem.FunctionMinimisationProblem">
    <function class="Functions.Spherical" domain="R(-50,50)^100"/>
  </problem>
  <measurements id="M" class="Simulator.MeasurementSuite" samples="50">
    <addMeasurement class="Measurement.FitnessEvaluations"/>
    <addMeasurement class="Measurement.FunctionOptimisationError"/>
  </measurements>
  <simulations>
    <simulation>
      <algorithm idref="A"/>
      <problem idref="S"/>
      <measurements idref="M" file="data/cpso.txt"/>
    </simulation>
    <simulation>
      <algorithm idref="B"/>
      <problem idref="S"/>
      <measurements idref="M" file="data/pso.txt"/>
    </simulation>
  </simulations>
</simulator>
```

Figure 6.20: More Complex Simulator Configuration

University of Pretoria etd – Peer, E S (2005)

implemented. In addition, several benchmark functions have been defined for testing optimisation algorithms. Neural network and coevolutionary game (based on Blondie 24, refer to Section 2.7) frameworks are currently being worked on by other students as part of their studies. No significant contributions have been received from parties outside of the University of Pretoria, but it has not yet been very widely advertised either. Further, nothing has been implemented in the fuzzy systems paradigm, mainly because nobody in the CIRG@UP is currently focusing on research in that field. The framework has been offered as a platform for implementing assignments for postgraduate courses and has received a fair amount of interest from those students. Table 6.2 lists the names of significant contributors[2], crediting them with the parts of CILib that they have been primarily responsible for.

Table 6.2: CILib Contributors

Names	**Contributions**
Barla-Szabo, D.	LFPSO
Engelbrecht, A. P.	Benchmark Functions, PSO Additions
Kroon, J.	Nonlinear Mapping Problems [71], Domain Visitor
Naicker, C.	NichePSO, Benchmark Functions, EC Framework
Pampara, G.	Ant System Framework, Containers
Papaconstantis, E.	Coevolutionary Games Framework
Peer, E. S.	CILib Core, Benchmark Functions, PSO Framework
Van der Stockt, S.	Neural Network Framework
Van Niekerk, F	Cooperative Algorithms

6.4 Limitations

CILib successfully meets many of the goals identified at the start of this chapter. The use of design patterns and the XML object factory provide for a very flexible framework, where classes can be composed at will to produce any permutation permitted by the design. Experimentation is facilitated by the simulator, which provides for making

[2]http://sourceforge.net/project/memberlist.php?group_id=72233

University of Pretoria etd – Peer, E S (2005)

measurements of any interesting property during the execution of an algorithm. The domain system presented in Section 6.2.1 ensures that algorithms can use efficient types wherever possible, trading off the OO neatness of a polymorphic type system in favour of better performance, with a view to make the design cleaner as better compilers become available. A clean separation between algorithms, problems and measurements enables algorithms to be separated out and used in real world applications, not only within the research framework. In addition, the open source development model and having multiple people working on the same code base has forced improvements on the design, to make it accommodate their needs, and contributed towards numerous bug fixes.

That said, the CILib design is by no means perfect and continuous refactoring will be necessary as the framework grows to support more. Further, although CILib has generated numerous collaborative opportunities internally, it has yet to prove itself to a wider audience. A lack of documentation, which this dissertation hopes to alleviate, has also contributed to a steep learning curve for those wishing to use the software. Also, it has been difficult to convince some contributors of the benefits of unit testing (refer to Section 5.5), particularly when the correct outcomes for stochastic processes are not known *a priori*. Thus, there is lack of test cases for much of the implementation. Already, test cases for certain benchmark functions have proven their worth, where an error, which was discovered by a unit test, would have resulted in incorrect simulation results.

The following is a non-exhaustive list of some more specific limitations that have been identified:

- **Expensive fitness evaluations:** To accommodate discrete optimisation problems in CILib, the return value of benchmark functions was altered from a primitive `double` value to a `java.lang.Object` type. This means that every evaluation of an objective function typically results in an new instance of `java.lang.Number` being created. In addition to the extra object creation, the use of objective functions in tight loops places a severe strain on the garbage collector, since large amounts of memory will be consumed and need to be reclaimed. The mutable polymorphic type system presented in Section 6.2.1 may provide an efficient solution for this problem, since the same object used in the previous evaluation of an individual's position during a previous iteration can be reused by passing it as a reference to an objective function.

University of Pretoria etd – Peer, E S (2005)

- **Loose configuration file format:** The configuration file format was designed with hand crafting the document in mind. So, instead of having tags with consistent names and attributes with values corresponding to property names, it was decided to shorten the format by having the element name itself refer to the property name. In retrospect, it would have been better to follow an approach that can be validated against a static schema, which would have made writing the GUI tools discussed in the next chapter simpler. For example, instead of implementing a custom schema validator that needs to introspect the source code to perform its work, it would have been possible to make use of the XMLBeans[3] framework, capable of mapping an XML document directly onto Java objects.

- **Scalability:** The simulator spawns a new thread of execution for each sample. The motivation for this was that Unix tools such as GNU `awk`[4], which can be used for processing results, operate most conveniently on data presented in columns for each measurement of each sample. Since text files are most naturally written in rows, executing experiments sequentially would mean that information for subsequent columns would not be available. By running the experiments in parallel, it was hoped that all the information required for a given row would become available at roughly the same time, avoiding the need to buffer a large quantity of measurement results, which can quickly grow to hundreds of megabytes in size. Unfortunately, because of this, the simulator can not scale to large numbers of samples. The extra scheduling overhead and larger footprint required for keeping multiple executing algorithms in memory at the same time can become prohibitive. The implicit assumption that this memory overhead would be less than buffering the results also does not always hold, particularly if one thread of execution becomes starved of CPU time, in which case the buffering overhead is incurred anyway. The following chapter presents a solution to this problem, by storing the results in a structured database, as well as being able to scale experiments up to a cluster of workstations. Alternatively, the simulator could trivially be changed to write results in rows, requiring post processing for tools like `awk`, or results could be temporarily buffered to disk so that simulations can be run sequentially.

[3] http://xmlbeans.apache.org/

[4] http://www.gnu.org/software/gawk/gawk.html

University of Pretoria etd – Peer, E S (2005)

In spite of these and other limitations, CILib is already useful in its current state and has the potential to become an important collaborative resource in the future.