Extending the 'Open-Closed Principle' to Automated Algorithm Configuration

Jerry Swan jerry.swan@york.ac.uk
Dept. of Computer Science, University of York, UK.

Steven Adriænsen steven.adriaensen@vub.ac.be
Vrije Universiteit Brussel, Belgium.

Adam D. Barwell adb23@st-andrews.ac.uk
University of St Andrews, Scotland.

Kevin Hammond kevin@kevinhammond.net
University of St Andrews, Scotland.

David R. White david.r.white@ucl.ac.uk
Department of Computer Science, University College London, UK.

Abstract

Metaheuristics are an effective and diverse class of optimization algorithms: a means of obtaining solutions of acceptable quality for otherwise intractable problems. The selection, construction, and configuration of a metaheuristic for a given problem has historically been a manually intensive process based on experience, experimentation, and reasoning by metaphor. More recently, there has been interest. In this paper, we identify shared state as the inhibitor of greater automation in algorithm configuration. To solve this problem, we introduce the Automated Open Closed Principle (AOCP), which lists design requirements supporting reuse of algorithm frameworks and automated assembly of of algorithms from an extensible palette of components. We demonstrate how the AOCP enables a greater degree of automation than previously possible via an example implementation.

Keywords

Automated Design of Algorithms, Automatic Programming, Programming by Optimization, Metaheuristics, Functional Programming, Ant Programming, Search Based Software Engineering, Systems Self Assembly.

1 Introduction

Metaheuristics define families of algorithms for obtaining acceptable solutions to hard optimization problems. Each metaheuristic defines a general framework for optimization that must be tailored to individual applications in order to exploit domain-specific information or to incorporate novel search mechanisms. A practitioner must select an appropriate metaheuristic, customize it in terms of its constituent components — such as functions implementing perturbation operators or termination criteria — and specify many parameter settings. For example, they may select a suitable heuristic for recombining two solutions to produce another solution, or set the rate of crossover in a Genetic Algorithm (Luke, 2013). The practitioner is thus faced with a meta-optimization problem in the design space of metaheuristics.

To solve this manual design problem, practitioners have previously relied upon anecdotal evidence, trial-and-error methods such as 'one factor at a time' parameter tuning and 'reasoning by metaphor'. This has led to a plethora of algorithms, components, and parameters,

divided into separate research silos. As new extensions are made to existing algorithms, the design space continues to expand; concepts from one family of metaheuristics are reinvented elsewhere (Weyland, 2010). This manual process of development is fundamentally *ad hoc*, does not promote reusability of components or separation of concerns, and does not scale.

The development of flexible frameworks that facilitate the reusability of components, the incremental design of metaheuristics, and ultimately the automation of this manual design process, is thus an important challenge for metaheuristics research. Whilst related work such as 'Programming by Optimization' (Hoos, 2012) and *configurators* (Hutter et al., 2009; Ansótegui et al., 2009; Hutter et al., 2011a; López-Ibáñez et al., 2016) have supported the optimization of algorithm parameters and even the exploration of restricted combinatorial design spaces (KhudaBukhsh et al., 2009; López-Ibáñez and Stützle, 2012; Mascia et al., 2014; Adriaensen et al., 2014; Bezerra et al., 2016), they are limited in their generality by the prevalence of *state dependencies* between components; new components cannot be added without modifying the grammar that is provided to a configurator and/or changing the call graph between components. It is these state dependencies that prevent a clean separation of concerns in metaheuristic design.

In this paper, we identify state dependencies as the major obstacle to the incremental assembly of metaheuristics, and describe *the Automated Open Closed Principle*, a solution to this problem which uses principled state threading to manage state dependencies. We describe an implementation for an example metaheuristic application. Our main contributions are as follows:

1. We identify *state dependencies* as the main obstacle to composition and reuse of components, and the eventual automation of metaheuristic design.

2. We propose the *Automated Open Closed Principle* (AOCP) to overcome this obstacle.

3. We provide an example implementation using functional programming constructs that demonstrates the efficacy of the AOCP.

4. We discuss the other advantages of following the AOCP in automated metaheuristic design.

2 State Dependencies between Metaheuristic Components

Given the role of metaheuristics as generic problem solving strategies, there is a natural desire to parameterize metaheuristic frameworks by a relatively small number of components such as perturbation operators, selection methods, and termination criteria. As discussed in detail in subsequent sections, a genuinely clean separation of concerns (both between framework and components and between components themselves) is difficult to obtain because of *state dependencies*. In particular, the lack of a principled means of handling such dependencies has inhibited progress towards automated assembly, particularly at larger scales. This problem is particularly acute when considering the addition of new components to an existing system, whereby the original system must be modified to allow for their inclusion. This modification violates the 'Open-Closed principle' (OCP) of framework design (Larman, 2001): an implementation should be closed to *modification* but nonetheless open to *extension* via custom components. Crucially, it inhibits reusability of components and forestalls the possibility of the automated exploration of the design space.

As a concrete example of this difficulty, Listing 1 gives a simple local search framework that allows for three design decisions, viz. the choice of perturbation, acceptance and termination conditions. Each specific triple of components (*perturb*, *accept*, *isFinished*) used to configure

```
def localSearch(incumbent: Sol,
      perturb: Sol => Sol,
      accept:  (Sol,Sol)  => Sol,
      isFinished: Sol  => Boolean): Sol {

    while( not finished( incumbent ) )
        incumbent = accept( incumbent, perturb( incumbent ) )

    return incumbent;
}
```

Listing 1: Local Search framework parameterized by design choices

the framework corresponds to a specific local search algorithm. This allows us to concisely specify a large design space as the Cartesian product of alternative design decisions, and also exposes the relationship between different designs in terms of decisions that they have in common. We can further generalize by parameterizing each component in turn by its own design choices. This design space could be automatically explored using a configurator approach; configurators typically require a grammar to be provided in a configuration file, which would necessitate a second specification of this design space.

To permit the substitution of different choices for each component, they must conform to a well-defined interface. In practice, this usually implies a 'one-size fits all' function signature: in our example, perturbation is assumed to have type $Sol \rightarrow Sol$. However, suppose we now wish to incorporate a further heuristic that requires information about the search trajectory, for example through a tabu list of solutions that prevents the resampling of those previously encountered (Glover and Laguna, 1997). Not only must the configurator grammar be extended to include the possibility of different tabu mechanisms, but we must also change the implementation of local search to keep track of the trajectory. This clearly violates the OCP.

In Listing 2, we give an updated version in which a list of previous incumbent solutions is denoted by $[Sol]$.

```
def localSearch( current: Sol,
    perturb: (Sol,[Sol]) => Sol,
    accept: (Sol,Sol,[Sol]) => (Sol, [Sol]),
    finished: (Sol,[Sol]) => Boolean )
    : Sol {

    history = []
    while( not finished( current, history ) )
        (current,history) = accept(current,perturb(current,history),history);

    return current;
}
```

Listing 2: Explicit incorporation of solution history

The modified implementation now supports solution-based tabu mechanisms, but we will incur further violations of the OCP if we wish to incorporate components with new state dependencies, for example:

- Metropolis-Hastings acceptance, which requires the delta of fitness values (Kirkpatrick et al., 1983).
- Perturbation with a tabu *operator* (e.g. permutation indices) (Glover and Laguna, 1997).

- 'Simulated Annealing with reheating' (Luke, 2013), in which the acceptance criterion and the reheating procedure need access to the value of *temperature*.

Explicitly incorporating *all* of these features, merely because *some* component may require it, is computationally expensive and does not scale. More importantly, it simply cannot be anticipated in advance what information will be required by some component that is yet to be devised. In order to accommodate this, we need a generic way of providing *extrinsic state*, i.e. (potentially shared) state that is maintained across component invocations, but which does not appear in the signature of component interfaces.

In the case of most existing metaheuristic implementations, extrinsic state is represented in an essentially *ad hoc* manner by passing references to shared state between components (e.g. via non-local variables). This is problematic for automated assembly, as explicit representation of these shared variables must be propagated through the call graph of the implementation, connecting components to the objects that they use. Furthermore, any variables that are declared in an *ad hoc* way throughout the implementation must be gathered manually in order to present them to a configuration tool.

We now discuss our approach to solving this problem.

3 The Automated Open-Closed Principle

The 'Open-Closed Principle' is usually ascribed to (Meyer, 1988) and states that frameworks should be 'open' in the sense of being extensible via the addition of new components, but 'closed' in that they do not require framework modification to achieve this.

The principle is typically expressed in terms of *substitutability*: the components are required to conform to established behaviors known *a priori* to the framework. The notion of component substitutability that suffices for conventional software development is that of 'observable behavior', typically defined by an interface that a component conforms to, along with 'contracts' that specify pre- and post- conditions on component functionality (Liskov, 1987). A simple example of a component contract in metaheuristics is that the return value of *accept:* $Sol \times Sol \rightarrow Sol$ should be equal to one of its arguments.

However, since new components may introduce new state dependencies, the OCP is insufficient to support flexible incremental assembly of metaheuristics from component parts. For example, an annealing mechanism that is used as a component in a local search algorithm may have a dependency on a temperature variable. In our example in Section 2, such the introduction of new state dependencies requires manual propagation of the additional state through the call graph. We therefore propose an extension of the OCP for automated assembly to yield the 'Automated Open-Closed Principle' (AOCP):

> *"A software system should be open to automated combinatorial assembly over an extensible palette of components without requiring human intervention."*

In order for a system's components to be highly reusable and substitutable, to facilitate incremental design, and thus also to be amenable to automated assembly without human intervention, a system must obey the AOCP. To conform to the AOCP, a system should exhibit the following properties:

1. **Open**: Present a framework parameterized by component definitions, together with an open-ended palette of such components.

2. **Closed**: Allow the addition of new components to the palette, without requiring human intervention to modify pre-existing components.

3. **Introspectable Design**: Component signatures and dependencies on shared state are available in machine-readable form.

4. **Observable**: State transitions can be made fully observable to the assembly process.

The third and fourth properties ensure that the assembly process can correctly determine the behavior of the components in order to measure their contribution to the performance of the overall framework, i.e. to perform credit assignment over the assembled components. Section 5 gives a working example of such a system.

To researchers who are accustomed to writing their desired metaheuristic 'from scratch', the proposed approach might initially appear to offer few benefits. However, it is most meaningful to consider the AOCP in the wider motivating context: that of large scale, potentially automated, assembly of metaheuristics for scientific discovery and reproducibility (Swan et al., 2015). The guiding notion is of a shared, community-wide repository of frameworks and components that can be freely combined to gain better insight into the correlation between problem features and solution mechanisms (Smith-Miles et al., 2014).

Without the proposed approach, exploring combinations of components at such a large scale would simply be impossible, due to the amount of manual intervention that would be required to manage state dependencies. An asymptotic cost of the associated manual labor can be obtained as follows: given a graph $(G = V, E)$ for a design with $n = |V|$ vertices (where vertices equivalently represent either grammar elements or subroutine invocations in the call graph), then for all pairs of components which are coupled via shared state, all paths between those components need to be manually updated to propagate that state. For each pair $u, v \in V$, then it is well-known[1], that the number of simple paths between u and v is given by $(n-2)!|E|$. Since there are up to n^2 such pairs of vertices, the number of updates is given by $\mathcal{O}(n^2(n-2)!|E|)$.

4 Implementing the AOCP

We now outline an implementation of the ACOP, exploiting well-established programming constructs that directly support the ACOP requirements. Whilst this implementation uses functional programming techniques, since such techniques lend themselves well to clearly defining and managing state dependencies, the mechanisms that we describe can alternatively be realised in an imperative manner. However, this requires work on the part of the framework implementer to achieve something that can already be achieved automatically by the means that we describe below. We are not suggesting that metaheuristics practitioners should become skilled functional programmers, rather the intention is that repositories of frameworks and components can be created using these principles and can subsequently be re-combined by practitioners, possibly via automation.

Our implementation uses *state threading* as an alternative to the *ad hoc* approaches to extrinsic state taken by most contemporary metaheuristic implementations. State threading requires an explicit input/output parameter for the extrinsic state, which is then threaded through the search process. For example, in Listing 2, the extrinsic state for `localSearch` comprises the list of previous incumbent solutions, `history`. Instead of defining `history` as a shared or global variable that any of the three components access and update directly, `perturb`, `accept`, and `finished` each take an additional parameter that is used to *thread* the extrinsic state through the search algorithm.

Explicit representation of state facilitates the composition and reuse of components, possibly automatically, since all dependencies for that component are clearly described and

[1] strictly, for $n \geq 4$

```
type Perturb[Sol]    = Sol => State[Int,Sol]
type Accept[Sol]     = (Sol,Sol) => State[Int,Sol]
type IsFinished[Sol] = Sol => State[Int,Boolean]

class LocalSearch[Sol] {

  def apply[Sol](incumbent : Sol,
    perturb : Perturb[Sol],
    accept : Accept[Sol],
    finished : IsFinished[Sol]) : State[Int,Sol] = {

    def until(s : Sol) : State[Int,Sol] = {
      for {
        // Increments loop counter in the state
        i  <- State.modify[Int](i => i+1)
        perturbed <- perturb(s)
        accepted <- accept(s, perturbed)
        c  <- finished(accepted)
        result <-
          if (c)
            State.pure[Int,Sol](accepted)
          else
            until(accepted)
      } yield result
    }

    for {
    // Initialises the state
      _ <- State.set[Int](0)
      result <- until(incumbent)
    } yield result
  }
}
```

Listing 3: Local Search in Scala with State monad

can be easily reasoned about mechanically. Explicit representation of state also allows a component to dynamically inspect the environment that is passed to it. Such checks might be performed prior to the application of a perturbation, for example, to automatically adjust parameters and so improve its effectiveness. Explicit state and modularity enables static safety checks whilst also providing the means to concisely develop self-adjusting components, without creating an overly tight coupling.

Despite the advantages of this approach, manually threading the state through the algorithm, as in Listing 2, can reduce the clarity of the code by introducing additional boilerplate. It is also error prone, since the programmer must ensure that the correct value of the state is passed to the correct stage of the algorithm. It is instead desirable to use some mechanism that *implicitly* threads the state, but does so in a *well-defined* and *consistent* manner.

The explicit representation of shared state in this manner is a well-known design pattern in the functional programming community, where it is expressed as the *State monad*. Whilst a formal definition is highly technical (Mac Lane, 1969; Moggi, 1991), it suffices here to consider monads to be a principled means of sequencing computations whilst abstracting over possible effects, such as state manipulation. Monads are a functional design pattern that follow well-defined laws to give strong formal guarantees to the programs that use them (Wadler, 1995). They provide a general abstraction mechanism that can be used to explicitly represent state; to clearly separate different kinds of state manipulation or other effects (such as I/O or exception handling), and to improve the reproducibility of behaviours on a per-component basis. A

```
type Perturb[Env,Sol]    = Sol => State[Env,Sol]
type Accept[Env,Sol]     = (Sol,Sol) => State[Env,Sol]
type IsFinished[Env,Sol] = Sol => State[Env,Boolean]

class LocalSearch[Env,Sol] {

  def apply[Sol](incumbent : Sol,
                 perturb : Perturb[Env,Sol],
                 accept : Accept[Env,Sol],
                 finished : IsFinished[Env,Sol],
                 iter : Lens[Env,Int]) : State[Env,Sol] = {
    def until(s : Sol) : State[Env,Sol] = {
      for {
        _ <- iter %= {i => i+1} // Increments loop counter in environment
        perturbed <- perturb(s)
        accepted <- accept(s, perturbed)
        c  <- finished(accepted)
        result <- if (c) {
                    State.pure[Env,Sol](accepted)
                  } else {
                    until(accepted)
                  }
      } yield result
    }

    for {
      // Initialises environment
      _ <- State.modify[Env](env => iter.set(0)(env));

      result <- until(incumbent)
    } yield result
  }
}
```

Listing 4: Local Search in Scala with State monad and lenses

statically-checked type-system can be used to separate stateless from stateful operations and to provide information about which state components are being manipulated.

For our purposes, we can consider a monad to comprise a pair of functions: *pure* (also called 'return' in Haskell) and *flatMap* (also known as 'bind', and denoted by >>= in Haskell):

$$pure_F : A \rightarrow F[A]$$
$$flatMap_F : F[A] \times (A \rightarrow F[B]) \rightarrow F[B]$$

F is an example of a *higher-kinded type*, which can be considered as a generic container or context for objects of type T, denoted $F[T]$. Common examples of such contexts include pure collection-types such as Lists or Trees, but also include *stateful and parallel computations*. If we think of $F[T]$ as imposing a computational context on top of some type T, then the *pure* function can be seen as a way of imposing this context upon a raw value of that type. Given a context $F[A]$ and some function $m : A \rightarrow F[B]$ for turning an object of type A into another context $F[B]$ over type B, then the *flatMap* operation first extracts a value of type A from $F[A]$ and then uses m to generate the next context in the sequence, based on this value. The type F captures precisely the class of effect that is encapsulated by the monad.

Languages such as Haskell and Scala provide syntactic sugar for monads. In particular, they allow a sequence of *flatMap* operations to be chained together using syntax that looks like a traditional *for* loop. This is illustrated in Listing 3, a re-formulation of our local search example in Listing 1 that uses the State monad. The state, in this case an integer representing

```
class TabuAccept[Env,Sol](history : Lens[Env,[Sol]]) {
  def apply(incumbent : Sol, perturbed : Sol) : State[Env,Sol] = ...
}
```

Listing 5: Acceptance component in Scala for Local Search with solution history

the number of iterations, is implicitly threaded through each stage of the computation. This can be extended to yield a re-formulation of Listing 2, thus incorporating history by changing the type of the state from `Int` to `(Int, [Sol])` and modifying all components and state accesses, for example. Although the algorithm looks similar to its imperative counterpart, in our case all the state effects are explicit and the type system delineates precisely where they can occur: there are no implicit side-effects and the validity of the sequence of operations can be checked by the compiler using its normal type checking mechanism.

In order to ensure that individual algorithms are easily and mechanically composable, we generalise over the type of the state. For example, instead of specifying that the type state has type `Int` in Listing 3, we use the type variable `Env` ('environment'). In principle, this abstracts over all possible types of environment, and results in a single generic definition of Local Search. Since we have abstracted over the environment, components still need a mechanism to access the information within the environment. The contemporary functional programming solution is to use *lenses* (Foster et al., 2005). Informally, a lens of type *Lens[Source,Target]* is a composable mechanism that focusses an object of type *Source* into an object of type *Target* that is contained within *Source*, for example by selecting a sub-field of a record type. This is illustrated in Listing 4, a re-formulation of Listing 3 to use lenses in addition to the State monad. Instead of explicitly accessing the environment in a manner that assumes a specific `Env`, the lens `iter` is used to express the same operation in a generic manner. The lens abstracts the access mechanism for a specific *Source* and *Target* of the algorithm or component. Should the practitioner wish to add solution history to the local search algorithm in Listing 4, for example, they now only need to add a lens to access the solution history that is accessible from any components that may wish to access it. This is illustrated in Listing 5, which defines an acceptance component with access to the solution history via the lens `history`. Without the lens, re-formulating Local Search in order to include solution history requires the practitioner to modify all components and state accesses within the algorithm itself.

In our approach, component dependencies are expressed via lenses in order to allow them to be as agnostic as possible about the specific means by which the *Target* value is obtained from the *Source* value. Lenses are used in conjunction with monads in order to allow for simple composition of components with explicit state dependences. This ensures that all state dependencies are clearly defined and are available in a machine-readable form. Monads and lenses facilitate the AOCP as follows:

- **Open**: State information explicitly manipulated by *LocalSearch* is provided in the signature of `apply` of Listing 4, to be made available for reusable components, e.g. as described in Section 5. A subsequently-authored component can get access to this state by providing an appropriate lens.

- **Closed**: As described above, the key motivation for the use of state monads is precisely that they allow the framework to meet the extended Closed requirements of the AOCP w.r.t. management of state dependencies.

- **Introspectable Design**: The strong typing of the proposed approach means that the assembled system is guaranteed to have both state and parameter dependencies satisfied.

- **Observable**: Since the state monad is responsible for propagating all state transitions, they can be directly observed. Further, this information can be made available as a form of dynamic instrumentation, to be exploited online by other components.

Related Work on State Handling

A number of authors have studied state-handling in metaheuristics via *combinators* — pure functional, self-contained components. Following initial work in Genetic Programming and exhaustive search (Briggs and O'Neill, 2008), monadic combinators were used in constraint programming (Schrijvers et al., 2013), with subsequent work in metaheuristics (Senington and Duke, 2013). The desirability of the monadic approach for metaheuristic design was recently advocated (Swan et al., 2015) as part of a wider research program, echoed by an emphatic call for re-usable libraries such as *CILib* (Pampara and Engelbrecht, 2015). The only work that we are aware of in which state-handling is used *in explicit support of metaheuristic assembly* is given by (Kocsis et al., 2015), who propose a combinator-based approach where component dependencies are amalgamated into a shared workspace.

5 Example Application of the Automated Open Closed Principle

In this section, we describe a system that follows the Automated Open Closed Principle. Here, as discussed in the previous section, we use a combination of state monads and lenses to define a local search framework and its component parts. Subsequently, we expose these components to an automated tool, which assembles them into a working local search algorithm. As our target problem domain, we have chosen for the well-known Traveling Salesperson Problem (TSP) (Luke, 2013). Please note that the experiments performed in this section are merely illustrative, i.e. they aim to illustrate that the resulting local search framework indeed supports "automated combinatorial assembly" as is intrinsic to the AOCP. It was *not* our objective to design a heuristic competitive with the state-of-the-art for the TSP.

Metaheuristic Components:

As per Listing 4, we consider a local search framework that has three top-level design choices (*perturb,accept,isFinished*). We consider a single termination condition (*isFinished*), terminating local search after maxIter = 10000 iterations. We consider the following types of perturbations (*perturb*):

- **RandomSwap**: Swaps two cities at random.
- **RandomInsert:** Removes a randomly selected city, reinserting it in a random position.
- **RandomShuffle:** Randomly exchanges all cities.
- **RandomShuffleSubset(m):** Selects $k = 2 + \lfloor m * (n-2) \rfloor$ cities at random and exchanges them randomly, where n is the number of cities in the TSP instance.
- **ReverseSubtours(m):** Reverses $k = 1 + \lfloor 4 * m \rfloor$ randomly selected subtours.

Remark that the last two perturbations are in turn parametrized, taking a real-valued mutation rate $m \in [0,1]$ as argument. We also consider three types of acceptance conditions (*accept*):

- **AcceptImproving**: Accepts all incoming tours shorter than the incumbent.
- **AcceptImprovingOrEqual**: Accepts all incoming tours no longer than the incumbent.

- **AcceptMetropolisHastings(*s*)**: Accepts incoming tours with likelikhood $e^{\frac{f(t)-f(t')}{T}}$, where $f(t)$ and $f(t')$ are the length of the incumbent and incoming tour respectively, and T is the current temperature (~ a positive scalar). The initial temperature T_0 is instance-specific and determined as described in (White, 1984). The value of T in subsequent iterations is dynamically controlled by a cooling schedule s. We consider two types:
  - **LinearCoolingSchedule**: The temperature after iter iterations is $T_0 * (1 - \frac{\text{iter}}{\text{maxIter}})$.
  - **GeometricCoolingSchedule(*r*)**: Reduces the current temperature with a factor r.

These components have a number of different state dependencies. The state monad in our example keeps track of:

- **iter**: The number of iterations performed.
- **maxIter**: The optimization budget.
- **T**: The current temperature.
- **rng**: A stream of (pseudo) random numbers.
- **f**: A procedure for computing the quality of a solution (length of a tour in our example).

These component-specific state dependencies are modeled using lenses which appear explicitly in the component signatures as shown in Listing 6. Remark that each component is agnostic about the environmental dependencies of any other component, resulting in a loose coupling. For instance, the core *LocalSearch* framework can therefore remain unchanged regardless of the complexity of the components with which it is configured. Whilst this framework is straightforward, algorithms such as Scatter-search or Learning Classifier Systems (Luke, 2013) are notoriously difficult to implement correctly; the ability to define a truly re-usable reference implementation via the AOCP serves a vital scientific purpose in ensuring the validity and reproducibility of reported results.

Automated Assembly:

Using this collection of components, a set of different candidate local search methods can be assembled. Remark that this set (a.k.a. the *design space*) is infinite, as the parameters m and r can take any real value in $[0,1]$. However, in this illustration, we only consider the finite subset of 350 candidate designs with $m, r \in \{0.0, 0.1, \ldots, 0.9, 1.0\}$.

In automated assembly, the objective is to (automatically) determine which combination of components is best suited for solving a given target problem (e.g. the TSP). In literature, a variety of different approaches have been explored to do so. A prominent example is genetic programming (Koza, 1994), which solves this (meta-)optimization problem using evolutionary algorithms. Please note that we do not intend to advocate any specific approach or optimizer. Rather, the aim of this paper is to facilitate the implementation of re-usable component repositories (~ design spaces) supporting "automated combinatorial assembly" using any of these (and future) design automation techniques. We will illustrate two different approaches:

Ant Programming (AP, (Boryczka, 2002)) is a variant of Ant-Colony Optimization (Luke, 2013) in which the combinatorial structure to be optimized is a program tree. In our illustration, we use ContainAnt (Kocsis and Swan, 2017) as our optimizer.[2] The search space of structures can usefully be described via a grammar. In the case of CONTAINANT, this grammar is specified directly in Scala code, by the fields and methods of a user-defined subclass

[2] `https://github.com/zaklogician/ContainAnt` (source code)

```
type Tour = org.mitlware.solutions.permutation.Permutation
type TourLengthLens[Env] = Lens[Env, Evaluate[Env,Tour,Double]]

class LocalSearchADoM[Env] extends Module {

  // Grammar specification ////////////////////

  // Lenses:
  val iter: Lens[Env, Iter] = ...
  val rng: Lens[Env, RNG] = ...
  val maxIter: Lens[Env, MaxIter] = ...
  val T: Lens[Env, Temperature] = ...
  val f: TourLengthLens[Env] = ...

  // Perturbation:
  def randomSwap(rng: Lens[Env, RNG]): Perturb[Env,Tour] = ...
  def randomInsert(rng: Lens[Env, RNG]): Perturb[Env,Tour] = ...
  def randomShuffle(rng: Lens[Env, RNG]): Perturb[Env,Tour] = ...
  def randomShuffleSubset(m: MutationStrength, rng: Lens[Env, RNG]):
    Perturb[Env,Tour] = ...
  def reverseSubtours(m: MutationStrength, rng: Lens[Env, RNG]):
    Perturb[Env,Tour] = ...

  // Alternative values for mutation strength
  val m_values = for(x <- 0 to 10) yield {MutationStrength(x/10.0)}

  // Acceptance:
  def acceptImproving(f: TourLengthLens[Env]): Accept[Env,Tour] = ...
  def acceptImprovingOrEqual(f: TourLengthLens[Env]): Accept[Env,Tour] = ...
  def acceptMetropolisHastings(s: CoolingSchedule[Env], rng: Lens[Env, RNG],
    f: TourLengthLens[Env]): Accept[Env,Tour] = ...

  // Cooling schedules:
  def linearCoolingSchedule(iter: Lens[Env,Iter], maxIter: Lens[Env,MaxIter],
    T: Lens[Env,Temperature]): CoolingSchedule[Env] = ...
  def geometricCoolingSchedule(r: CoolingRatio, T: Lens[Env,Temperature]):
    CoolingSchedule[Env] = ...

  // Alternative values for cooling ratio
  val r_values = for(x <- 0 to 10) yield {CoolingRatio(x/10.0)}

  // Termination criterion:
  def isFinished(iter: Lens[Env,Iter], maxIter: Lens[Env,MaxIter]):
    IsFinished[Env,Tour] = ...

  ////////////////////////////////////////////

  // LocalSearch is the target object for automated construction:
  def localSearch(
    perturb: Perturb[Env,Tour],
    accept: Accept[Env,Tour],
    finished: IsFinished[Env,Tour],
    iter: Lens[Env,Iter]): LocalSearch[Env,Tour] = ...
}
```

Listing 6: Grammar specification for our automated Local Search assembly example

of *containant.Module*. By this means, it is possible to specify any context-free grammar in terms of the attributes (*val*) and method signatures (*def*) that are automatically obtained from the grammar module via reflection. Remark that this grammar is statically-typed.The grammar for our example is shown is Listing 6. Note that CONTAINANT was not designed with any explicit knowledge of monads and lenses: to the Ant Programming search algorithm, they are treated just like any other type.

Programming by Optimization (PbO, Hoos (2012)). In PbO, design choices are exposed as program parameters whose values correspond to alternative decisions. Subsequently, the best combination of parameter values (configuration) is determined automatically with the help of automated tools known as algorithm configurators. We used SMAC Hutter et al. (2011b) as configurator in our illustration.[3] We exposed our design choices in the form of 5 parameters: *perturb*, *m*, *accept*, *s* and *r*; having 5, 11, 3, 2 and 11 possible values, respectively. As each component is parametrized by its design choices, doing so did not require us to modify any existing code. Nonetheless, creating the "tuning scenario" (wrapper program, parameter specification file, etc.) was a tedious and error-prone process. To alleviate this issue, *language extensions* have been proposed in prior-art Hoos (2012); Ansel et al. (2009), supporting the manual "annotation" of design choices in code, allowing them to be extracted automatically. Interestingly, in a system following the AOCP this process could be automated using reflection, supporting statically-typed specifications in *native code* similar to Listing 6.

Both AP and PbO evaluate "how good" a candidate design is by testing it on instances of the target problem (a.k.a. training instances). Remark that, to avoid bias, a disjoint set of instances must be used to assess the performance of the design returned by the optimizer (a.k.a. test instances). The TSP instances that we consider here are the subset of 76 symmetric TSPs from the well known TSPLIB that have edge weights in the 'Euclidean 2D' format.[4]. More specifically, the training set consists of the 12 instances with 100 cities or less and the test set consists of the 57 (of 64) remaining instances with less than 4,000 cities. It is well known that automated algorithm design is highly computationally-intensive, and these smaller instances are used so that all experiments completed in under 72 hours on a modern desktop PC.[5]

We run both ContainAnt and SMAC once, using their default parameter settings, permitting them a total of 6000 tests to determine which of the 350 candidate local searches minimizes the average *Relative Error* (RE), given by $\frac{\text{tour\_length} - \text{optimal\_tour\_length}}{\text{optimal\_tour\_length}}$, on the training instances, when applied to an initial solution obtained by the nearest neighbor heuristic.

Remark that both ContainAnt and SMAC are *anytime*, i.e. they can at any time be queried for the best design they found thus far. Figures 1 and 2 show the average RE on the training and test instances, respectively, of the anytime solution obtained by each meta-optimizer.[6]

We observe that both SMAC and ContainAnt discover better designs (lower average RE) over time. In this pair of runs, ContainAnt clearly started from a worse initial design. However, after 48 tests it identified a better alternative and in the remainder of its run found equally good designs roughly 2-3 times faster than SMAC. While this result provides additional evidence of the competitiveness of ContainAnt (c.f. (Kocsis and Swan, 2017)), it is insufficient to draw any conclusions about the relative performance of SMAC and ContainAnt in general. In other tuning scenarios (or another run) SMAC may do better. Also note that SMAC has various features (e.g. adaptive capping, incremental evaluation) which will likely make it preferable in

[3] `http://www.cs.ubc.ca/labs/beta/Projects/SMAC/` (version 2.10)

[4] `http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/index.html`

[5] Specifically, Windows 10, Intel™ i5 CPU 3.4 GHz

[6] Figures 1-3 show unbiased estimates of the average RE based on 100 tests per instance.

Figure 1: Anytime performance on *training set*

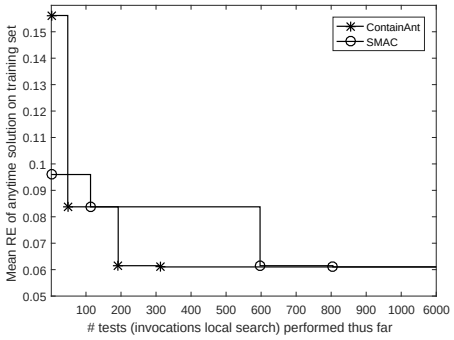


Figure 2: Anytime performance on *test set*

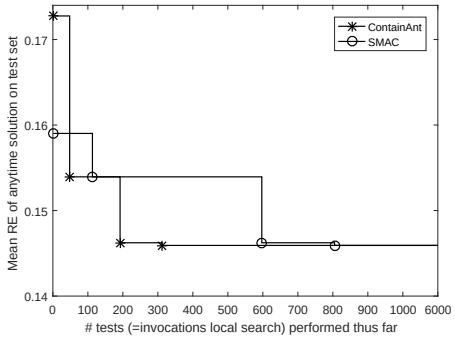
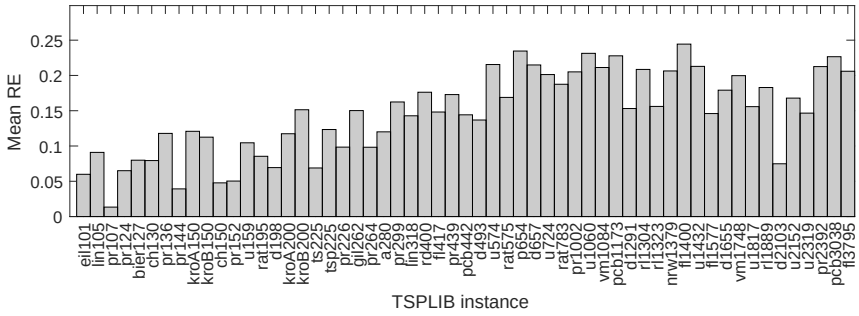


Figure 3: Performance on each test instance of the best candidate local search found



certain settings (e.g. optimization of runtime, noisy fitness criterion).

After 6000 tests, both SMAC and ContainAnt return the same design (using ReverseSubTours(0.2) and AcceptImproving), suggesting it is the best (on the training set) of the 350 candidates considered. This design obtains an average RE of 0.061 on the training instances, and an average RE of 0.146 on the test instances. Remark that the higher average RE on the test instances is likely due to the fact that these instances are harder. We do not believe our results suffer from over-tuning given the similarity of Figures 1 and 2.

For completeness, Figure 3 shows the average RE of this design on each of the 57 test instances. Unsurprisingly, its performance is not competitive with that of the state-of-the-art.

6 Consequences of the Automated Open-Closed Principle

As well as addressing the immediate problem of state dependencies, the mechanisms of the AOCP bring many benefits in metaheuristic assembly. Here, we discuss the most significant.

6.1 Parallelization

By explicitly expressing state dependencies, following the AOCP facilitates the creation of parallelizable metaheuristics. While mechanisms such as monads express the logical *ordering* of operations, they do not enforce sequential *execution* where dependencies do not exist. These facts can be exploited to automatically introduce parallelism (Castro et al., 2016; Scaife et al., 2006), and to restructure programs to automatically expose alternative parallelizations, so allowing automated determination of the most efficient parallel program. Structured par-

```
type Islands = List[List[Sol]]

def EMAS(
  islands: Islands,
  meetingAgent: Perturb[Env,Islands],
  migrationFun: Perturb[Env,Islands],
  isFinished: IsFinished[Env,Islands]): Islands =
    for {
      agentsMet <- runPar (parMap meetingAgent islands);
      agentsMigrated <- agentsMet;
      finished <- isFinished agentsMigrated
      result <- if (finished)
                  agentsMigrated
                else
                  EMAS(agentsMigrated,meetingAgent,migrationFun,isFinished)
    } yield result
```

Listing 7: Monadic formulation of an Evolutionary Multi-Agent System

allelism techniques, such as *algorithmic skeletons* (Cole, 1988), can then be used to obtain a good parallel implementation. Such techniques take a high-level, modular approach to parallelism, presenting to the programmer high-level patterns that abstract low-level implementation details such as communication, task creation, and scheduling. They avoid a number of problems such as race conditions and deadlocks that are difficult to diagnose or debug (Barwell et al., 2016).

Structured parallelism techniques have a long-observed correspondence with functional programming. For example, the '*Par* monad' library for Haskell (Marlow et al., 2011) provides the *parMap evaluation strategy* (Trinder et al., 1998), that applies a given function to each element in a data-structure in parallel in a simple and *thread-safe* way (Hammond and Michaelson, 1999). To illustrate this, consider the Evolutionary Multi-Agent System (EMAS) given in Listing 7. EMASs are a hybrid metaheuristic that combine multi-agent systems with evolutionary algorithms. Each agent represents a solution to a particular optimization problem that is iteratively improved by a series of reproductions and fitness contests with other agents (Stypka et al., 2018). The population of agents is subdivided into islands. Randomized migration between islands maintains diversity. Agents co-operate with or compete against other agents on the same island, and are periodically chosen for migration to other islands. During this process, *meetingAgent* is applied to each island of agents as a *map* operation, and is parallelized using *parMap*, where *runPar* enables the use of *parMap*.

6.2 Credit Assignment

Various mechanisms (e.g. perturbation scheme, acceptance and restart condition) affect the behavior of a metaheuristic. This gives rise to a structural *Credit Assignment Problem* (Minsky, 1961) (CAP): Which of these mechanisms is to blame/contributes to the poor/good performance we observe on a particular problem instance?

Beyond its scientific merit, the information that is available from credit assignment can be exploited to guide automated metaheuristic assembly. For instance, ParamILS (Hutter et al., 2009) performs an (iterated) local search in the one-exchange neighborhood, i.e. it changes a single component at the time and figures out whether this improves or worsens the performance of the incumbent design and updates it accordingly. SMAC (Hutter et al., 2011a) builds a regression model mapping configurations to performance predictions, effectively learning which combinations of parameter values perform well together. SMAC uses this model to determine which configuration to try next.

These approaches share the property that credit is assigned *post-execution*, based on *correlations* between the performance observations of the system *as a whole* and its components. While the AOCP supports these correlation-based approaches, it also facilitates the study of *causality* (Adriaensen and Nowé, 2016b) by examining the behavior of components *during execution* (c.f. previous work relating execution state to evaluation (Krawiec and Swan, 2013)). This allows for the decoupling of design and performance spaces, enabling performance observations to generalize beyond the specific design that is used to obtain them, based on similarities in execution space (e.g. using Importance Sampling (Adriaensen et al., 2017)). This has the potential for more efficient, accurate and insight-promoting credit assignment.

A simple example of behavioral information that can be exploited to perform more accurate credit assignment are *unused parameters*. Often, not all parameters of a metaheuristic framework are used during every run. The values of these unused parameters can obviously not be held responsible for the performance observed. Remark that while some contemporary configurators (e.g. ParamILS and SMAC) support parameter dependencies (a.k.a. conditional parameters), they are oblivious of the fact that even "active" parameters are sometimes not used (e.g. for certain problem instance + random seed combinations). Falsely assigning credit to the values of unused parameters introduces unnecessary noise, complicating configuration.

More generally, when taking a behavioral perspective, the structural CAP reduces to a temporal CAP, a problem that is widely studied in the Reinforcement Learning community (Sutton and Barto, 1998). Of particular interest to the modular approach laid out in this paper is 'hierarchical' reinforcement learning (Barto and Mahadevan, 2003), which studies the problem of learning complex tasks through learning simpler sub-tasks and exploiting their inter-dependencies. Also, the temporal CAP can be solved online using Temporal Difference (TD) techniques (Tesauro, 1995) such as Q-learning (Watkins and Dayan, 1992). Such methods are able to learn what works best and to exploit this knowledge *while* solving the problem.

While the use of these techniques in the context of metaheuristics has been explored (e.g. in the area of reactive search (Battiti et al., 2008)), their full potential is arguably yet to be unlocked, and it is our belief that the AOCP will also greatly facilitate this endeavor.

6.3 Scalability

The intrinsically recursive nature of metaheuristics has been widely noted (Vaessens et al., 1998; Swan et al., 2011, 2014; López-Ibáñez et al., 2014). Using a truly modular approach, combinations of components can themselves be considered as components (Woodward et al., 2014). For example, the local search framework of Listing 4 can itself be treated as a perturbation operator. The AOCP supports the full exploration of this recursion, enabling the exploration of deeply-nested combinations of heuristics.

6.4 Escaping the Poverty of Metaphor

Following the success of Genetic Algorithms, many metaheuristics have been derived via 'reasoning by metaphor'. This has led to a very large number of competing metaheuristics, and the relationship between them is often unclear, a situation that has recently been criticized (Sörensen, 2013). Without true separation of concerns, it is difficult to either formally identify the equivalence of two components or to combine components in an automated manner, making it impossible in many cases for one metaheuristic to borrow components from another. A modular representation escapes these limitations and offers opportunities to explore a much greater design space via automated hybridization.

As noted in Section 6.2, the absence of hidden state allows unused parameters to be discovered in order to properly identify useful components. An extreme example of the importance of progressing beyond the current situation is given by (Nair et al., 2016), where

a superior algorithm that did not incorporate an evolutionary metaphor was discovered by accidentally disabling an evolutionary component. The proposed approach provides the opportunity to discover such algorithms automatically, allowing elimination of accidental complexity to become routine practice as advocated by (Adriaensen and Nowé, 2016a).

7 Conclusions and Future Work

State-of-the-art approaches to automated metaheuristic configuration and design require considerable 'human in the loop' intervention to extend the frameworks they provide. Manual effort is required to manage inter-dependencies between the framework and its components. To address this problem, we introduce the 'Automated Open-Closed Principle' (AOCP), an extension to the well-known 'Open-Closed Principle' of software engineering, which we claim is essential to scalable design automation not only of metaheuristics, but of algorithms in general. Adoption of the AOCP leads to design space descriptions that:

1. Are re-usable at the level of both frameworks and components.
2. Are *purely functional*, with unintrusive support for component-specific state.
3. Can be assembled 'bottom-up', from an open-ended palette of components.
4. Do not require scale-dependent human intervention when incorporating components with new dependencies.

As validation of our approach, we use an exemplar of the AOCP to assemble a local search algorithm with a higher degree of automation than previously achieved.

Future work will explore the potential for scalability and reusability across a range of alternative metaheuristic frameworks, both in terms of the size of the palette of components and the size of the problems to be addressed. Since the notion of the 'Automated Open-Closed Principle' that we propose is a general one, it can further be used for automated assembly in other domains, including Machine Learning and Software Engineering.

8 Acknowledgements

The utility of communal design templates forms a cornerstone of the 'Metaheuristics in the Large' community initiative (Swan et al., 2015). The authors would like to thank all the many collaborators in this initiative.

References

Adriaensen, S., Brys, T., and Nowé, A. (2014). Designing reusable metaheuristic methods: A semi-automated approach. In *Evolutionary Computation (CEC), 2014 IEEE Congress on*, pages 2969–2976. IEEE.

Adriaensen, S., Moons, F., and Nowé, A. (2017). An importance sampling approach to the estimation of algorithm performance in automated algorithm design. In *Proceedings of the 11th International Conference on Learning and Intelligent Optimization (LION)*, pages 3–17. Springer.

Adriaensen, S. and Nowé, A. (2016a). Case study: An analysis of accidental complexity in a state-of-the-art hyper-heuristic for hyflex. In *Evolutionary Computation (CEC), 2016 IEEE Congress on*, pages 1485–1492. IEEE.

Adriaensen, S. and Nowé, A. (2016b). Towards a white box approach to automated algorithm design. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 554–560. AAAI Press.

Ansel, J., Chan, C., Wong, Y. L., Olszewski, M., Zhao, Q., Edelman, A., and Amarasinghe, S. (2009). *PetaBricks: a language and compiler for algorithmic choice*, volume 44. ACM.

Ansótegui, C., Sellmann, M., and Tierney, K. (2009). A Gender-based Genetic Algorithm for the Automatic Configuration of Algorithms. In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming*, CP'09, pages 142–157, Berlin, Heidelberg. Springer-Verlag.

Barto, A. G. and Mahadevan, S. (2003). Recent advances in hierarchical reinforcement learning. *Discrete event dynamic systems*, 13(4):341–379.

Barwell, A. D., Brown, C., Hammond, K., Turek, W., and Byrski, A. (2016). Using program shaping and algorithmic skeletons to parallelise an evolutionary multi-agent system in erlang. *Computing and Informatics*, 35(4):792–818.

Battiti, R., Brunato, M., and Mascia, F. (2008). *Reactive search and intelligent optimization*, volume 45. Springer Science & Business Media.

Bezerra, L. C., López-Ibánez, M., and Stützle, T. (2016). Automatic component-wise design of multiobjective evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 20(3):403–417.

Boryczka, M. (2002). *Ant Colony Programming for Approximation Problems*, pages 147–156. Physica-Verlag HD, Heidelberg.

Briggs, F. and O'Neill, M. (2008). Functional genetic programming and exhaustive program search with combinator expressions. *Int. J. Know.-Based Intell. Eng. Syst.*, 12(1):47–68.

Castro, D., Hammond, K., and Sarkar, S. (2016). Farms, pipes, streams and reforestation: reasoning about structured parallel processes using types and hylomorphisms. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, pages 4–17. ACM.

Cole, M. (1988). *Algorithmic skeletons : a structured approach to the management of parallel computation*. PhD thesis, University of Edinburgh, UK.

Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C., and Schmitt, A. (2005). Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 233–246.

Glover, F. and Laguna, M. (1997). *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA.

Hammond, K. and Michaelson, G. (1999). *Research Directions in Parallel Functional Programming*. Springer.

Hoos, H. H. (2012). Programming by optimization. *Commun. ACM*, 55(2):70–80.

Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2011a). Sequential Model-Based Optimization for General Algorithm Configuration. In *Proc. of LION-5*, page 507âĂŞ523.

Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2011b). Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization*, pages 507–523. Springer.

Hutter, F., Hoos, H. H., Leyton-Brown, K., and Stützle, T. (2009). ParamILS: An Automatic Algorithm Configuration Framework. *J. Artif. Int. Res.*, 36(1):267–306.

KhudaBukhsh, A. R., Xu, L., Hoos, H. H., and Leyton-Brown, K. (2009). Satenstein: Automatically building local search sat solvers from components. In *IJCAI*, volume 9, pages 517–524.

Kirkpatrick, S., Jr., D. G., and Vecchi, M. P. (1983). Optimization by simmulated annealing. *Science*, 220(4598):671–680.

Kocsis, Z. A., Brownlee, A. E. I., Swan, J., and Senington, R. (2015). *Haiku - a Scala Combinator Toolkit for Semi-automated Composition of Metaheuristics*, pages 125–140. Springer International Publishing, Cham.

Kocsis, Z. A. and Swan, J. (2017). Dependency injection for programming by optimization. submitted 13th July 2017.

Koza, J. R. (1994). Genetic programming as a means for programming computers by natural selection. *Statistics and computing*, 4(2):87–112.

Krawiec, K. and Swan, J. (2013). Pattern-guided genetic programming. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, GECCO '13, pages 949–956, New York, NY, USA. ACM.

Larman, C. (2001). Protected variation: the importance of being closed. *IEEE Software*, 18(3):89–91.

Liskov, B. (1987). 'Data Abstraction and Hierarchy' (keynote address). *SIGPLAN Not.*, 23(5):17–34.

López-Ibáñez, M., Dubois-Lacoste, J., Pérez Cáceres, L., Stützle, T., and Birattari, M. (2016). The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58.

López-Ibáñez, M., Mascia, F., Marmion, M., and Stützle, T. (2014). A template for designing single-solution hybrid metaheuristics. In Arnold, D. V. and Alba, E., editors, *Genetic and Evolutionary Computation Conference, GECCO '14, Vancouver, BC, Canada, July 12-16, 2014, Companion Material Proceedings*, pages 1423–1426. ACM.

López-Ibáñez, M. and Stützle, T. (2012). The automatic design of multi-objective ant colony optimization algorithms. *IEEE Transactions on Evolutionary Computation*, 16(6):861–875.

Luke, S. (2013). *Essentials of Metaheuristics*. Lulu, second edition. Available for free at http://cs.gmu.edu/~sean/book/metaheuristics/.

Mac Lane, S. (1969). *Categories for the Working Mathematician.* Springer.

Marlow, S., Newton, R., and Peyton Jones, S. (2011). A monad for deterministic parallelism. *SIGPLAN Not.*, 46(12).

Mascia, F., López-Ibáñez, M., Dubois-Lacoste, J., and Stützle, T. (2014). Grammar-based generation of stochastic local search heuristics through automatic algorithm configuration tools. *Computers & operations research*, 51:190–199.

Meyer, B. (1988). *Object-oriented software construction*, volume 2. Prentice hall New York.

Minsky, M. (1961). Steps toward artificial intelligence. *Proceedings of the IRE*, 49(1):8–30.

Moggi, E. (1991). Notions of Computation and Monads. *Information and Computation*.

Nair, V., Menzies, T., and Chen, J. (2016). An (accidental) exploration of alternatives to evolutionary algorithms for SBSE . In *International Symposium on Search Based Software Engineering*, pages 96–111. Springer.

Pampara, G. and Engelbrecht, A. (2015). Towards a generic computational intelligence library: Preventing insanity. In *2015 IEEE Symposium Series on Computational Intelligence: IEEE Workshop on Computational Intelligence Tools (2015 IEEE WCIT)*, Cape Town, South Africa.

Scaife, N., Michaelson, G., and Horiguchi, S. (2006). Parallel standard ML with skeletons. *Scalable Computing: Practice and Experience*, 7(2).

Schrijvers, T., Tack, G., Wuille, P., Samulowitz, H., and Stuckey, P. (2013). Search combinators. *Constraints*, 18(2):269–305.

Senington, R. and Duke, D. (2013). Decomposing metaheuristic operations. In Hinze, R., editor, *Implementation and Application of Functional Languages*, LNCS, pages 224–239. Springer Berlin Heidelberg.

Smith-Miles, K., Baatar, D., Wreford, B., and Lewis, R. (2014). Towards objective measures of algorithm performance across instance space. *Computers & OR*, 45:12–24.

Sörensen, K. (2013). Metaheuristics—the metaphor exposed. *International Transactions on Operational Research*, 22(1):3–18.

Stypka, J., Turek, W., Byrski, A., Kisiel-Dorohinicki, M., Barwell, A. D., Brown, C., Hammond, K., and Janjic, V. (2018). The missing link! A new skeleton for evolutionary multi-agent systems in erlang. *International Journal of Parallel Programming*, 46(1):4–22.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge.

Swan, J., Adriaensen, S., Bishr, M., Burke, E. K., Clark, J. A., Causmaecker, P. D., Durillo, J., Hammond, K., Hart, E., Johnson, C. G., Kocsis, Z. A., Kovitz, B., Krawiec, K., Martin, S., Merelo, J. J., Minku, L. L., Özcan, E., Pappa, G. L., Pesch, E., Garcia-Sànchez, P., Schaerf, A., Sim, K., Smith, J., Stützle, T., Voß, S., Wagner, S., and Yao, X. (2015). A research agenda for metaheuristic standardization. In *Proceedings of the Eleventh Metaheuristics International Conference (MIC), Agadir, Morocco*.

Swan, J., Özcan, E., and Kendall, G. (2011). *Hyperion – A Recursive Hyper-Heuristic Framework*, pages 616–630. Springer Berlin Heidelberg, Berlin, Heidelberg.

Swan, J., Woodward, J., Özcan, E., Kendall, G., and Burke, E. (2014). Searching the hyper-heuristic design space. *Cognitive Computation*, 6(1):66–73.

Tesauro, G. (1995). Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68.

Trinder, P. W., Hammond, K., Loidl, H., and Jones, S. L. P. (1998). Algorithms + strategy = parallelism. *J. Funct. Program.*, 8(1):23–60.

Vaessens, R. J. M., Aarts, E. H. L., and Lenstra, J. K. (1998). A local search template. *Comput. Oper. Res.*, 25(11):969–979.

Wadler, P. (1995). *Monads for functional programming*, pages 24–52. Springer Berlin Heidelberg, Berlin, Heidelberg.

Watkins, C. J. and Dayan, P. (1992). Q-learning. *Machine learning*, 8(3):279–292.

Weyland, D. (2010). A rigorous analysis of the harmony search algorithm: How the research community can be misled by a "novel" methodology. *Int. J. Appl. Metaheuristic Comput.*, 1(2):50–60.

White, S. R. (1984). Concepts of scale in simulated annealing. *AIP Conference Proceedings*, 122(1):261–270.

Woodward, J., Swan, J., and Martin, S. (2014). The 'Composite' design pattern in metaheuristics. In *Proc. GECCO Comp.*, pages 1439–1444, New York, USA. ACM.