An Aspect-Oriented Approach for Implementing Evolutionary Computation Applications [*]

Andres J. Ramirez, Adam C. Jensen, and Betty H.C. Cheng
Michigan State University
Department of Computer Science and Engineering
3115 Engineering Building
East Lansing, MI 48823
{ramir105, acj, chengb}@cse.msu.edu

ABSTRACT

Object-oriented frameworks support design and code reuse for specific application domains. To facilitate the development of evolutionary computation (EC) programs, such as genetic algorithms, developers often extend and customize EC frameworks with application code that defines the EC problem being solved. The application code, however, crosscuts the EC framework whenever candidate solutions are encoded, decoded, evaluated, and output. A change in the application logic, such as adding a parameter to the problem being solved, requires additional changes across code that extends the framework. This paper presents Arachne, an aspect-oriented approach for developing EC programs that extracts the crosscutting concerns of an application code into aspects that can be woven into the EC framework at compile time. To facilitate applying Arachne, we implemented a prototype tool to support the automatic generation of aspect code for two widely used EC frameworks, JGAP and ECJ. We demonstrate Arachne by applying it to re-engineer EC benchmark programs and an industry-provided problem.

Categories and Subject Descriptors

D.2.2 [**Software**]: Software Engineering - Design Tools and Techniques

General Terms

Design

*This work has been supported in part by NSF grants CCF-0541131, IIP-0700329, CCF-0750787, CCF-0820220, DBI-0939454, CNS-0854931, Army Research Office grant W911NF-08-1-0495, Ford Motor Company, and a Quality Fund Program grant from Michigan State University. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation, Army, Ford, or other research sponsors.

Keywords

Aspect-orientation, evolutionary algorithm, genetic algorithm

1. INTRODUCTION

Evolutionary computation (EC) is a family of stochastic search-based techniques, such as a genetic algorithm [10], that is often applied to solve complex optimization problems in a wide range of domains, including software engineering [8, 11, 24]. An object-oriented framework [6] provides a reusable set of cooperating classes that can be extended and customized for a particular type of software. As a result, developers often leverage EC frameworks [5, 18, 20] to facilitate the development of EC programs. This framework-oriented approach for EC-based programs, however, scatters and entangles the application code (e.g., how a candidate solution is represented within a framework's classes) throughout multiple extension points in the EC framework. The crosscutting application code, in turn, complicates maintenance tasks, increases the potential for system inconsistencies if changes are not properly propagated, and hinders the reuse of application code across different EC frameworks and approaches. This paper presents Arachne, an aspect-oriented programming (AOP) [14] approach for developing EC programs that extracts the crosscutting concerns of application code into aspects that can be woven into the EC framework at compile time. After applying Arachne, the resulting EC program has better modularity, becomes easier to maintain, and facilitates the reuse of application code across different EC programs and frameworks.

To use an EC framework, developers must first determine how to encode, or map, the general structure of a candidate solution to classes provided by the EC framework (e.g., a vector of integers). Developers must also determine how to measure the quality, or fitness, of a candidate solution to guide the search process towards better areas of the solution space. In this manner, a candidate solution's encoding and its semantics permeate multiple classes and extension points in an EC framework. As a result, developers must scatter and entangle application code throughout various extension points where the EC framework maps an encoded solution to the application domain and vice-versa. This approach implies that a single change in the underlying problem or structure of a candidate solution (e.g., adding an optimization parameter) requires propagating corresponding changes to the encoding, decoding, evaluation, and output modules of the EC framework. Given the tight coupling between a solution and its representation and semantics, it is difficult to build black-box frameworks that hide the internal details of the frameworks' design and implementation. Furthermore, the dependency between these modules increases the difficulty of preserving system consistency and complicates the reuse of application code in other EC systems.

This paper introduces Arachne, an aspect-oriented approach for developing EC programs without requiring developers to scatter and entangle application code across various EC framework extension points. Specifically, Arachne uses *aspects* to encapsulate application code that maps candidate solutions to either a representation within the EC framework or back to the problem domain, thereby separating the application code from the EC framework until compile time. In this manner, Arachne hides the internal details about an EC framework's design and implementation by encapsulating the locations, or pointcuts, where application code must be inserted into the EC framework. To reduce the efforts required with identifying pointcuts in a target EC framework, we developed a prototype tool that supports the generation of aspect code for several widely-used EC frameworks. Leveraging Arachne in the development of an EC program also facilitates maintenance tasks while preserving system consistency by automatically propagating modifications in the application code to their corresponding EC framework extension points. Lastly, by separating the application code from the EC framework, Arachne facilitates the *migration* of an EC program across different EC frameworks. Such a migration, which is often a time-consuming task, may be useful for developers who are changing frameworks either to obtain different EC computation support [5] or to compare the relative performance of different frameworks.

Arachne supports a systematic process to separate the application code from the EC framework at the implementation level, as well as to automatically weave them together at compile time. First, in the Arachne approach, a solution comprises not only the application code that defines the optimization problem, but also the evaluation criteria that determines the quality of a candidate solution. Next, Arachne introduces aspects to map a solution either to a set of classes in the EC framework or back to the problem domain. Each aspect specifies the extension points in the target EC framework where the corresponding application code will be inserted during compilation. Essentially, these *Encoding* and *Decoding* aspects enable Arachne to redirect the EC framework's evaluation and output of candidate solutions to the application code. Then an aspect weaver [9, 14] automatically compiles the application code and EC framework together, thus producing a self-contained executable program.

We conducted several experiments to assess the efficacy of applying AOP to separate the application code from the EC framework. First, we applied Arachne to re-engineer three EC application benchmarks provided with two widely used EC frameworks, ECJ [18] and JGAP [20]. We then re-implemented Plato [21], an EC program developed for an industry-provided problem that autonomously reconfigures networks of remote data mirrors. For each experiment, we compare the resulting aspect-oriented system with its traditional object-oriented counterpart in terms of code reuse, maintenance, and modularization, leveraging the Software Engineering metrics suite by Chidamber *et al.* [4] and AOP metrics suite [3] where applicable. Experimental results suggest AOP is beneficial in modularizing EC programs, thus simplifying maintenance tasks, preserving system consistency, and facilitating the reuse of application code across different EC frameworks. The remainder of this paper is organized as follows. In Section 2 we review background and related work in evolutionary computation, object-oriented frameworks, and aspect-oriented programming. We then present the Arachne approach in Section 3, followed by the set of experiments performed for this study in Section 4. Lastly, we summarize our work, discuss our findings, and present future directions for this work in Section 5.

2. BACKGROUND & RELATED WORK

In addition to related work, this section presents three topics fundamental to this work: evolutionary-computation, object-oriented frameworks, and aspect-oriented programming.

2.1 Background

In this subsection, we first describe evolutionary computation and how it searches for optimal or near-optimal solutions. Then we introduce object-oriented frameworks and some of the challenges in designing and reusing EC frameworks in particular. Lastly, we describe how AOP separates crosscutting concerns at the implementation level.

Evolutionary Computation. Evolutionary approaches, such as genetic algorithms (GAs) [10], are stochastic search-based techniques for solving complex optimization problems. Evolutionary computation (EC) has been successfully applied in various fields, such as software engineering [8, 11, 24], antenna design, scheduling, and robotics [16]. In general, an evolutionary algorithm begins by generating a *population*, or collection, of random candidate solutions, or *individuals*. Next, a *fitness function* is applied to evaluate each individual and assign a scalar value proportional to the individual's quality. The evolutionary algorithm then uses this fitness value to *select* those individuals that will survive onto the next *generation*, or iteration, of the search process. To form new solutions, evolutionary algorithms typically apply some key operators such as crossover and mutation. The *crossover* operator recombines parts of two existing solutions in the population into a new solution. In contrast, the *mutation* operator randomly changes parts of an encoded solution. By applying these two operators, an evolutionary algorithm is able to balance exploitation and exploration of the solution space by combining parts of existing solutions in promising areas of the solution space while also exploring new areas that might contain different and useful solutions, respectively. This iterative process is repeated until either a satisfactory solution is found, or the maximum number of generations is exhausted. Typically, the individual with the highest fitness value is then returned as the output of the search process.

As an example, consider the Knapsack Problem [7], an NP-complete problem that arises in domains such as resource allocation and cryptography. The problem is defined as follows: a set of items, each with a non-negative weight and cost, are to be placed into a knapsack that has a finite storage capacity. The objective is to maximize the total cost of the items in the knapsack while ensuring that the total weight of the items does not exceed its storage capacity. One possible encoding for the Knapsack problem is a vector of boolean variables that specifies whether the i^{th} item is included in the knapsack or not. With this representation, each variable with a true value in the vector represents an item that is included in the knapsack. Similarly, a fitness function can be defined to assign a fitness value proportional to the total cost of the items in the knapsack, as long as the weight of those items does not exceed the knapsack constraints. The crossover operator then takes two different vectors of items and creates a new vector by combining parts of the two existing solutions. The mutation operator, on the other hand, accepts a vector of items and randomly adds or removes an item from the vector by changing a boolean variable. This process is repeated until the maximum number of generations is exhausted, or a high-quality solution is found.

Frameworks. Object-oriented frameworks [6] provide reusable design and code that can be customized for specific application domains, as well as to generate families of applications. Frameworks comprise both "frozen spots" that are portions of the design hierarchy and code that cannot be changed, and "hot spots" that refer to the points in the framework that the application developer can extend and include in their application code. Frameworks are also characterized by inversion of control, or call-backs, where the framework itself controls a program's flow of control. In particular, developers implement application functionality that is then called by the framework's code. In general, two main types of frameworks exist, black-box and white-box frameworks. Black-box frameworks are typically easier to use and provide well-defined interfaces for developers to call from their application code, thus hiding the framework's internal details from the application developer. White-box frameworks, on the other hand, are meant to be customized and extended by developers with application-specific code. As a result, white-box frameworks are more flexible than black-box frameworks, yet they require developers to become familiar with the internals before reuse is possible.

Several object-oriented frameworks, such as ECJ [18] and JGAP [20], are widely available and used to support the development of EC programs [5]. A key difference between EC frameworks is how generic and extensible the framework is for encoding or representing solutions in different application domains without requiring application developers to extend many classes in the framework. While several EC frameworks do provide generic representations that facilitate the implementation of EC programs across different application domains, ultimately it has not been possible to build a black-box EC framework as developers must always override some base functionality in the framework, such as the specific encoding of the problem or the fitness function used to evaluate the quality of a candidate solution [5]. In practice, developers typically reuse EC frameworks by customizing the configuration of the evolutionary algorithm, and the encoding, decoding, fitness evaluation, and output of candidate solutions [1, 5]. As a result, developers must often become familiar with the design and implementation of an EC framework, which may comprise over 50,000 lines of code.

Aspect-Oriented Programming. In object-oriented programming, code scattering may arise when a single concern (e.g., logging) is implemented throughout multiple modules [15]. AOP, in contrast, implements individual concerns by extracting them into *aspects* that are separate from the rest of the system [14]. *Weaving* is the process of composing individual aspects into a single system. Weaving rules are defined separately from aspects and applied systematically by an aspect compiler to the crosscutting modules in order to produce a self-contained executable program. *Join points* [14] are well-defined points in the execution flow of a program at which crosscutting functionality can be woven. *Pointcuts* identify a specific set of join points by applying regular expressions and pattern matching to filter parts of the program that match a certain signature. An *advice* defines additional code that will run at corresponding join points. The advice code can be executed before or after a join point, or even replace the code in the join point entirely. An *aspect* encapsulates these crosscutting concerns into a discrete module and comprises methods, fields, pointcuts, and advice.

2.2 Related Work

While, to the best of our knowledge, no other approaches have explicitly addressed the introduction of aspects into EC frameworks, in this subsection we overview a few approaches that have addressed the problem of modularizing object-oriented frameworks and programs.

Aspect-oriented development of frameworks. Recently, various aspect-oriented approaches [17, 22] have been proposed to facilitate the design, implementation, and extension of object-oriented frameworks. In particular, Kulesza *et al.* [17] introduced the concepts of extension join points and extension aspects. An extension join point exposes the hot spots in an object-oriented framework where developers may introduce new crosscutting functionality, either by selecting optional features or extending classes provided by the framework. Extension aspects encapsulate the new crosscutting functionality, thereby separating the core framework modules from the application-specific concerns until both are woven together during compilation. In a similar approach, Santos [22] *et al.* proposed the concept of specialization aspects as a means to modularize a framework's hot-spots. In their approach, a specialization aspect is an abstract aspect that defines the corresponding hot-spots in a framework. These specialization aspects can then be redefined and composed with other aspects in order to implement the necessary code that supports a particular framework feature within a single module. Both of these approaches can be leveraged to develop *new* aspect-oriented EC frameworks that mitigate the entanglement of application code within the framework implementation. In contrast, the Arachne approach presented in this paper focuses on separating crosscutting concerns in the application code from *existing* and widely used EC frameworks that were implemented without aspect-orientation.

Mixins and Traits. Two solutions have been proposed to separate features or units of code into discrete modules: mixins [2] and traits [23]. Mixins provide a light-weight, inheritance-based mechanism for reusing the same methods and fields in multiple classes. Mixins differ from traditional multiple inheritance because they are not intended for instantiation, but instead enable developers to combine and collect related functionality implemented in different mixins. Traits comprise a set of methods and can be used as building blocks for classes, but they are not classes themselves and may not be inherited. Rather than using inheritance as the composition operator, traits can be composed using symmetric sum, method exclusion, and other operators. Although Arachne separates functionality into isolated modules to streamline development, it differs from the traditional use of mixins. Specifically, Arachne separates the most commonly revised components of EC frameworks into aspects, but it does not create separate class containers for the code in the aspects. Instead, the class declarations remain apart of the framework code and pointcuts are inserted into the bodies of any class methods that are important for each component. As such, the finer-grained pointcuts in Arachne reduce the number of modifications required to the framework code as compared to mixins, thereby simplifying the approach and minimizing the effort needed to use Arachne with a new framework.

3. THE ARACHNE APPROACH

This section describes how Arachne encapsulates application-specific concerns into aspects such that the number of touch points (e.g., references or function calls) between the aspects and the EC framework is minimized. First, we present an example EC program that was implemented by extending an EC framework to illustrate how application-specific code crosscuts different modules in the EC framework, thus hampering its maintainability and understandability. We then present the Arachne approach in detail as we apply it to the example program. Next, we present a prototype tool that supports the Arachne approach by providing partial aspect code generation. Lastly, we compare the resulting aspect-oriented implementation to the original implementation of the EC program.

3.1 Crosscutting Application Logic in Evolutionary Computation Frameworks

In EC programs, a key design decision for developers is to determine how to represent, or encode, the general structure of a candidate solution in classes and data structures (e.g., an integer vector) in the EC framework. This *encoding* process explicitly maps a solution in the application domain to an individual in the evolutionary algorithm, thus preserving the semantics of evolutionary operators such as crossover and mutation. Recalling the Knapsack Problem example introduced earlier, a solution in the application domain is a knapsack that contains a set of items. When the solution is encoded into an individual in the evolutionary algorithm, only the details that are relevant to the problem (i.e., the capacity of the knapsack, and the weight and cost of each item) are encoded. Similarly, after the evolutionary algorithm modifies an encoded individual using its mutation and crossover operators and evaluates the individual's fitness, the algorithm must decode the individual back to a solution in the application domain. As a result, EC frameworks concerns are entangled with application-specific code that defines how candidate solutions should be encoded, decoded, and evaluated [5].

Next, we illustrate how application-specific code may become entangled and scattered throughout different portions of an EC framework. We use an implementation of the Knapsack Problem that is distributed as an application exemplar with the Java Genetic Algorithms Package (JGAP) [20]. In this implementation, Knapsack accepts a set of items, each with a weight and value, as inputs, and outputs the items and quantities to include in a collection such that the total value is maximized without exceeding a pre-specified maximum weight constraint.

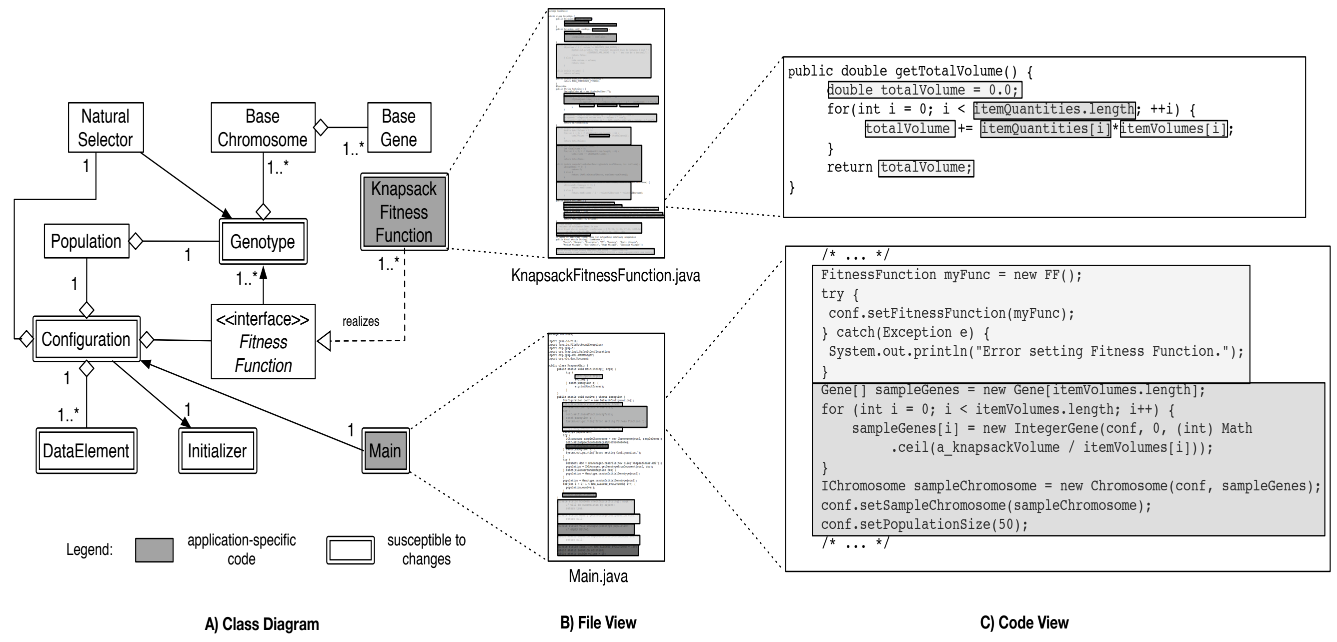
Figure 1A) presents a UML class diagram that shows the classes in the implementation and relationships between them. In this class diagram, shaded classes denote application-specific code, arrows denote inter-class relationships, and double boxes denote classes that may be affected if the Knapsack encoding needs to be modified. For instance, two user-defined classes extend the JGAP framework with the Knapsack application code: Main and KnapsackFitnessFunction. Main configures and controls the execution of the genetic algorithm, and it also specifies how to encode candidate solutions within JGAP. In particular, Main defines a vector of items and quantities as the structure for candidate solutions to the Knapsack optimization problem. KnapsackFitnessFunction, on the other hand, realizes the FitnessFunction interface as required by JGAP. When invoked by the genetic algorithm to evaluate a candidate solution, KnapsackFitnessFunction iterates through the vector of items and computes the total weight and value of the encoded solution to the Knapsack Problem.

In this approach, the encoding and evaluation concerns of an EC framework are spread over multiple extension points, thereby causing consistency problems when the application-specific code changes. In particular, Figure 1B) gives a File View representation of the two classes, KnapsackFitnessFunction and Main, where differently shaded boxes represent different EC framework concerns instantiated with application-specific information, such as encoding, decoding, evaluation, and so forth. As this figure illustrates, Main entangles several EC framework concerns with application-specific code, all within a single monolithic class. This entanglement between the application code and the JGAP framework complicates maintenance tasks, such as modifying a candidate solution's encoding, as developers need to sift through different concerns to identify where code needs to be changed or updated. Similarly, Figure 1C) presents two code snippets from these classes (Code View), where each EC framework concern is shaded with a gray box. As this figure illustrates, application-specific code is duplicated and scattered across both KnapsackFitnessFunction and Main. Specifically, KnapsackFitnessFunction comprises application-specific code based on the encoding and semantics of a candidate solution as defined in the implementation of Main (i.e., associating a cell in the vector with a particular volume).

While it seems that changes to the application code would only affect Main and KnapsackFitnessFunction, it turns out that several other classes in the JGAP framework are also susceptible to change, as indicated by double-boxed classes in Figure 1. For instance, if the encoding is modified, then in addition to revising both Main and KnapsackFitnessFunction, developers may also need to extend or customize the following classes: Initializer and Configuration (both for configuring JGAP), Genotype (for specifying a new type of encoding and decoding), and DataType (for formatting output data). Given that most EC programs are significantly more complex than this example, it is desirable to minimize dependencies between the application code and the EC framework.

3.2 Arachne Approach

The objective of Arachne is to minimize the amount of application-code that is scattered and entangled across various crosscutting concerns in an EC framework. To achieve this objective, Arachne encapsulates application-specific code within aspects that can then be woven into the corresponding EC framework extension points at compile



```
public double getTotalVolume() {
     double totalVolume = 0.0;
     for(int i = 0; i < itemQuantities.length; ++i) {
          totalVolume += itemQuantities[i]*itemVolumes[i];
     }
     return totalVolume;
}
```

```
     /* ... */
FitnessFunction myFunc = new FF();
try {
 conf.setFitnessFunction(myFunc);
} catch(Exception e) {
 System.out.println("Error setting Fitness Function.");
}
Gene[] sampleGenes = new Gene[itemVolumes.length];
for (int i = 0; i < itemVolumes.length; i++) {
     sampleGenes[i] = new IntegerGene(conf, 0, (int) Math
          .ceil(a_knapsackVolume / itemVolumes[i]));
}
IChromosome sampleChromosome = new Chromosome(conf, sampleGenes);
conf.setSampleChromosome(sampleChromosome);
conf.setPopulationSize(50);
     /* ... */
```

Figure 1: Knapsack implementation with JGAP.

time. Thus, a key part of the Arachne approach is to identify join points in a target EC framework where application-specific code needs to be inserted during compilation. In general, these join points are related to the encoding, decoding, and fitness evaluation of candidate solutions, as well as the configuration of the evolutionary algorithm and the output of data [1, 5]. Next, we illustrate the Arachne process by manually applying it, step by step, to the Knapsack problem. In particular, the data flow diagram presented in Figure 2 overviews the key steps of Arachne. As this figure illustrates, developers must first define an aspect to map the structure of a candidate solution to a specific encoding or data structure in the target EC framework. Next, developers must define an aspect to evaluate the fitness of candidate solutions. Lastly, developers must implement an aspect to configure the evolutionary algorithm provided by the EC framework and format the output of data. Aspect weavers, such as AspectJ [9, 15], integrate the application code into the specified pointcuts in the target EC framework, thereby producing a self-contained executable application.

The following steps illustrate the Arachne approach by applying it to the GA-based Knapsack exemplar provided by the JGAP framework [20]. We omit some of the implementation details due to space constraints. Finally, we leverage the AspectJ [9, 15] language extension to illustrate the AOP portion of Arachne. The specific syntax and idioms for applying Arachne may vary depending on the AOP language support selected.

1. Encode/Decode Aspect. *Create an aspect to define how to represent candidate solutions in the EC framework.*

The Encoding/Decoding aspect comprises a set of pointcuts and advice to specify the encoding or representation of candidate solutions in the target EC framework, and vice-versa. In this step, three subtasks are performed: (a) identify join points in the target EC framework where candidate solutions are mapped to specific data structures provided by the framework; (b) define pointcuts to override the functionality implemented by the framework at these join points; (c) implement an advice to map the structure of a candidate solution to a specific encoding or data structure compatible with the target EC framework. In particular, this advice should return an instance of a base encoding class provided by the EC framework.

In addition to encoding candidate solutions within the EC framework, the Encoding/Decoding aspect must also map encoded solutions back to the application domain using a

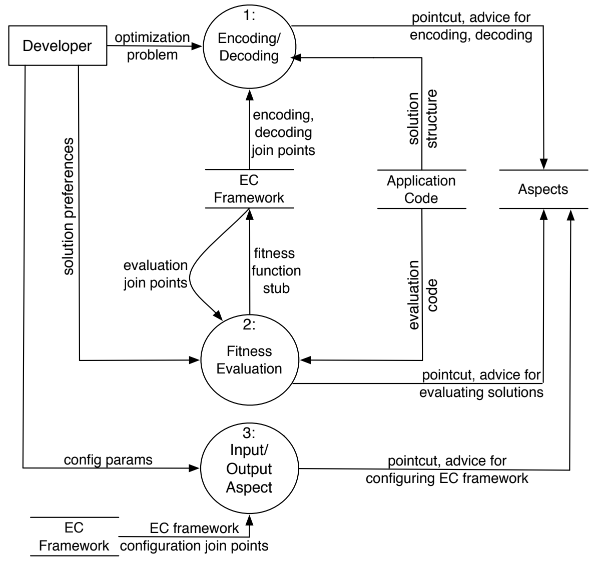


Figure 2: Data Flow Diagram describing the Arachne approach.

similar strategy. This decoding operation is important as it enables a fitness function to evaluate the quality of an encoded solution, as well as the EC framework to output the solution at the end of the evolutionary process. To accomplish this objective, Arachne requires developers to (a′) identify join points in the target EC framework where individuals are *decoded.* Next, (b′) define pointcuts that match and override these join points. And, finally, (c′) implement an advice to map an individual back to a solution in the application domain. In particular, this advice should return an instance of a candidate solution independent of any EC framework.

Example. In the Knapsack example, we first (a) identified join points in the JGAP framework where candidate solutions were encoded. Specifically, candidate solutions in JGAP must be mapped to instances of the Gene class, and then associated with a Configuration class that integrates the particular encoding with other classes in the framework. We then (b) defined a pointcut to match and override these join points with our application code. In particular, each candidate solution in Knapsack comprises a vector of items, values, and quantities. As such, we then (c) implemented an advice that mapped this vector to an array of Gene instances. Next, for the decoding we (a′) identified join points in the JGAP framework where candidate solutions were decoded. In general, JGAP decodes candidate solutions before computing their fitness values and before outputting the result of the evolutionary process. Next, we defined pointcuts to match and override these join points (b′). Lastly, we defined an advice to map IChromosomes, a wrapper for Gene objects, to a vector of items, values, and quantities (step c′). The decoding advice then uses this vector to instantiate a new Knapsack object and return it to the calling function. The code snippet shown in Figure 3 illustrates the encoding and decoding implementation in the Encoding/Decoding aspect.

```
 // (b)
pointcut encode(Configuration conf) :
call(* Main.getEncoding(Configuration)) &&
     args(conf);

// (c)
Gene[] around(Configuration conf) : encode(
     conf) {
   int len = Knapsack.items.length;
   Gene[] genes = new Gene[len];
   for(int i = 0; i < len; ++i) {
      genes[i] = new IntegerGene(conf,
        0, (Knapsack.solution.volume() /
             Knapsack.items[i]));
   }
   return genes;
}

// (b')
pointcut decode(IChromosome s) :
call(Knapsack Main.mapToSol(IChromosome)) &&
      args(s);

// (c')
Knapsack around(IChromosome s): decode(s) {
   int[] config = new int[s.getGenes().
        length];
   for(int i = 0; i < config.length; ++i) {
      config[i] = s.getGene(i).getAllele().
           intValue();
   }
   return new Knapsack(config, Knapsack.
        volume);
}
```

Figure 3: Encoding/Decoding Aspect

2. Fitness Evaluation. *Create an aspect to evaluate the quality of candidate solutions.*

In general, EC frameworks require developers to implement fitness functions that evaluate the quality of a candidate solution. Arachne is no different. However, instead of extending the EC framework with application-specific evaluation code, Arachne requires developers to implement the evaluation code separate from the EC framework. In particular, the evaluation code should be implemented as part of the classes that define the optimization problem model (i.e., the application logic). Note that this constraint does not require any additional implementation effort from developers as they would also have to implement the evaluation criteria within some extension point of the EC framework. However, moving the evaluation criteria to classes that define the optimization problem eliminates the scattering of application-specific code within different extension points of the EC framework (as shown in Figure 1C).

Once the evaluation criteria is implemented separately from the EC framework, developers should define an EvaluationAspect. This aspect should comprise a single pointcut and an advice that overrides the evaluation of individuals in the target EC framework. This requires developers to identify and specify join points and pointcuts where individuals are evaluated and fitness values are assigned. Lastly, developers must implement an advice to decode the candidate solution (refer to Step 1 for the Encoding/Decoding aspect that provides this functionality), invoke that evaluation code, and return the computed fitness value.

Example. In the Knapsack example, we implemented the evaluation criteria as part of the Knapsack class, which defines the general structure of a solution independently of any EC framework. In addition, we also had to extend the JGAP framework by implementing KnapsackFitnessFunction, a stub class that realizes the FitnessFunction interface as required by JGAP. Note that this class only declares a function stub, as required by the JGAP framework, and is not susceptible to changes in the application code as it contains no evaluation criteria. Next, we identified join points in the JGAP framework where fitness values are computed, such as the evaluation function that must be implemented by KnapsackFitnessFunction as part of the FitnessFunction interface. We then defined a pointcut to match the empty evaluation function in KnapsackFitnessFunction. Lastly, we implemented an advice that decoded an individual (see Encoding/DecodingAspect), invoked the evaluation code in Knapsack, and returned the fitness value back to the original evaluation method such that the evolutionary algorithm can leverage this fitness information to guide the search process towards better solutions. Figure 4 presents a code snippet of the EvaluationAspect.

3. Input/Output Aspect. *Create an aspect to configure the evolutionary algorithm and format the output data.*

The InputOutput aspect comprises a set of pointcuts and advice to configure the evolutionary algorithm and format the output data generated throughout the evolutionary process. First, developers must identify join points and specify pointcuts in the EC framework where either the evolutionary algorithm is configured (i.e., population size) or data

```
pointcut evaluate(IChromosome a_subj) :
call (private * FF.evaluate(IChromosome)) &&
      args(a_subj);

double around(IChromosome a_subj) :
   evaluate(a_subj) {
   Solution candidate =
      Main.mapToSol(a_subj);
   return candidate.evaluate();
}
```

Figure 4: Evaluation Aspect

is output. In general, EC frameworks initialize the evolutionary algorithm immediately before the evolutionary algorithm begins its search process. Similarly, EC frameworks typically output the solution with highest fitness value immediately after the evolutionary algorithm terminates. As a result, developers must define an advice that configures the various parameters in the evolutionary algorithm, as well as formats the output data as necessary. Note that not every EC program will require the formatting of output data, especially if the EC framework already provides some basic functionality for this task.

Example. In the Knapsack example, we defined a pointcut to match the configuration of the genetic algorithm, as well as format the output of solutions. Specifically, in JGAP an evolutionary algorithm is configured by invoking methods in the Configuration class. Similarly, JGAP is typically configured to output the solution with highest fitness value in the population once the evolutionary algorithm terminates. Next, we implemented two advice methods to configure the genetic algorithm and format the output data, respectively. In particular, the advice that configures the evolutionary algorithm initializes the Configuration object and then returns it to the JGAP framework. Similarly, the advice that formats the output data first decodes the individual with highest fitness value (the decoding is performed by the Encoding/Decoding aspect) and then invokes a print method in Knapsack to display the evolved solution. The code snippet in Figure 5 overviews the output formatting in Input/OuputAspect. Due to space constraints we omit the advice for configuring the genetic algorithm.

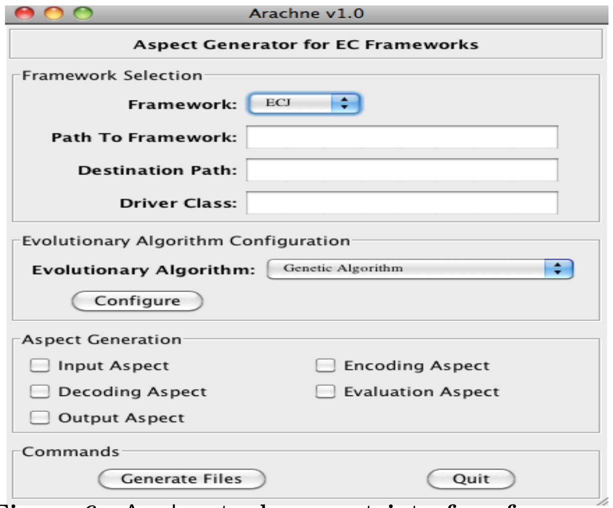
```
pointcut output(Genotype pop) :
call (private * Main.doOutput(Genotype))
     && args(pop);

void around(Genotype pop) : output(pop)
{
   best = pop.getFittestChromosome();
   System.out.println(
      Main.mapToSol(best));
}
```

Figure 5: Input/Output Aspect

3.3 Arachne Tool Support

Arachne requires developers to implement several aspects in order to maintain application-specific code separate from the target EC framework until compile time. Each aspect corresponds to a single concern in the EC framework, such as solution encoding and fitness computation. Within these aspects, developers specify the corresponding join points and pointcuts for the target EC framework where an aspect-weave will insert the corresponding instances of the application logic. Unless the application programming interface or design hierarchy of a target EC framework is modified, these join points, pointcuts, and portions of the advice can be reused for other applications. As a result, we have implemented a prototype tool that supports the partial generation of aspect code for specific target EC frameworks, thereby reducing the effort required from developers to apply Arachne. Currently, our prototype tool supports the generation of partial aspect code for the JGAP [20] and ECJ [18] frameworks. Additional EC frameworks can be supported in the future.

The interface for the prototype tool we implemented, shown in Figure 6, enables developers to choose a target EC framework, configure the parameters of the evolutionary algorithm (e.g., population size), and select the corresponding aspects to be generated. We note, however, that the prototype tool does not support complete aspect code generation. Specifically, developers must still provide some details, such as how a candidate solution should be encoded in a target EC framework. Nonetheless, this aspect code generator enables developers to reuse all join points and pointcuts for a target EC framework, as well as some of the advice code that is EC framework-specific (e.g., configuring the evolutionary algorithm). In addition, this prototype tool supports the generation of required EC framework-specific constructs, such as the fitness function stub that must *realize* the FitnessFunction interface in JGAP. In this manner, this prototype tool enables developers to focus on designing and implementing the application logic instead of dealing with internal details of a particular EC framework. To this end, we have applied this prototype tool to support the use of both JGAP and ECJ.


Arachne v1.0
Aspect Generator for EC Frameworks
Framework Selection
Framework: ECJ
Path To Framework:
Destination Path:
Driver Class:
Evolutionary Algorithm Configuration
Evolutionary Algorithm: Genetic Algorithm

Aspect Generation
☐ Input Aspect ☐ Encoding Aspect
☐ Decoding Aspect ☐ Evaluation Aspect
☐ Output Aspect
Commands

Figure 6: Arachne tool support interface for aspect code generation.

3.4 Resulting AO EC Application

Arachne facilitates application-specific and application-independent code reuse in EC programs built with the support of EC frameworks. Application-specific reuse typically occurs when implementing or migrating an EC program to a different target EC framework. In particular, Arachne facilitates the application-specific reuse of the solution structure (e.g., classes that define what is being optimized), fitness functions, and output data formatting implementations. Application-independent reuse typically occurs when implementing different EC programs with the support of the same target EC framework. Specifically, Arachne, and in particular the prototype aspect generator tool support for Arachne, facilitates the application-independent reuse of the join points and pointcuts in the EC framework, as well as the advice that specifies the evolutionary algorithm configuration. For instance, 294 (out of 340) lines of code, comprising join points, pointcuts, and advice, were directly reused by leveraging the prototype aspect code generator to implement the previously described Knapsack application. Building upon these 294 lines of code, only another 46 lines of code were required to implement the full application example. Note that it is also possible to leverage Arachne to facilitate the reuse of both application-specific and application-independent code, such as when migrating existing EC programs to a target EC framework supported by the Arachne aspect code generator.

The resulting aspect-oriented Knapsack application implemented with JGAP is shown in Figure 7. In this figure, shaded elements comprise Knapsack-specific application code, dashed boxes represent aspects introduced by Arachne, and double boxes represent classes susceptible to changes if the application code is modified. One class (Main) and one aspect (Encoding/Decoding) are expanded to show how the aspects are distributed in each file, and each file is further expanded to highlight the Encoding joinpoint and its respective advice that will be woven in at compile time. As this figure illustrates, the application-specific code is modularized into a self-contained class, such as Knapsack, or into corresponding aspects that connect Knapsack to the JGAP framework, such as Encoding/DecodingAspect. We note that the Evaluation and Input/Output aspects defer the fitness evaluation and output data formatting to Knapsack and thus are not shaded in Figure 7 since they contain no application-specific code. Lastly, by separating concerns from the EC framework, changes to the application code are localized to the Knapsack class and the Encoding/Decoding aspect.

Applying Arachne to the design of Knapsack facilitated the reuse of application-specific code across both the JGAP and ECJ frameworks. In particular, no modifications were required to integrate the Knapsack class with either JGAP or ECJ. As a result, we were able to reuse the Knapsack solution definition, evaluation criteria, and data output formatting implementations across both EC frameworks; the reused components are, respectively, the Knapsack class, Evaluation aspect, and Input/Output aspect. In terms of application-independent code reuse, neither the encoding nor pointcuts in ECJ matched those from JGAP. However, by leveraging the prototype tool support for Arachne, we were able to reuse the ECJ-specific join points, pointcuts, and evolutionary algorithm configuration advice, thus only requiring developers to select an appropriate encoding for the Knapsack problem in the ECJ framework.

4. CASE STUDY

This section presents two sets of experiments where we apply Arachne to re-implement EC programs by using aspects to group together crosscutting concerns in the application code. These experiments enable us to evaluate how general the Arachne process is, even when applied to different EC programs and frameworks. In addition, these experiments also enable us to assess how aspect-orientation affects the reuse and maintenance of the application code once it has been grouped into aspects. Throughout each experiment we use the AspectJ [9, 14] extension to support AOP for Java.

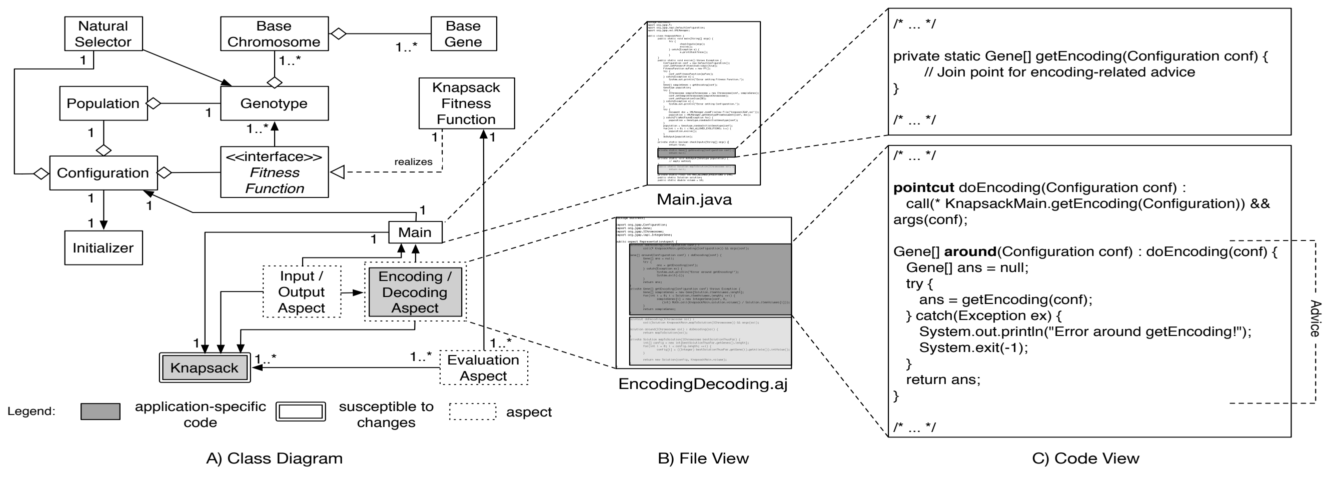
4.1 Reusing functional logic across different evolutionary frameworks

The main objective of this experiment is to explore the application-specific and application-independent code reuse across different target EC frameworks. First, we *extract* the application code from a set of exemplar applications previously implemented and distributed with the JGAP [20] and ECJ [18] frameworks. We then apply Arachne to modularize the application code. Lastly, we use the resulting aspect-oriented implementation to *migrate* the extracted application code between the two EC frameworks. That is, we migrated a set of exemplars from JGAP to ECJ, and a different set of exemplars from ECJ to JGAP. Although it is typically undesirable to migrate an application between frameworks, this migration might be necessary in EC-based programs to either improve the EC process or incorporate broader sets of evolutionary algorithms and techniques [5]. As such, the primary focus of this experiment is to determine whether aspect-orientation facilitates the migration process between two different EC frameworks while enabling reuse of application code.

In each exemplar application, a genetic algorithm [10] efficiently searches for optimal or near-optimal solutions to NP-Complete optimization problems, such as Knapsack and Traveling Salesman Problem [7]; variants of these optimization problems arise frequently in many application domains. While these exemplars illustrate how to use the corresponding EC frameworks in similar types of applications, they also indirectly promote the entanglement of crosscutting concerns across the underlying EC framework.

A key step in the migration process is to extract the corresponding application code from each exemplar such that it may then be ported to a different target EC framework (i.e., from JGAP to ECJ). We first identified classes that either extended or customized the hierarchy or behavior of the underlying EC framework. We then located and extracted code within these classes responsible for configuring the evolutionary algorithm, as well as encoding, decoding, evaluating the fitness, and outputting of candidate solutions. While this extraction step is nontrivial, the alternative of rewriting the application code and integrating it with the target EC framework merely perpetuates the entanglement of crosscutting concerns across the EC framework. To facilitate the migration efforts, we leveraged the Arachne prototype tool support and partially generated aspect code for Encoding/Decoding, fitness evaluation, and Input/Output for the target EC framework, ECJ in this case. While we had to manually map the encoding or representation of candidate solutions from one EC framework to another (i.e., from JGAP classes to ECJ classes), by using the prototype tool we were able to directly reuse 252 lines of code comprising all join points and pointcut definitions, as well as advice code specific to configuring the ECJ framework.

The migration process was successful for each exemplar. In terms of application-specific reuse, we were able to reuse the solution encoding, fitness evaluation criteria, and output data formatting implementation for each exemplar that we migrated to a different EC framework. In terms of application-independent reuse, we were able to leverage the Arachne aspect generation tool to reuse the join points, pointcuts, and genetic algorithm configuration advice for all exemplars being ported to either the JGAP or ECJ frameworks. In this manner, the Arachne approach and tool



```
/* ... */

private static Gene[] getEncoding(Configuration conf) {
      // Join point for encoding-related advice
}

/* ... */
```

```
/* ... */

pointcut doEncoding(Configuration conf) :
  call(* KnapsackMain.getEncoding(Configuration)) &&
args(conf);

Gene[] around(Configuration conf) : doEncoding(conf) {
  Gene[] ans = null;
  try {
    ans = getEncoding(conf);
  } catch(Exception ex) {
    System.out.println("Error around getEncoding!");
    System.exit(-1);
  }
  return ans;
}

/* ... */
```

Figure 7: Aspect-oriented implementation of Knapsack in JGAP obtained by applying Arachne**.**

support enabled us to reuse both application-specific and application-independent code, thereby facilitating the overall migration process.

4.2 Plato Implementation with EC Frameworks

In this experiment, we apply Arachne to re-engineer Plato [21], an EC program developed for an industry-provided problem; we re-engineer one version with JGAP and another with ECJ. Plato is a decision-making engine for autonomic [13] and self-adaptive systems [19] that applies genetic algorithms to generate system configurations that balance competing functional and non-functional requirements in response to current system and environmental conditions. Specifically, Plato generates candidate network configurations for diffusing large amounts of data between remote data mirrors [12] (data centers for replicating and protecting data against failures and network outages) with the primary objective of minimizing operational costs while maximizing network performance and data reliability. First, we extract the application code from the corresponding EC framework implementation. We then apply the Arachne approach, with the aid of the prototype aspect generation tool, to modularize and separate the crosscutting concerns in Plato from the EC frameworks. Finally, we compare and analyze the resulting aspect-oriented implementation with its object-oriented implementation.

Encoding. Plato represents each candidate network configuration as a complete graph, where each edge is active or inactive, and associated with one of seven different methods for distributing data across the network. A set of custom encodings is required to represent candidate network configurations as individuals in JGAP. As Figure 8 illustrates, these custom encodings are implemented by extending and integrating several classes provided by JGAP. For example, ActiveGene and PropGene extend BaseGene to encode the operational status and data diffusion method for a network link, respectively. LinkAllele is used to represent a single edge in the complete graph, and thus comprises one instance of ActiveGene and PropGene each. BaseChromosome defines a candidate network configuration to be a vector of LinkAlleles, one for each edge in the complete graph. Similarly, NetworkGenotype extends Genotype with additional functionality to represent candidate network solutions as individuals in the population.

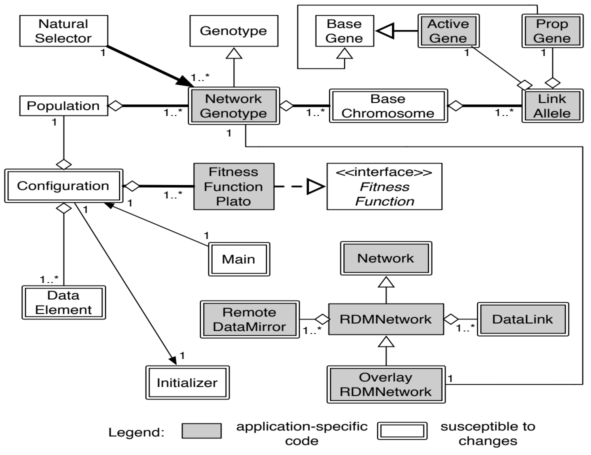


Figure 8: Object-oriented implementation of Plato with JGAP framework.

Fitness Functions. In addition to customizing the JGAP framework with network-specific representations, Plato also leveraged a set of fitness functions to evaluate the quality of candidate solutions in terms of operational costs, network performance, and data reliability. As Figure 8 shows, FitnessFunctionPlato realizes the FitnessFunction interface as required by JGAP. To evaluate a candidate network configuration, FitnessFunctionPlato must first map a NetworkGenotype to an OverlayRDMNetwork class. In this decoding operation, FitnessFunctionPlato iterates through the vector of LinkAlleles stored in the BaseChromosome, and then extracts each link configuration. FitnessFunctionPlato then constructs an overlay network that only comprises active links in the graph. The resulting overlay network is evaluated, and solutions that best balance competing objectives obtain a higher fitness value.

Maintenance Challenges. While the Plato implementation reused significant amounts of code from the JGAP framework, the Plato-specific extensions to the JGAP framework may complicate common maintenance tasks. For instance, the shaded classes in Figure 8 illustrate that the application code that defines the network-specific encoding is scattered across various classes that extend JGAP. (The touch points between the application and framework classes are denoted by bold connecting arrows.) The dependencies between the network model and the network-specific encodings and evaluation criteria complicate the task of preserving system consistency while performing even simple modifications to the application code, as this implies propagating corresponding changes across multiple files. Furthermore, double-boxed classes in Figure 8 are also susceptible to major modifications in the application code. In either case, developers must identify the files and the corresponding lines of code affected by changes in the application-specific code. Moreover, it may be undesirable to implement the network-specific encoding and evaluation criteria by extending the JGAP framework, as it tends to introduce code-bloat that is unlikely to be reused in different application domains [5].

4.3 Applying Arachne to Plato

We applied Arachne to separate the crosscutting application code in Plato from the JGAP framework, and we call the resulting program Plato-JGAP-AO for brevity. Figure 9 shows the resulting AO implementation, where shaded classes comprise application code for modeling and representing networks of remote data mirrors and dashed classes represent aspects that modularize the crosscutting application code. As this figure illustrates, the network model is now completely separate from the JGAP framework. Moreover, the Input/Output and Encoding/Decoding aspects are responsible for mapping the general structure of candidate network configurations to individuals in JGAP. Similarly, the Evaluation aspect overrides the FitnessFunctionPlato class, which merely realizes the FitnessFunction interface yet provides no functionality whatsoever, with the evaluation criteria implemented within OverlayRDMNetwork. By leveraging AspectJ [9, 15] we were able to integrate the revised network model with the JGAP framework and thus leverage genetic algorithms to generate candidate network configurations.

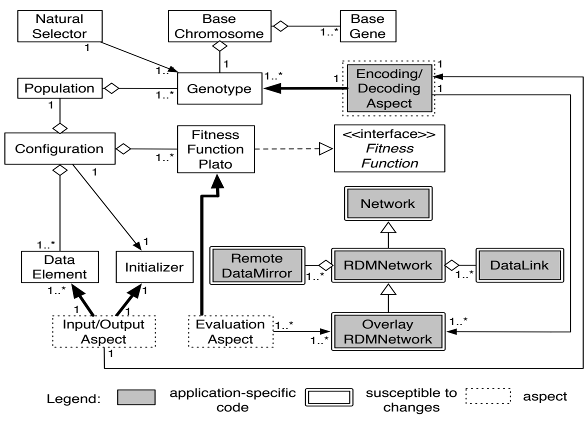


Figure 9: Implementation of Plato-JGAP-AO.

Facilitating Maintenance. The resulting aspect-oriented implementation also modularizes the application code into classes and aspects according to their functional objective, including encoding, decoding, fitness evaluation, and output of candidate network configurations. In particular, the encoding of candidate network configurations to JGAP-specific data structures is implemented solely within the Encoding/Decoding aspect. For instance, only the Encoding/Decoding aspect required modification to change the encoding of data diffusion methods in Plato-JGAP-AO. In contrast, to change the same data diffusion method encoding in the OO-based implementation, several classes required modification, including LinkAllele, PropGene, and NetworkGenotype (previously shown in Figure 8). In this manner, Arachne facilitates the modification of application code as aspect-oriented tools can automatically weave the corresponding changes into the JGAP framework, thereby preserving system consistency.

Reuse Across EC Frameworks. For comparison, we also applied Arachne to implement Plato in the ECJ framework, which we call Plato-ECJ-AO for brevity. Figure 10 shows the resulting implementation. As the figure illustrates, the network model, shown in the shaded classes, is completely separate from the ECJ framework. The Encoding/Decoding aspect is responsible for mapping the general structure of candidate network configurations to individuals in ECJ. Similarly, the Evaluation aspect overrides the fitness evaluation logic in NetIndividual, and the Input/Output aspect overrides user input and the display of statistical information in the Main and Statistics classes, respectively. With the exception of NetIndividual, no other class extends or modifies the ECJ framework. Also note that we were able to reuse the Network, RDMNetwork, OverlayRDMNetwork, RemoteDataMirror, and DataLink classes from Plato-JGAP-AO without requiring any modifications. As such, reusing these classes enabled us to reuse the remote data mirror network model definition, set of fitness functions, and data output formatting implementation. In addition, we were also able to reuse the set of pointcuts previously identified in the first experiment for ECJ.

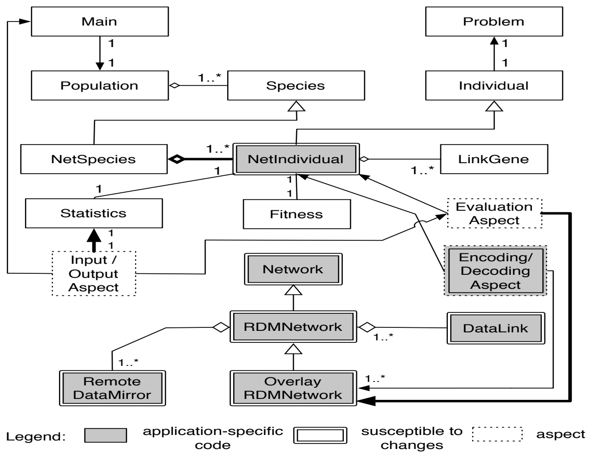


Figure 10: Implementation of Plato-ECJ-AO.

Metric-based Evaluation of Arachne Solution. We used the CK Metrics suite [4] and the AOP Metrics suite [3] to objectively measure the effects of the Arachne approach on the resulting implementations. These metrics were gathered to determine whether aspect-orientation was beneficial or detrimental to the resulting EC program implementation

Table 1: Object-oriented & AOP metrics for PlatoJ implementations in JGAP and ECJ frameworks.

Metric	PlatoJ-JGAP	PlatoJ-JGAP-AO	PlatoJ-ECJ	PlatoJ-ECJ-AO
Total Lines of Code (impl. code only)	1212	1262	1092	1165
Specialization Index	0.27	0.212	0.405	0.383
Instability	1	0.778	1	0.5
Number of Attributes	5.909	3.643	5.5	4.2
Number of Packages	1	2	1	2
Method Lines of Code	4.612	4.209	4.329	3.958
Weighted Methods Per Class	25.182	21.357	25.7	24
Number of Overridden Methods	1.818	1.429	2	2
Nested Block Depth	1.306	1.266	1.302	1.294
Lack of Cohesion of Methods	0.341	0.295	0.382	0.295
McCabe Cyclomatic Complexity	1.731	1.689	1.725	1.678
Number of Parameters	0.662	0.661	0.745	0.79
Efferent Coupling	6	5	6	3
Depth of Inheritance Tree	1.636	1.5	2	2

in terms of code reuse, flexibility, extensibility, and complexity, each of which affects future maintenance tasks. Table 1 shows the collected object-oriented and aspect-oriented metrics for both versions of PlatoJ, based on the JGAP and ECJ frameworks, respectively. A shaded cell in this table indicates a better metric value than its counterpart. In general, the resulting aspect-oriented implementations indicated improvements for the majority of metrics reported. In particular, the aspect-oriented metrics report a more modular design that, in turn, reduces complexity at the implementation level (as indicated by the *Method Lines of Code*, *Weighted Methods per Class*, *Nested Block Depth*, and *McCabe Cyclomatic Complexity* metrics). Moreover, by modularizing the crosscutting application-specific details, the corresponding classes and aspects increased in cohesion (as indicated by the Lack of Cohesion in Methods). These results suggest the aspect-oriented version of the EC programs are more focused in their implementation details, thereby easing maintenance tasks in the future.

While most metrics indicated improvements in the resulting aspect-oriented implementations, two particular metrics indicated slight detrimental effects. First, the total number of packages upon which the resulting EC programs depend increased from 1 to 2. This increase in package dependency is a direct result of importing the AspectJ support to weave in the separate application logic into the corresponding locations in the target EC framework. Second, the total number of lines of code also increased in each aspect-oriented implementation. These additional lines of code define the join points, pointcuts, and advice in each aspect, which would not be present in the object-oriented version. While it is typically undesirable to increase the total lines of code in a system as it implies greater maintenance efforts, in this case the Arachne tool support generated the majority of those lines of code automatically for the developer. Thus, while the size of the aspect-oriented EC program slightly increased because of Arachne, developers did not have to spend effort on implementing the majority of those additional lines.

5. CONCLUSIONS

We have presented Arachne, an aspect-oriented approach for developing EC programs. Arachne leverages AOP not only to separate crosscutting application code from its supporting EC framework, but also to automatically weave portions of the application code into their corresponding extension points in the target EC framework during compilation. To support the Arachne approach, we have also developed a prototype tool that generates aspect code, thereby encapsulating join points and pointcuts for different EC frameworks. Both Arachne and its supporting tool enable developers to focus on the implementation of the application code instead of learning details of the implementation and hierarchy of the EC framework. We demonstrated Arachne and its supporting tool by re-engineering a set of EC programs that include EC benchmarks and an industry-based EC program, previously implemented by extending and customizing two widely used EC frameworks, JGAP and ECJ.

Experimental results show aspect-orientation improves the design and quality of resulting EC programs. Differences between the object-oriented and aspect-oriented metrics collected from these experiments suggest that Arachne improves the modularity and cohesiveness of EC programs. In particular, resulting EC programs are better modularized because each part of the application code that interfaces with the EC framework is isolated into corresponding aspects. In practice, we interpret these metrics as indicative that the aspect-oriented code is more cohesive, focused, and easier to verify and validate, as shown by the Lack of Cohesion in Methods, Method Lines of Code, and McCabe Cyclomatic Complexity metrics, respectively. In this manner, Arachne facilitates identifying where to perform changes to the application code, and then automatically propagate these to their corresponding extension points in the EC framework during weaving. We also note that the majority of collected software engineering metrics suggest that aspect-orientation leads to an improved design and implementation of EC programs. However, a possible threat to validity in this study, and therefore to its results, is the small size of the programs being considered. A more thorough exploration of these effects needs to be carried out with additional EC implementations.

It is worth noting that Arachne did not cause any adverse side-effects upon the application behavior or performance in any of the experiments conducted for this work. The weaving process that inserts application code into corresponding extension points of an EC framework occurs at compile time, not at run time. Furthermore, even though

Arachne encapsulates the fitness evaluation into a dedicated module apart from the EC framework, the implementation for evaluating candidate solutions is identical in both the object-oriented and aspect-oriented versions. As a result, the aspect-oriented EC program does not suffer from degraded performance when compared to its object-oriented counterpart implementation.

While this study focused on object-oriented frameworks for building EC applications, we believe the Arachne approach is also applicable to other application-domain frameworks. Arachne targets a bidirectional dependency between the application code and its supporting framework. In the same spirit, Arachne should be applicable to object-oriented frameworks in other application domains that meet the following criteria. First, the object-oriented framework must provide a set of data structures or containers that can be combined or extended with the objective of representing, encoding, or storing the problem being solved within the framework. Second, the application logic must crosscut several classes in the representation or encoding extension points. Third, the internal details or semantics of various classes in the representation extension points must crosscut at least two other extension points in the object-oriented framework. In this manner, EC frameworks serve as unique and concrete examples of object-oriented frameworks where application code must be mapped into a set of framework classes in several extension points while, at the same time, the semantics of this mapping also crosscut other extension points in the EC framework.

Future directions include applying Arachne to re-engineer additional EC programs, as well as extending the set of supported EC frameworks. We are also exploring whether dynamic aspect weaving techniques may support the dynamic insertion and removal of fitness functions for EC programs, thereby enabling changes to how candidate solutions are evaluated in response to changing requirements.

6. REFERENCES

[1] T. Back, U. Hammel, and H.-P. Schwefel. Evolutionary computation: Comments on the history and current state. *IEEE Transactions on Evolutionary Computation*, 1:3–17, 1997.

[2] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 303–311. ACM, 1990.

[3] M. Ceccato and P. Tonella. Measuring the effects of software aspectization. In *First Workshop on Aspect Reverse Engineering*, 2004.

[4] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[5] C. Gagne and M. Parizeau. Genericity in evolutionary computation software tools: Principles and case-study. *International Journal on Artificial Intelligence Tools*, 15(2):173–194, 2006.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 195.

[7] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1990.

[8] M. Harman, F. Islam, T. Xie, and S. Wappler. Automated test data generation for aspect-oriented programs. In *Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development*, pages 185–196, Charlottesville, Virginia, USA, 2009. ACM.

[9] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *Proceedings of the Third International Conference on Aspect-oriented Software Development*, pages 26–35, Lancaster, UK, 2004. ACM.

[10] J. H. Holland. *Adaptation in Natural and Artificial Systems.* MIT Press, Cambridge, MA, USA, 1992.

[11] A. C. Jensen and Betty H.C. Cheng. On the Use of Genetic Programming for Automated Refactoring and the Introduction of Design Patterns. In *Proceedings of the 2010 Genetic and Evolutionary Computation Conference*, Portland, OR, USA, 2010. ACM.

[12] K. Keeton, C. Santos, D. Beyer, J. Chase, and J. Wilkes. Designing for disasters. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 59–62, Berkeley, CA, USA, 2004. USENIX Association.

[13] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[14] G. Kiczales and E. Hilsdale. Aspect-oriented programming. In *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, page 313, Vienna, Austria, 2001. ACM.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[16] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems).* The MIT Press, December 1992.

[17] U. Kulesza, V. Alves, A. Garcia, C. J. de Lucena, and P. Borba. Improving Extensibility of Object-Oriented Frameworks with Aspect-Oriented Programming. In *Reuse of Off-the-Shelf Components*, volume 4039 of *Lecture Notes in Computer Science*, pages 231–245. Springer Berlin / Heidelberg, 2006.

[18] S. Luke. ECJ: A Java evolutionary computation library. http://cs.gmu.edu/ eclab/projects/ecj/, 2006.

[19] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and Betty H.C. Cheng. Composing adaptive software. *Computer*, 37(7):56–64, 2004.

[20] K. Meffert. JGAP: Java Genetic Algorithms and Genetic Programming Package. http://jgap.sf.net.

[21] A. J. Ramirez, D. B. Knoester, Betty H.C. Cheng, and P. K. McKinley. Applying Genetic Algorithms to Decision Making in Autonomic Computing Systems. In *Proceedings of the Sixth International Conference on Autonomic Computing*, pages 97–106 (Best Paper Award), Barcelona, Spain, June 2009.

[22] A. L. Santos, A. Lopes, and K. Koskimies. Framework specialization aspects. In *Proceedings of the Sixth International Conference on Aspect-oriented Software Development*, pages 14–24, Vancouver, British Columbia, Canada, March 2007. ACM.

[23] N. Shärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behaviour. *ECOOP 2003–Object-Oriented Programming*, pages 327–339, 2003.

[24] W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest. Automatically Finding Patches Using Genetic Programming. In *Proceedings of the 2009 International Conference on Software Engineering*, pages 364–374, Vancouver, Canada, May 2009. IEEE Computer Society.