Integrating HeuristicLab with Compilers and Interpreters for Non-Functional Code Optimization

Daniel Dorfmeister
Software Competence Center Hagenberg
Hagenberg, Austria
daniel.dorfmeister@scch.at

Oliver Krauss
Johannes Kepler University Linz
Linz, Austria
University of Applied Sciences Upper Austria
Hagenberg, Austria
oliver.krauss@fh-hagenberg.at

ABSTRACT

Modern compilers and interpreters provide code optimizations during compile and run time, simplifying the development process for the developer and resulting in optimized software. These optimizations are often based on formal proof, or alternatively stochastic optimizations have recovery paths as backup. The Genetic Compiler Optimization Environment (GCE) uses a novel approach, which utilizes genetic improvement to optimize the run-time performance of code with stochastic machine learning techniques.

In this paper, we propose an architecture to integrate GCE, which directly integrates with low-level interpreters and compilers, with HeuristicLab, a high-level optimization framework that features a wide range of heuristic and evolutionary algorithms, and a graphical user interface to control and monitor the machine learning process. The defined architecture supports parallel and distributed execution to compensate long run times of the machine learning process caused by abstract syntax tree (AST) transformations. The architecture does not depend on specific operating systems, programming languages, compilers or interpreters.

CCS CONCEPTS

• **Human-centered computing** → *Visualization toolkits*; • **Computing methodologies** → *Machine learning algorithms*; • **General and reference** → **Experimentation**; **Performance**; *Evaluation*;

KEYWORDS

Optimization, Compiler, Interpreter, Distributed Computing, Architecture, Metaheuristics, HeuristicLab, Truffle, Graal

ACM Reference Format:
Daniel Dorfmeister and Oliver Krauss. 2020. Integrating HeuristicLab with Compilers and Interpreters for Non-Functional Code Optimization. In *Genetic and Evolutionary Computation Conference Companion (GECCO '20 Companion), July 8–12, 2020, Cancún, Mexico.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3377929.3398103

ACM ISBN 978-1-4503-7127-8/20/07…$15.00
https://doi.org/10.1145/3377929.3398103

1 INTRODUCTION

Genetic Compiler Optimization Environment (GCE) [5, 6] integrates genetic improvement (GI) [9, 10] with execution environments, i.e., the Truffle interpreter [23] and Graal compiler [13]. Truffle is a language prototyping and interpreter framework that interprets abstract syntax trees (ASTs). Truffle already provides implementations of multiple programming languages such as Python, Ruby, JavaScript and C. Truffle languages are compiled, optimized and executed via the Graal compiler directly in the JVM. Thus, GCE provides an option to generate, modify and evaluate code directly at the interpreter and compiler level and enables research in that area.

HeuristicLab [20–22] is an optimization framework that provides a multitude of meta-heuristic algorithms, research problems from literature and a graphical user interface allowing for statistical analysis by the user. HeuristicLab has support for genetic programming (GP) used primarily for symbolic regression [4].

In this paper, we present an approach to integrate low-level execution environments with a high-level optimization framework, i.e., to enable integration of GCE and HeuristicLab. This allows GCE to take advantage of the algorithms and graphical user interface (GUI) provided by HeuristicLab, and thus an easy way to configure, observe and evaluate the optimization process for the user. In addition, HeuristicLab can integrate with execution environments using GCE and use more complex grammars used by real-world programming languages and their compilers. The architecture is designed to be independent of specific programming languages and their execution environments, and is operating system agnostic.

The primary purpose of this approach is to enable research into code optimizations. Many of the architectural considerations deal with the ability to distribute the executions on multiple machines, and enable advantages available due to the direct compiler integration, such as knowledge about the code syntax, and the available variables on stack and heap. While the approach can generally be used for genetic programming as well as genetic improvement, the primary target is optimizing existing code. Due to executing compiled code as opposed to an interpreted DSL, non-functional features – specifically run-time performance and memory usage – can be measured and optimized while taking real-world conditions, introduced by the hardware architecture and optimizations by the compiler, into account. This is related primarily to the research field of genetic improvement [9], but can also be applied to fitness functions for generating new functionality with genetic programming.

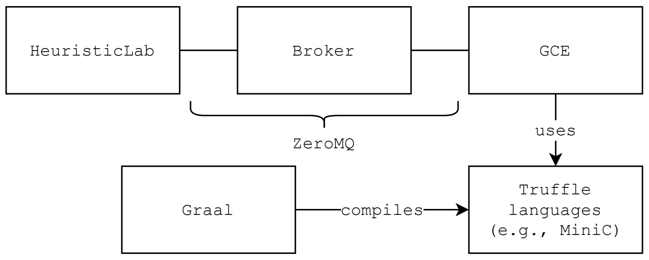


Figure 1: Overview of the technologies used in the approach described in this publication

The major advantages of this approach are achieved via the combination of HeuristicLab and GCE. HeuristicLab provides a multitude of algorithms and options to configure, monitor and evaluate optimization experiments not offered by GCE. GCE provides the ability to generate and manipulate source code executed and optimized directly in a compiler enabling research in that area, and providing programming languages written in Truffle that have more capabilities than achievable with the domain-specific languages provided by HeuristicLab.

HeuristicLab includes several grammars for domain-specific languages (DSLs), e.g., for symbolic regression and Robocode[1], which even uses the Java compiler for evaluation of solution candidates. There is no support for general purpose languages, e.g., C or JavaScript. As frameworks that support a wide range of real-world programming languages already exist, e.g., Truffle [23], re-implementing them for HeuristicLab is not reasonable. We show how to use external language implementations in HeuristicLab and, furthermore, how to use them for transformations of abstract syntax trees (ASTs).

Transformation and evaluation of ASTs can be run-time intensive, especially when accurately measuring run-time performance, which makes parallel execution of the genetic operators, i.e., creation, crossover and mutation, important. We present a way to not only execute AST transformations in parallel but also to distribute them across multiple machines.

In summary, the contributions of this publication are:

- Integration of GCE, a low-level execution environment, with HeuristicLab, a high-level optimization framework.
- An approach that is independent of operating systems, programming languages and execution environments.
- Parallelism is supported for both conducting experiments at the same time, and using several machines in one experiment.
- Real-world programming languages and compilers can be used rather than DSLs.
- AST transformations that are directly conducted and evaluated in the execution environment instead of a domain-specific language implemented in the optimization framework.

2 BACKGROUND

The following section gives an overview of the HeuristicLab framework, the ZeroMQ messaging library, the Graal compiler, the Truffle interpreter, the Genetic Compiler Optimization Environment (GCE) and the MiniC language. Figure 1 shows how these technologies interact with each other. In the context of this work, these technologies are used as follows:

- HeuristicLab provides a wide range of heuristic and evolutionary algorithms, which can be configured easily via the provided graphical user interface (GUI).
- ZeroMQ is a light-weight messaging library used for communication between HeuristicLab and GCE.
- Graal is a compiler for optimized languages. It is a highly optimizing just-in-time (JIT) compiler that is available in OpenJDK since Java 9 and is used to compile Truffle abstract syntax trees (ASTs) from byte code to machine code.
- Truffle is the interpreter framework that the optimized languages are written in. It can interpret AST nodes on the JVM without the use of Graal, albeit with a lower performance.
- The Genetic Compiler Optimization Environment (GCE) integrates with Truffle to optimize non-functional features of a given (sub-)AST, such as run-time performance or code size. It also supports integration with external optimization frameworks.
- MiniC is a simple Truffle language and subset of ANSI C11 used to showcase the capabilities of GCE.

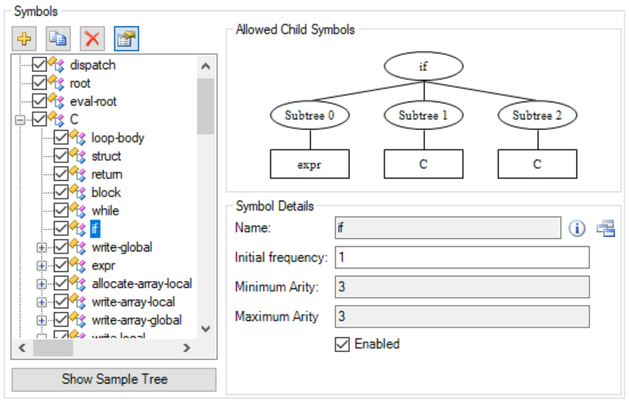
2.1 HeuristicLab

HeuristicLab [20–22] is an extensible optimization framework for heuristic and evolutionary algorithms featuring a GUI. The GUI is partially shown in Figure 2, where a grammar for genetic programming and the `if` symbol with its allowed child symbols and configuration options can be seen.

HeuristicLab is implemented in C# an thus requires the .NET Framework or Mono. However, HeuristicLab is able to use another application for evaluation of solution candidates [12]. By using the *HL3 External Evaluation Java library*[2,3] in a Java application, the evaluation of a solution candidate can be implemented easily. This library and the `ExternalEvaluationProblem` in HeuristicLab handle the inter-process communication. Though, operators beside the evaluator, e.g., the crossover and the mutation operator, cannot make use of the Java application.

The most flexible way to extend HeuristicLab are plugins, which allow to add algorithms, problems, and operators. In this way, communication between HeuristicLab and the Java application is possible in all operators and even before a single operator is executed, e.g., to conduct configuration tasks. Thus, the Java application can send a list of supported options, which can be configured by the user before starting the algorithm. Extension via a plugin is also the basis for our approach.

[1]https://robocode.sourceforge.io/
[2]https://dev.heuristiclab.com/trac.fcgi/wiki/Documentation/Howto/OptimizeExternalApplications
[3]https://dev.heuristiclab.com/trac.fcgi/export/HEAD/misc/documentation/Tutorials/OptimizingExternalApplicationswithHeuristicLab.pdf



Name: if
Initial frequency: 1
Minimum Arity: 3
Maximum Arity: 3
[✓] Enabled

Figure 2: A part of HeuristicLab's GUI, representing the MiniC grammar including its symbols (left) and details of the if symbol (right), such as allowed child symbols and a usage frequency

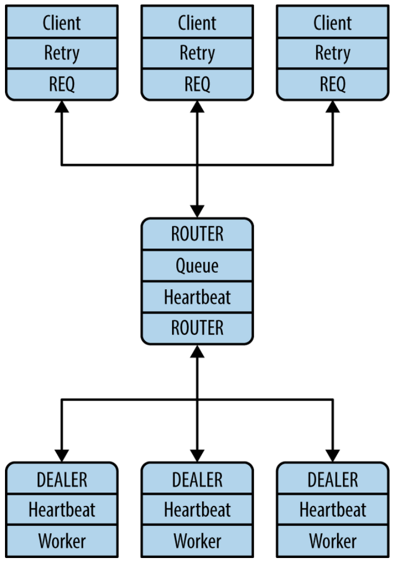


Figure 3: The Paranoid Pirate pattern [2] is used for communication between the client (HeuristicLab) and the worker (part of GCE), both connecting to the broker (center)

HeuristicLab Hive [17] enables distributed execution of arbitrary jobs, e.g., algorithm executions or single solution evaluations. By installing the Hive Slave software on a computer it can be utilized to execute Hive jobs. If external evaluation should be used, additional setup of the application executing the evaluation is required on every computer. In addition, the user is responsible to assign certain jobs only to computers supporting the required functionality as the Hive Server, which distributes the jobs to computers, does not have a deeper understanding of the nature of the jobs. As neither of these restrictions are feasible, HeuristicLab Hive will not be considered henceforth.

2.2 ZeroMQ

As the operations in meta-heuristic algorithms that deal with manipulating ASTs and evaluating them can be seen as events, a message queuing approach was selected as basis for communication. ZeroMQ [2] is a light-weight messaging library, which supports a wide range of programming languages and operating systems. This is the reason it was chosen as core technology for this approach.

The API of ZeroMQ uses a range of socket types, e.g., the REQ, REP, DEALER, and ROUTER sockets. A basic pattern is the request-reply pattern, which uses the REP and the REQ sockets. The client uses the REQ socket to send requests to a REP socket and receive replies in an alternating way. If a REQ socket is connected to multiple REP sockets, round-robin scheduling is used. The DEALER/ROUTER sockets are asynchronous versions of the REQ/REP sockets, respectively, they use fair queuing for incoming messages.

The ZeroMQ socket types can be combined to form the Paranoid Pirate pattern [2], which can be seen in Figure 3. The Paranoid Pirate pattern can be used for robust reliable queuing. A naive broker is introduced to manage multiple workers and perform load balancing. New workers can register at any time. Workers send heartbeats to the broker on a regular basis, so the broker can remove them from the queue when they stop. The clients reconnect to the broker and resend a request if no response was received within a timeout period, which allows the broker to restart after a failure – no messages are lost.

Due to the advantages the Paranoid Pirate pattern provides in consideration to distributed parallel architectures and the ability to deal with nodes that become unavailable for any reason, it is utilized as a basis for communication in the presented architecture. In our approach the pattern has been adapted to add logic to the broker dealing with distribution of workloads according to the workers' capabilities.

2.3 Graal

Graal [13] is an aggressively optimizing just-in-time (JIT) compiler. It is written in Java as part of the OpenJDK project. It compiles Truffle ASTs in byte code to machine code, and features several optimizations. Some of these optimizations are speculative and can be undone if necessary. This is called deoptimization. Graal uses an intermediate representation (IR), which is a directed graph containing control and data flow. The IR can be viewed using the Ideal Graph Visualizer (IGV). Graal and IGV are open source and provide a basis for research in compiler optimizations. [18, 19]

2.4 Truffle

Truffle [23] is an open-source self-optimizing interpreter framework for prototyping programming languages. It uses abstract syntax trees (ASTs), wherein language concepts (conditionals, loops, variables, ...) are represented as nodes. Truffle does not feature a lexer, parser or linker, and only contains the options for implementing AST nodes. These nodes can be executed in any JVM. Each node in a Truffle language consists of a generic implementation and specializations for every data type this node supports, enabling dynamic typing and a self-optimization mechanism that rewrites

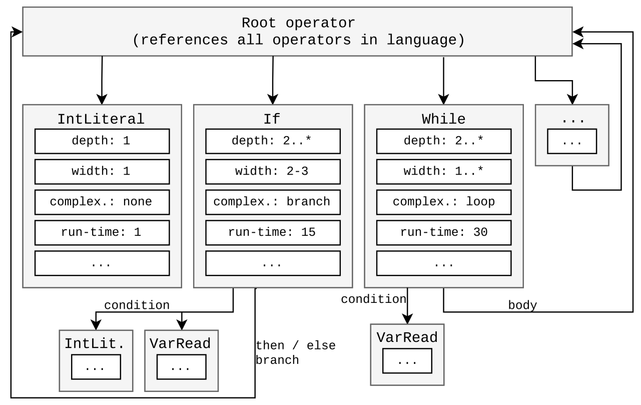


Figure 4: Syntax graph. Nodes (grey) represent language concepts, and contain information about them (white) utilized in the genetic operators. The edges to child nodes represent the syntax. [7]

these nodes using specialization and generalization. Truffle integrates with Graal for high-performance compilation and execution. As they are executed in the JVM, Truffle languages provide several features for developers such as garbage collection. Truffle currently provides several open and closed source guest language implementations, including Python, Ruby, JavaScript and C. [1, 24]

2.5 Genetic Compiler Optimization Environment

The Genetic Compiler Optimization Environment (GCE) [5, 6] is an optimizer framework that directly integrates with the Truffle framework. GCE itself is language agnostic as well as algorithm agnostic. GCE is designed to use genetic improvement (GI) [10] to optimize individual AST nodes in a Truffle language. GCE analyzes any given Truffle language, collecting the following information:

- Node classes existing in the language
- Relationships between these node classes
- Grouping node classes into terminals and non-terminals
  - Non-terminals are assigned a minimal depth and width required to create a minimally valid AST.
- Specific behaviour is analyzed as well
  - Allocation, read and write access to stack or heap
  - Branching behavior
  - Looping behavior
  - Calls to other functions

The above information is used to automatically create a syntax graph that can be utilized in all major genetic operators, i.e., creation, crossover and mutation. This syntax graph (see Figure 4) creates or selects ASTs according to a specific request (size, expected run-time performance, ...) and can only create ASTs that can compile in the target language. Not all of these ASTs are useful as they can still contain endless loops, dead code (i.e., code that is never executed), and other issues. The major advantage is the removal of uncompilable ASTs, which otherwise can amount to 80 % of generated solutions [14–16]. The syntax graph also provides an abstraction layer of any specific Truffle language. [7]

GCE also provides a knowledge base, storing information on conducted experiments and executed ASTs, such as the recorded fitness (size, run-time performance, ...) and the outcome of test cases. This information is furthermore used in a pattern mining approach to identify recurring AST optimizations, and automatically adapt the syntax graph with further restrictions against execution errors. This part of the framework is not utilized in this work. [8]

While GCE does provide an internal optimization suite with several genetic algorithms and genetic operators, it also provides concepts to integrate external optimization frameworks. These concepts allow the utilization of the syntax graph in external frameworks and a method for evaluating a given AST and fitness function directly with Truffle and Graal. GCE was implemented primarily to enable research into the improvement of non-functional requirements of software, such as run-time performance or memory use. For this purpose it provides capabilities to accurately measure these requirements, and utilize them in a fitness function in external systems. This enables the utilization of the major advantages of GCE, and provides the basis for this work.

The work presented in this publication aims to couple GCE with HeuristicLab as GCE does not provide a user interface and the convenience of configuration, monitoring the optimization process and evaluating the results that are provided by HeuristicLab. Additionally, the internal optimization suite contains only genetic algorithms, whereas HeuristicLab provides a multitude of other search-based algorithms and configuration options.

Accurate run-time performance measurement is quite costly as the Graal compiler requires a warm-up of about 100.000 executions and another 100.000 executions are needed to minimize side effects in the run-time measurement [6]. For this reason, the architecture presented in this publication is designed to enable distributing experiments over multiple GCE instances at the same time.

2.6 MiniC

MiniC is a test language specifically developed to test and integrate GCE with Truffle. It is a subset of the ANSI C11 standard of C [3]. At the time of writing it is primarily missing structs, pointers and unsigned data types.

The language currently consists of 169 node classes of which 30 are terminals. These classes can be used as non-terminal and terminal symbols in a GI experiment with GCE, and via the work we present here, HeuristicLab. A Coco/R-generated parser [11] is used to transform MiniC code into Truffle AST nodes.

In the context of this work MiniC is used to showcase the code and ASTs that can be optimized with the architecture.

3 ARCHITECTURE

The architecture of the approach we present consists of three components: HeuristicLab, Broker and Genetic Compiler Optimization Environment (GCE).

Section 3.1 gives an overview of the individual components, section 3.2 explains the broker and messaging infrastructure. Section 3.3 describes the HeuristicLab plugin and section 3.4 shows the integration of the Genetic Compiler Optimization Environment (GCE) with interpreter and compiler in detail.

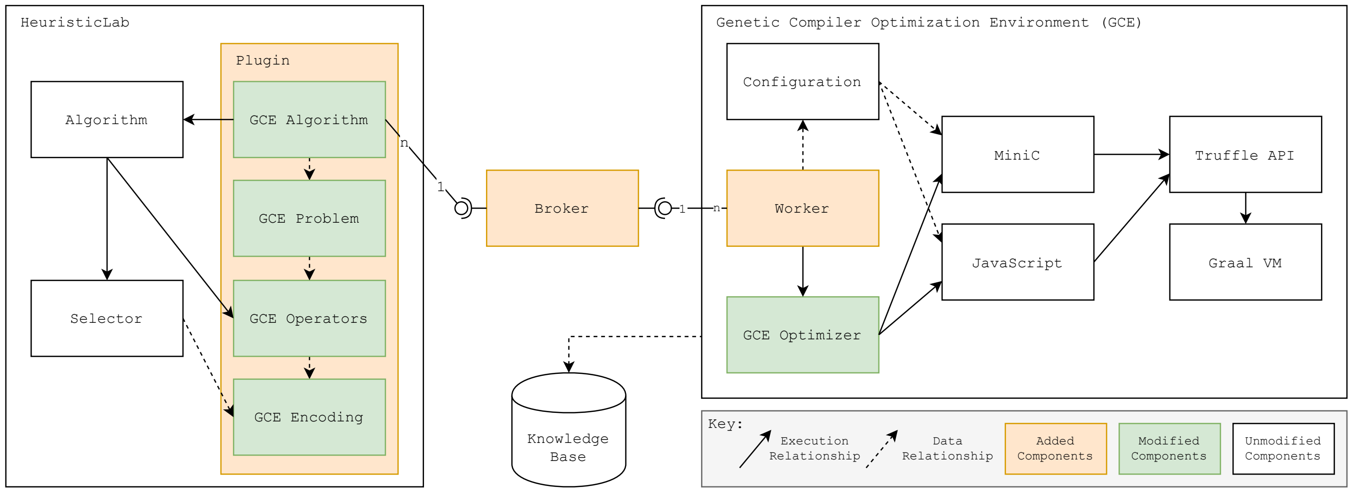


Figure 5: The architecture of the approach to combine HeuristicLab and GCE comprising a HeuristicLab plugin, a broker implemented using ZeroMQ and the worker as part of GCE

3.1 Overview

Figure 5 shows the defined architecture. The broker handles the communication between the client, HeuristicLab, and the workers.

The code required by HeuristicLab to communicate with the broker is contained within a plugin that extends and uses several classes of HeuristicLab. HeuristicLab can execute multiple algorithms in parallel, they can all be connected to the broker at the same time.

The workers, which are part of GCE, also connect to the broker. They can send configuration options to the clients, and use the GCE Optimizer to execute AST transformations. Both the configuration and the GCE Optimizer access Truffle languages, e.g., MiniC and JavaScript, which rely on the Truffle API and the Graal VM.

3.2 Broker and Messaging Infrastructure

ZeroMQ [2] provides a message queue but does not require a dedicated broker. Our approach introduces a light-weight broker that considers different worker capabilities for handling distributed processing. This could also have been done in HeuristicLab directly but was chosen to be implemented in a standalone broker to enable future integration with other optimization frameworks, in addition to allowing load balancing when multiple HeuristicLab instances run different algorithms at the same time.

Figure 6 shows the sequence flow of messages for a genetic algorithm started in HeuristicLab. First, the algorithm sends messages to initialize a worker and create and evaluate an initial population. Second, the main loop of the genetic algorithm starts, which selects parents that should be crossed, sends a `CrossoverRequest` (see Listing 1) to the broker, which forwards the message to an appropriate worker based on the algorithm run identifier. The worker looks up the parent solutions in the knowledge base, performs the crossover and sends a `CrossoverResponse` message to the broker, which forwards the message to the HeuristicLab instance that sent the corresponding request. The mutator and evaluator, which are also part of the main loop, send messages according to the crossover. Finally, before the algorithm ends, it sends a `StopAlgorithm` message to deinitialize the worker.

The Paranoid Pirate pattern was adapted slightly to reflect the nature of the workers described in subsection 3.4. When a worker registers at the broker, it also sends the programming languages it supports and whether it supports the provision of the configuration

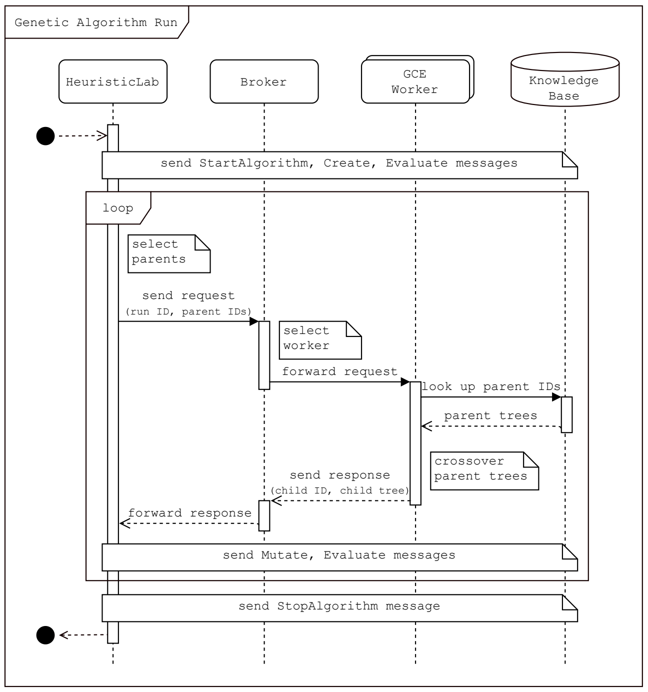


Figure 6: Sequence flow of messages between the components during the run of a genetic algorithm in HeuristicLab focusing on the crossover operator

options described in subsection 3.3. This allows the broker to route requests to a subset of the registered workers that support the operation requested by the client. Also, the client sends a unique identifier with every message related to an algorithm run, as the worker must be initialized for a specific algorithm run and must know, which subsequent messages are associated with a specific algorithm run as the source code is compiled specifically per run.

ZeroMQ messages consist of one or more frames. Data that is relevant to the broker for routing is sent in individual frames. The last frame contains a payload only relevant to the worker, which is ignored by the broker. Thus, the broker can handle any payload and only must be adapted when the protocol changes.

Protocol Buffers[4] are used for the payload of the messages, as they are light-weight and work with most relevant programming languages. The Any message type is used to wrap all Protocol Buffers messages, which adds a unique identifier that can be used by the worker to identify the type of request to handle. Listing 1 shows a request by the crossover operator, which contains unique identifiers for both parent solutions to crossover, and the dedicated response, which contains a unique identifier of the child solution created by the worker and the child solution represented by the root node of an AST. Each node has a unique identifier, a symbol represented by a unique identifier and a list of child nodes, which may be empty.

Listing 1: Protocol Buffers message definitions for the crossover operator

```
syntax = "proto3";

message CrossoverRequest {
  int64 parentSolutionId0 = 1; // first parent
  int64 parentSolutionId1 = 2; // second parent
}

message CrossoverResponse {
  int64 childSolutionId = 1; // child solution
  TreeNode childSolution = 2; // child's root node
}

message TreeNode {
  int64 id = 1; // the ID of the node
  int64 symbolId = 2; // the ID of the node's symbol
  repeated TreeNode children = 3; // child nodes
}
```

3.3 HeuristicLab Plugin

HeuristicLab has out-of-the-box support for tree-based genetic programming [4], which can be extended by creating a HeuristicLab plugin.

As shown in Figure 5, the plugin we created for this publication consists of four logical groups of classes, i.e., (1) an algorithm that only handles basic communication and calls another algorithm to perform machine learning, (2) a problem definition that retrieves configuration options and lets the user configure operators and further settings, (3) several operators that use GCE to perform operations on solution candidates, and (4) classes to represent the encoding of the solution candidates, i.e., symbols, trees, and grammars.

[4]https://developers.google.com/protocol-buffers/

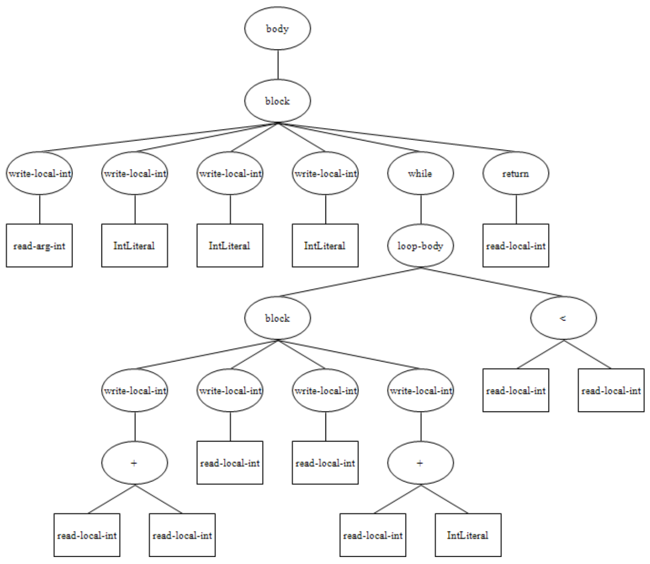


Figure 7: Abstract syntax tree of the Fibonacci algorithm as rendered by HeuristicLab (corresponding source code see Listing 2)

When the *algorithm* is started, it sends a configuration and basic data about the problem that should be optimized, e.g., which programming language is required and the source code, so a group of workers capable of handling the problem can be identified by the broker and selected for subsequent requests related to the same problem. The data is also used by the workers for initialization, for details, see subsection 3.4.

The problem definition is used by the contained algorithm to configure itself and the utilized operators. The contained algorithm can be one of the algorithms that comes with HeuristicLab, e.g., a genetic algorithm or an evolution strategy, an algorithm defined in the user interface of HeuristicLab, or an algorithm that is added by a plugin.

When the algorithm finishes, is stopped by the user, or aborted due to an unrecoverable error, the broker is informed. The broker in turn messages all workers involved in the algorithm run, so they can clear their cache of the source code and experiment context.

Listing 2: MiniC sample code that is optimized by HeuristicLab and executed by GCE (corresponding AST see Figure 7)

```
int fibonacci(int n) {
  int i, now, prev, next;
  prev = 0;
  now = 1;
  i = 0;
  while (i < n) {
    next = now + prev;
    prev = now;
    now = next;
    i = i + 1;
  }
  return prev;
}
```

The *problem definition*, when created, retrieves configuration options from a worker. These configuration options consist of the following categories:

- A set of supported programming languages:
  - Languages are represented by their grammar, the symbols contained are grouped for clarity.
  - Each symbol has a unique identifier, a name and description, a minimum and maximum arity and a list of possible child symbols. Symbols with a maximum arity of zero are terminal symbols.
  - The user can configure the arity, the initial frequency on the population, and whether a symbol should be used at all, as shown in Figure 2.
- A set of operators per operator type:
  - The user can select which operator implementation to use for, e.g., solution creation or mutation, and configure the parameters of the operator.
  - The operators can have primitive parameters, e.g., simple text fields for strings or integers.
  - Complex parameters are also supported, where the user can select one or more options from a predefined list of values, which can be configurable themselves.
- Additional parameters: This works like the configuration of operators, without the necessity of parameters being related to an operator. These parameters can also be grouped.

When the algorithm is started, the configuration of the language, the operators and the additional parameters are collected and sent to the worker. The problem also defines a set of test cases, a small program written in the selected language, together with input and corresponding output values. Listing 2 shows an example of a test program written in the MiniC language.

The *operators* are derived from the respective symbolic expression tree operator, e.g., the crossover is derived from the class `SymbolicExpressionTreeCrossover`, as shown in Listing 3. The only significant additions are a lookup parameter to determine the identifier of the current algorithm run, which is needed by the workers, and an override for the method that performs the operation: First, the Protocol Buffers message for the request is constructed, which may contain data about which ASTs to transform if applicable. Second, the request is sent together with the current algorithm run identifier, and the operator waits for a response. Last, the response message is interpreted and transformed to its HeuristicLab counterpart – Figure 7 shows an AST generated by the worker and rendered by HeuristicLab. Hence, adding operators that may be needed by other algorithms, e.g., particle swarm optimization, is viable.

The classes needed to *encode solution candidates*, i.e., trees, are derived from existing classes in HeuristicLab. The only addition is the support for identifiers in symbols and trees. Symbols are distinguished by their name, which makes adding several symbols with the same name but different allowed child symbols to a grammar impossible. As this happens frequently in real world programming languages (e.g., multiple overrides for mathematical operators depending on the related data types), we had to create a new grammar storing symbols by their identifiers.

Listing 3: Crossover that communicates with a GCE worker using the Protocol Buffers messages from Listing 1, as implemented in the HeuristicLab plugin (simplified)

```
// attributes, constructors etc. omitted
public class GCECrossover :
    SymbolicExpressionTreeCrossover {
  // looks up the value of the param. in the problem
  private readonly ILookupParameter<StringValue>
    runIdParam;

  public override ISymbolicExpressionTree Crossover
    (IRandom random,
     ISymbolicExpressionTree parent0,
     ISymbolicExpressionTree parent1) {
    // instantiate Protocol Buffers message
    CrossoverRequest request =
      new CrossoverRequest {
        ParentSolutionId0 = ((GCETree) parent0).Id,
        ParentSolutionId1 = ((GCETree) parent1).Id
      };
    // Send extension method implements
    // Paranoid Pirate pattern client
    CrossoverResponse response =
      request.Send<CrossoverResponse>(
        runIdParam.ActualValue.Value);
    // ToSymbolicExpressionTree extension method
    // transforms Protocol Buffers message to a tree
    GCETree child = response.ChildSolution
      .ToSymbolicExpressionTree(
        response.ChildSolutionId,
        (GCEGrammar) grammar);
    return child;
  }
}
```

HeuristicLab allows to execute multiple algorithms at the same time, hence serving as multiple clients from the broker's perspective. HeuristicLab also has a parallel engine, which can be used instead of the default sequential engine. The parallel engine can execute multiple operators at the same time, which typically accelerates an algorithm run by a significant factor depending on the degree of parallelism and available workers.

3.4 Worker

A specialized configuration worker module in GCE is responsible for publishing available options to the broker. This concerns the grammar of the available Truffle languages, including all information required by the HeuristicLab problem definition, e.g., maximum arity and allowed child symbols. It also publishes implemented operators, and their possible configuration options, which are useful to HeuristicLab. For example, this includes different creation strategies using the syntax graph provided by GCE.

A separate worker module in GCE connects to the broker. The worker module publishes its available Truffle languages with an identification matching the published languages from the configuration module. The worker module passively awaits requests from the HeuristicLab plugin. Whenever an experiment is started on the worker, it parses the code submitted in the request, and initializes the language and program context in Truffle.

Whenever creation, crossover or mutation requests are submitted in an algorithm run, the worker module adapts the AST in the program context according to the specifications utilizing the GCE Optimizer. Due to this, the generated ASTs are always able to compile, and are valid within the overall program context. This includes access to global and local variables available outside of the considered (sub-)AST. The worker stores generated AST nodes in the knowledge base (see Figure 5). This allows later analysis of node classes for modifying the occurring frequency in solutions in HeuristicLab. Storing ASTs in the knowledge base also enables using distributed workers without transmitting entire ASTs in the messages between HeuristicLab and GCE.

The worker also conducts the evaluation, according to all requirements of the fitness function selected in HeuristicLab. The following features are currently provided:

- Results of all defined test cases
- The run-time performance benchmarked after optimization with Graal
- The estimated run-time performance as alternative to benchmarking with Graal
- The code size in Truffle (the amount of AST nodes)

4 CONCLUSION AND OUTLOOK

In this paper, we present a novel architecture to integrate low-level execution environments, provided by GCE, with a high-level optimization framework, i.e., HeuristicLab. This is achieved by extending HeuristicLab with a plugin that can communicate with a broker that is based on ZeroMQ, and by adding a worker to GCE, which can handle the requests of the HeuristicLab plugin. The goal is to enable research into code optimization or generation while taking the real world behavior of a compiler into account.

This approach has the advantage that the algorithms and GUI provided by HeuristicLab can be used while also being able to use actual execution environments and real-world programming languages without having to re-implement them in HeuristicLab. This also enables research into compiler and interpreter optimizations via HeuristicLab. Because of the distributed architecture and the resulting communication between HeuristicLab, broker and workers, there is some overhead in the execution. There also is need for specific implementation as the genetic operators provided by HeuristicLab cannot be used and must be implemented in GCE to utilize its syntax graph and fully utilize its functionality. The overhead in execution time is mitigated by supporting parallel AST transformations and distributing them across multiple machines.

The approach uses technologies that are supported on a wide range of operating systems and programming languages, e.g., ZeroMQ and Protocol Buffers, thus being easy to extend to compilers besides Graal, which should be considered in the future. Also, GCE currently only contains genetic operators, thus only supporting genetic algorithms. Adding support for operators required by further algorithms implemented in HeuristicLab, e.g., particle swarm optimization, would allow to take full advantage of the integration with HeuristicLab.

ACKNOWLEDGEMENTS

The work described in this paper was done within the research project Emergency Detection in Elevator Networks (EDEN), funded by the Austrian Research Promotion Agency (FFG) under the General Programme, project number 866124.

REFERENCES

[1] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. 2015. High-performance Cross-language Interoperability in a Multi-language Runtime. *SIGPLAN Not.* 51, 2 (Oct. 2015), 78–90. https://doi.org/10.1145/2936313.2816714

[2] Pieter Hintjens. 2013. *ZeroMQ* (1 ed.). O'Reilly Media, Inc.

[3] ISO. 2011. *ISO/IEC 9899:2011 Information technology — Programming languages — C.* International Organization for Standardization, Geneva, Switzerland. 683 (est.) pages. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853

[4] Michael Kommenda, Gabriel Kronberger, Stefan Wagner, Stephan Winkler, and Michael Affenzeller. 2012. On the Architecture and Implementation of Tree-Based Genetic Programming in HeuristicLab. In *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO '12).* Association for Computing Machinery, New York, NY, USA, 101–108. https://doi.org/10.1145/2330784.2330801

[5] Oliver Krauss. 2017. Genetic Improvement in Code Interpreters and Compilers. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion 2017).* Association for Computing Machinery, New York, NY, USA, 7–9. https://doi.org/10.1145/3135932.3135934

[6] Oliver Krauss. 2018. Towards a Framework for Stochastic Performance Optimizations in Compilers and Interpreters: An Architecture Overview. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes (ManLang '18).* Association for Computing Machinery, New York, NY, USA, Article Article 9, 7 pages. https://doi.org/10.1145/3237009.3237024

[7] Oliver Krauss, Hanspeter Mössenböck, and Michael Affenzeller. [n.d.]. Towards Knowledge Guided Genetic Improvement. In *2020 IEEE/ACM International Workshop on Genetic Improvement (GI)* (2020-10).

[8] O. Krauss, H. Mössenböck, and M. Affenzeller. 2019. Mining Patterns from Genetic Improvement Experiments. In *2019 IEEE/ACM International Workshop on Genetic Improvement (GI).* 28–29.

[9] William B. Langdon. 2015. Genetic Improvement of Software for Multiple Objectives. In *Search-Based Software Engineering*, Márcio Barros and Yvan Labiche (Eds.). Springer International Publishing, Cham, 12–28.

[10] W. B. Langdon and M. Harman. 2015. Optimizing Existing Software With Genetic Programming. *IEEE Transactions on Evolutionary Computation* 19, 1 (Feb 2015), 118–135. https://doi.org/10.1109/TEVC.2013.2281544

[11] Hanspeter Mössenböck, Markus Löberbauer, Albrecht Wöß, and University of Linz. 2018. *The Compiler Generator Coco/R.* http://www.ssw.uni-linz.ac.at/Coco/ Last Accessed - 2020-04-17.

[12] Miguel Mujica Mota, Idalia Flores, and Daniel Guimarans. 2015. *Applied Simulation and Optimization: In Logistics, Industrial and Aeronautical Practice.* https://doi.org/10.1007/978-3-319-15033-8

[13] OpenJDK. 2020. *Graal Project.* http://openjdk.java.net/projects/graal/ Last Accessed - 2020-04-17.

[14] Michael Orlov and Moshe Sipper. 2009. Genetic Programming in the Wild: Evolving Unrestricted Bytecode. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation (GECCO '09).* ACM, New York, NY, USA, 1043–1050. https://doi.org/10.1145/1569901.1570042

[15] M. Orlov and M. Sipper. 2011. Flight of the FINCH Through the Java Wilderness. *IEEE Transactions on Evolutionary Computation* 15, 2 (April 2011), 166–182. https://doi.org/10.1109/TEVC.2010.2052622

[16] Justyna Petke, Mark Harman, William B. Langdon, and Westley Weimer. 2014. *Using Genetic Improvement and Code Transplants to Specialise a C++ Program to a Problem Class.* Springer Berlin Heidelberg, Berlin, Heidelberg, 137–149. https://doi.org/10.1007/978-3-662-44303-3_12

[17] Andreas Scheibenpflug, Stefan Wagner, Gabriel K. Kronberger, and Michael Affenzeller. 2012. HeuristicLab Hive - An Open Source Environment for Parallel and Distributed Execution of Heuristic Optimization Algorithms. In *1st Australian Conference on the Applications of Systems Engineering ACASE'12*, Robin Braun and Zenon Chaczko (Eds.). Sydney, Australia, 63–65. http://research.fh-ooe.at/en/publication/3064

[18] Doug Simon, Christian Wimmer, Bernhard Urban, Gilles Duboscq, Lukas Stadler, and Thomas Würthinger. 2015. Snippets: Taking the High Road to a Low Level. *ACM Trans. Archit. Code Optim.* 12, 2, Article 20 (June 2015), 20:20:1–20:20:25 pages. https://doi.org/10.1145/2764907

[19] Lukas Stadler, Gilles Duboscq, Hanspeter Mössenböck, and Thomas Würthinger. 2012. Compilation Queuing and Graph Caching for Dynamic Compilers. In *Proceedings of the Sixth ACM Workshop on Virtual Machines and Intermediate Languages (VMIL '12).* ACM, New York, NY, USA, 49–58. https://doi.org/10.1145/2414740.2414750

[20] Stefan Wagner. 2009. *Heuristic Optimization Software Systems - Modeling of Heuristic Optimization Algorithms in the HeuristicLab Software Environment.* Doctor Technicae. Johannes Kepler University, Linz, Austria.

[21] S. Wagner, G. Kronberger, A. Beham, M. Kommenda, A. Scheibenpflug, E. Pitzer, S. Vonolfen, M. Kofler, S. Winkler, V. Dorfer, and M. Affenzeller. 2014. *Architecture*

and Design of the HeuristicLab Optimization Environment. Springer International Publishing, Heidelberg, 197–261. https://doi.org/10.1007/978-3-319-01436-4_10

[22] Stefan Wagner, Stephan Winkler, Erik Pitzer, Gabriel Kronberger, Andreas Beham, Roland Braune, and Michael Affenzeller. 2007. Benefits of Plugin-Based Heuristic Optimization Software Systems. In *Computer Aided Systems Theory – EUROCAST 2007*, Roberto Moreno Díaz, Franz Pichler, and Alexis Quesada Arencibia (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 747–754.

[23] Christian Wimmer and Thomas Würthinger. 2012. Truffle: A Self-optimizing Runtime System. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH '12).* ACM, New York, NY, USA, 13–14. https://doi.org/10.1145/2384716.2384723

[24] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013).* ACM, New York, NY, USA, 187–204. https://doi.org/10.1145/2509578.2509581