PISA — A Platform and Programming Language Independent Interface for Search Algorithms

Stefan Bleuler, Marco Laumanns, Lothar Thiele, Eckart Zitzler

Computer Engineering and Networks Laboratory (TIK)
Department of Information Technology and Electrical Engineering
Swiss Federal Institute of Technology (ETH) Zurich, Switzerland

Email: {bleuler, laumanns, thiele, zitzler}@tik.ee.ethz.ch

TIK-Report No. 154
Institut für Technische Informatik und Kommunikationsnetze,
ETH Zürich
Gloriastrasse 35, ETH-Zentrum, CH–8092 Zürich, Switzerland

October 18, 2002

Abstract

This paper introduces a text based interface (PISA) that allows to separate the algorithm-specific part of an optimizer from the application-specific part. These parts are implemented as independent programs forming freely combinable modules. It is therefore possible to provide these modules as ready-to-use packages. As a result, an application engineer can easily exchange the optimization method and try different variants, while an algorithm designer has the opportunity to test a search algorithm on various problems without additional programming effort.

1 Introduction

Complex optimization problems can be found in many application areas. One aspect that contributes to the complexity of these problems comprises the characteristics of the search space; exact algorithms are often not applicable. Multiple objectives form another type of difficulty that classical optimization methods were not designed for. Accordingly, alternative techniques have been developed

in the course of time, e.g., evolutionary algorithms, tabu search, and simulated annealing. For each of these approaches, a large number of single- and multi-objective variants exists, which differ more or less from each other.

The variety of optimization strategies, though, poses new problems. It becomes increasingly difficult

- for an application engineer to choose, implement, and apply state-of-the-art algorithms without in-depth programming knowledge and expertise in the optimization domain, and
- for a developer of optimization methods to test and compare algorithms on different test problems.

In both cases, the main problems are the implementation overhead and potential implementation errors. Today's optimization methods usually involve complex operations and require a considerable programming effort; the same holds for the application and test problem side.

Programming libraries have been designed to facilitate the implementation of optimization algorithms. They are usually geared to a particular technique, e.g., evolutionary algorithms, and provide reusable and extendible program components that can be combined in different ways. If a specific optimization method is to be tailored to a specific application and knowledge in both algorithm domain and application domain is available, then these libraries provide valuable tools to reduce the programming effort. However, still considerable implementation work is necessary, if we intend to test various algorithms on a certain application or apply a specific algorithm to different test problems. Furthermore, programming libraries require a certain training period, and their use is restricted to specific programming languages and often also to specific computing platforms.

This paper proposes a different concept. The idea is to divide the implementation of an optimization method into an application-specific part and an algorithm-specific part as shown in Fig. 1. The latter contains the selection procedure, while the former encapsulates the representation of solutions, the generation of new solutions, and the calculation of objective function values. Since the two parts are realized by distinct programs that communicate via a text-based interface, this approach provides maximum independence of programming languages and computing platforms. It even allows to use precompiled, ready-to-use executable files, which, in turn, minimizes the implementation overhead and avoids the problem of implementation errors. As a result, an application engineer can easily exchange the optimization method and try different variants, while an algorithm designer has the opportunity to test a search algorithm on various problems without additional programming effort (cf. Fig. 1). Certainly, this concept is not meant to replace programming libraries. It represents a complementary approach that allows to build collections of optimizers and applications, all of them freely combinable across computing platforms.

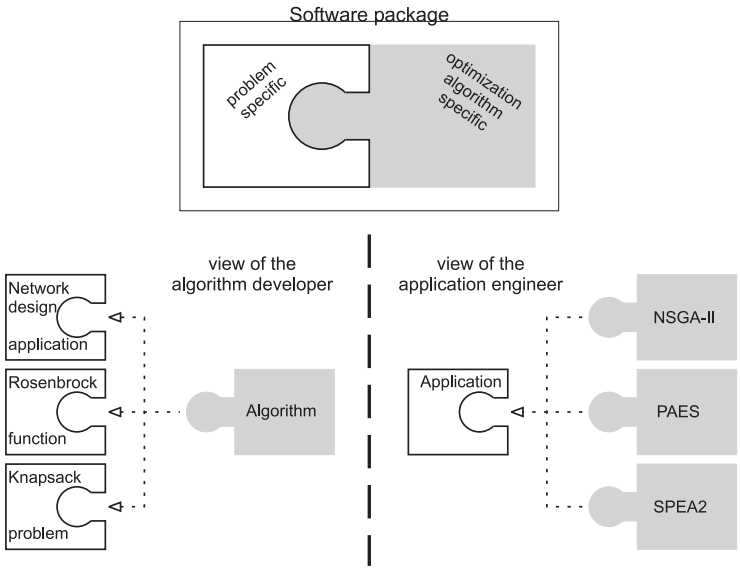

Figure 1: Illustration of the concept underlying PISA. The applications on the left hand side and the multiobjective optimizers on the right hand side are examples only and can be replaced arbitrarily.

2 Related Work

In the following we will focus on the field of evolutionary multiobjective optimization although much work has been done with respect to other heuristics as well.

2.1 Existing Interfaces

Several attempts have been undertaken to ease the programming process and implementation effort of evolutionary algorithms. Most software engineering solutions resulted in (object oriented) class libraries, mainly written in C++, e.g. GAlib, EO [8] or TEA [5], or Java, e.g. JEO [1], but except for the TEA library none has a support for multiobjective optimization. A second group consists of implementations in MATLAB tool-boxes, for which also multiobjective versions exist [9]. These programming frameworks are characterized by a modular description of the algorithms, where different operators can be combined. Whenever a non-standard problem representation is used with search operators that are not part of the library core, additional source code must be added by the user.

Programming libraries are excellent tools for research in the algorithm domain: Experiments with different parameters and operators can easily be set up and new algorithms can be designed. However, from the application point of view it would be desirable to have program packages that can be used by engineers without any specific knowledge about the implementation of the opti-

mization algorithms. Such approaches exist outside the evolutionary algorithms domain, e.g., the CPLEX [3] package for solving linear programming problems. For this purpose it is helpful to explicitly define the underlying model.

Formal models for evolutionary algorithms do exist, as will be described in the following, but no approach is widely in use so far. Instead, implementation and modeling have been dealt with independently: Neither does an existing interface rely on one of the proposed models nor was a model designed with an implementation in mind. From a methodological viewpoint, however, an implementation should always follow a model, ideally a formal one.

2.2 Unified Modeling of Multiobjective Optimizers

Building generic models is common practice in the field of evolutionary computation to abstract from algorithm-specific details and to provide a general formulation. The components of these models mainly consist of (stochastic) operators that represent the 'genetic' or 'evolutionary' operations, e.g., in the universal evolutionary algorithm given in [2].

A multiobjective extension of this model was proposed in [11], which allows different sizes for parent and offspring population. However, this model does not consider an archive as an active part of the algorithm. In [4], eight operational steps were proposed that an effective multiobjective evolutionary algorithm should incorporate as a generic structure. The authors remarked that the succession of these steps could be varied in different ways covering a broad range of algorithms, but that some memetic algorithms using local search or techniques based on explicit manipulation of building blocks do not fit into this framework.

The difficulties in finding a common structure for the succession of the different operations has led to a more coarsely grained formulation in the Unified Model of Multiobjective Evolutionary Algorithms (UMMEA [7]). This model was, however, not primarily designed from a software-technological viewpoint for easy implementation, but rather to facilitate comparative case studies on the effects of different operators and parameters.

This multitude of different generic models shows that it is an unsolved task to find a common structure which exhaustively describes all possible multiobjective optimization algorithms. The main question here is what level of abstraction or detail is appropriate for our aim of providing a framework for easy and representation-independent implementation.

3 Design Foundations

3.1 Design Goals

Our aim is to design and implement a standardized, extendible and easy to use framework for multiobjective optimization algorithms. For the development of such a framework, we follow several design goals:

Separation of concerns. The algorithm-specific component and the problem-specific component should have a maximum independence from each other. It should be possible to implement only the part of interest, while the other part is treated as a ready-to-use black box.

Small overhead. The additional effort necessary to implement interfaces and communication mechanisms has to be as small as possible. The extra running time due to the data exchange between the components of the system should be minimized.

Simplicity and flexibility. The approach must have a simple and comprehensible way of handling input and output data and setting parameters, but should hide all algorithm-specific work and implementation details from the user. The specification of the flow control and the data exchange format should state minimal requirements for all implementations, but still leave room for future extensions and optional elements.

Portability and platform independence. The framework itself and hence the possibility to embed any existing algorithm into it should not depend on machine types, operating systems or programming languages. Optimizers and applications, written in different programming languages must interconnect seamlessly. It is obvious that running a module on a different operating system might require re-compilation, but porting an existing program to another operating system or machine type must not be complicated by the interface implementation. Furthermore, when porting is difficult, it must be possible to run the two processes on different machines with possibly different operating systems, letting them communicate over a network link.

Reliability and safety. A reliable and correct execution of the different components is very important for the broad acceptance of the system. For instance unusual parameter settings must not cause a system failure.

Given these design goals, the development of a programming framework becomes a multiobjective problem itself, and it is impossible to reach a maximum satisfaction in all design aspects. Therefore, a compromise solution is sought. Here, we focus on simplicity and small overhead and are willing to accept less flexibility. The motivation behind this is that the system will only be employed by many people if it is easy to use and does not require excessive programming work. To compensate for the lack of flexibility, we will make the format extendible to a certain degree so that it will still be possible for interested users to adapt it to specific needs and features. How all these design goals are realized will be described in the next section.

3.2 Basic Model

The design of a framework for multiobjective evolutionary optimization should be based on some kind of generic model for such algorithms. The model used

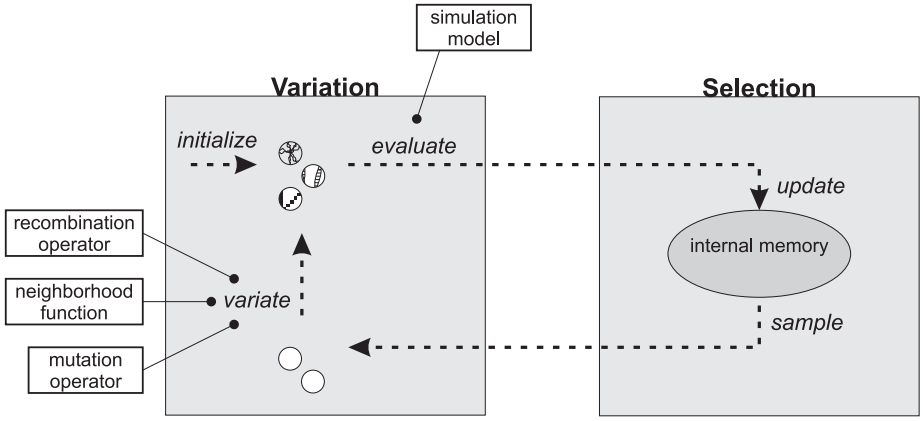

Figure 2: Model of a general search algorithm. Circles stand for individuals and the external boxes give examples of operators which could perform the respective basic operation.

for the design of PISA is shown in Fig. 2. It follows the distinction between problem-dependent and problem-independent parts. This goes hand in hand with the commonly accepted division into variation and selection because the selection operates only in objective space not in decision space, disregarding a few exceptions.

Usually, though, variation is considered to be part of the optimizer rather than part of the problem representation, and class libraries for optimization often implement variation operators for standard test problem representations like binary strings. In any real world application, however, variation is highly problem dependent. In order to build reusable components, all problem and representation specific operators must reside completely in one module. This complies with the design goal of separation of concern. Most evolutionary multiobjective optimization algorithms and many others (e.g. simulated annealing, tabu search) fit into the proposed model.

4 Architecture

Based on the model proposed in the previous section a more detailed formal model for our framework can be established. Our model will be based on Petri nets, because in contrast to state machines, Petri nets allow to describe the data flow and the control flow within a single computational model. The resulting architecture is depicted in Fig. 3, where the term *variator* is used to denote the problem-dependent part and *selector* the problem-independent part. A transition (rectangular boxes) can fire if all inputs are available. On firing a transition performs the stated operations and provides all outputs.

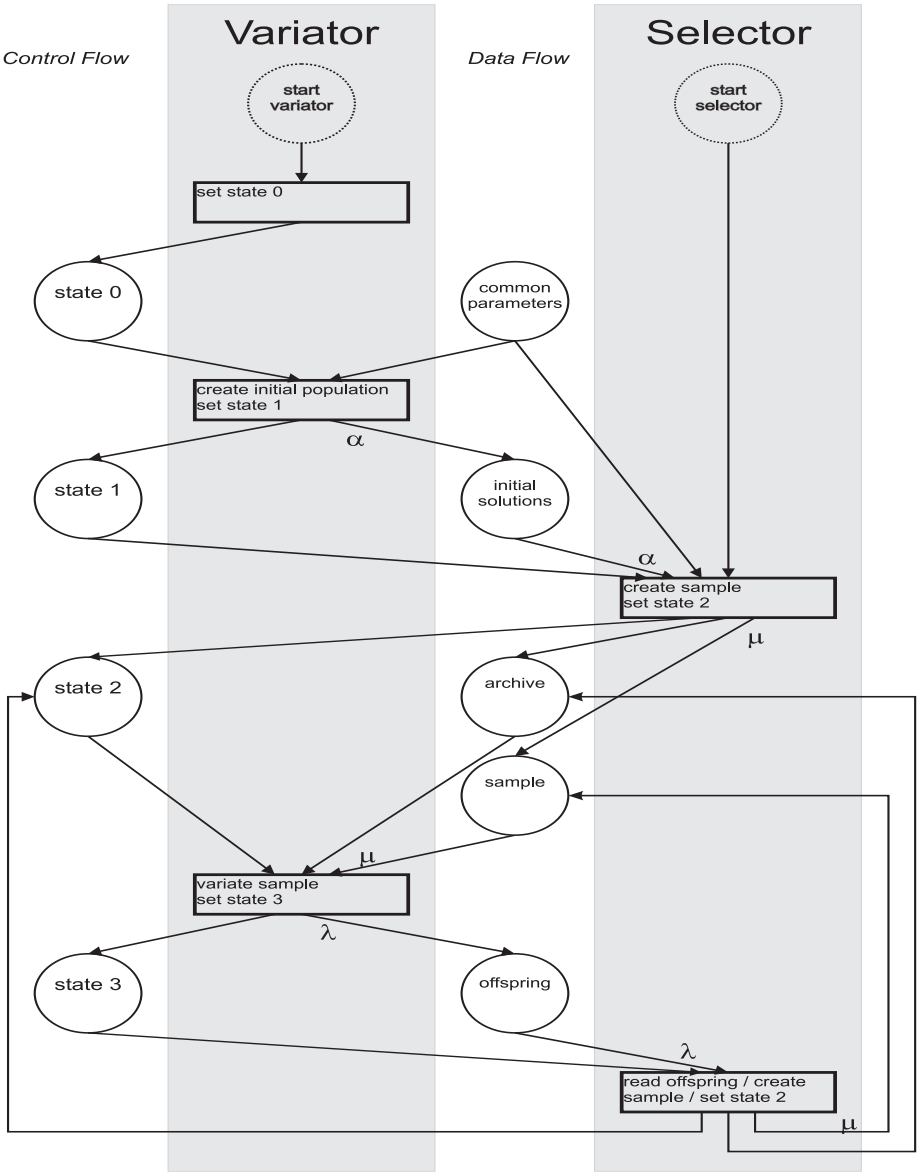

Figure 3: The control flow and data flow specification of PISA using Petri nets. The transitions (rectangular boxes) represent the operations by the processes implementing the variator and the selector part. The places in the middle represent the data flow and correspond to the data files which both processes read and write. The places at the left margin represent reading and writing of the state variable that is stored in a common state file and hence direct the control flow.

State	Action	Next State
State 4	Variator terminates.	State 5
State 6	Selector terminates.	State 7
State 8	Variator resets. (Getting ready to start in state 0)	State 9
State 10	Selector resets. (Getting ready to start in state 0)	State 11

Table 1: Stop and reset states.

4.1 Control Flow

The model ensures that there is a consistent state for the whole optimization process and that only one module is active at any time. Whenever a module reads a state that requires some action on its part, the operations are performed and the next state is set. The implementation of the flow control is discussed in Section 5.1.

The core of the optimization process consists of state 2 and state 3: In each generation the selector chooses a set of parent individuals and passes them to the variator. The variator generates new individuals on the basis of the parents, computes the objective function values of the new individuals, and passes them back to the selector.

In addition to the core states two more states are shown in Fig. 3: State 0 and state 1 trigger the initialization of the variator and the selector, respectively. In state 0 the variator reads the necessary parameters (common parameters shown in Fig. 3 and local parameters not shown). For more information on parameters refer to Section 5.3. Then, the variator creates an initial population, calculates the objective values of the individuals and passes the initial population to the selector. In state 1, the selector also reads the required parameters, then selects a sample of parent individuals and passes them to the variator.

The abovementioned states provide the basic functionality of the optimization. To improve flexibility in the use of the modules states for resetting and stopping are added (see Table 1). The actions taken in states 5, 7, 9 and 11 are not defined. This allows a module to react flexibly, e.g., if the selector reads state 5, which signals that the variator has just terminated, it could choose to set the state to 6 in order to terminate as well. Another selector module could instead set the state to 10, thus, causing itself to reset.

4.2 Data Flow

The data transfer between the two modules introduces some overhead compared to a traditional monolithic implementation. Thus, the amount of data exchange for each individual must be minimized. Since all representation specific operators are located in the variator, the selector does not have to know the representation of the individuals. Therefore, it is sufficient to convey only the following data to the selector for each individual: an index, which identifies the individual in both modules, and one objective vector. In return, the selector

only needs to communicate the indices of the parent individuals to the variator. The proposed scheme allows to restrict the amount of data exchange between the two modules to a minimum. In the following we will refer to passing the essential information as passing a population or a sample of individuals.

As to objective vectors the following semantics is used: An individual is superior to another in regard to one objective, if the corresponding element of the objective vector is smaller, i.e., objective values are to be *minimized*.

Furthermore, the two modules need to agree on the sizes of the three collections of individuals passed between each other: the initial population, the sample of parent individuals, and the offspring individuals. These sizes are denoted as α, μ and λ in Fig. 3. Instead of using some kind of automatic coordination, which would increase the overhead for implementing the interface we have decided to specify the sizes as parameter values. Setting μ and λ as parameters requires that they are constant during the optimization run. Most existing algorithms comply with this requirement. Nevertheless, dynamic population sizes could be implemented using the facility of transferring auxiliary data (cf. Section 5.2).

As described in Section 4.1, a collection of parent individuals is passed from the selector to the variator and a collection of offspring individuals is returned. The actual individuals are stored on the variation side. Since the selector might use some kind of archiving method, the variator would have to store all individuals ever created, because one of them might be selected as a parent again. This can lead to unnecessary memory exhaustion and can be prevented by the following mechanism: the selector provides the variator with a list of all individuals that could ever be selected again. This list is denoted as archive in Fig. 3. The variator can optionally read this list, delete the respective individuals and re-use their indices. Since most individuals in a usual optimization run are not archived, the benefit from this additional data exchange is much larger than its cost. Section 5.2 describes how the data exchange is implemented.

5 Implementation Aspects

After describing the architecture of the interface based on Petri nets in the previous section, this section discusses the most important issues of implementation.

5.1 Synchronization

In order to reach the necessary separation and compatibility, the selector and the variator are implemented as two separate processes. These two processes can be located on different machines with possibly different operating systems. This complicates the implementation of a synchronization method. Most common methods for interprocess communication are therefore not applicable.

Closely following the Petri nets model (cf. Fig. 3), a common state variable which both modules can read and write is used for synchronization. The two

processes regularly read this state variable and perform the corresponding actions. If no action is required in a certain state, the respective process sleeps for a specified amount of time and then rereads the state variable.

Coherent with our decision for simplicity and ease of implementation, the common state variable is implemented as an integer written to a text file. In contrast to the alternative of using sockets, file access is completely portable and familiar to all programmers. The only requirement is access to the same file system. On a remote machine this can for example be achieved through simple `ftp put` and `get` operations. As another benefit of using a text file for synchronization it is possible for the user to influence the two processes by changing the state variable in the state file with a text editor.

5.2 Data Exchange

Another important aspect of the implementation is the data transfer between the two processes. Following the same reasoning as for synchronization, all data exchange is established through text files. Using text files with human readable format allows the user to monitor data exchange easily, e.g., for debugging. For the same reason, a separate file is used for each collection of individuals shown in Fig. 3. The resulting set of files used for communication between the two modules and for parameters is shown in Fig. 4. Simple examples of possible contents are shown as well to illustrate to file format.

To achieve a reliable data exchange through text files, the receiving module should be able to detect corrupted files. For instance, a file could be corrupted because the receiving process tries to read the file before it is completely written. However, it is assumed that the file is at least partially written when the state variable is changed. The detection of corrupted files is enabled by adding two control elements to the data elements: The first element specifies the number of data elements following. After the data elements an 'END' tag ensures that the last element has been completely written. The receiving module can read the specified number of elements without looking for a 'END' and then check if the 'END' tag is at the expected place.

Between the two control elements blocks of data are written, describing one individual each. In this example, such block consists of an index and two objective values if written by the variator and only one index if written by the selector.

The file format described so far provides the exchange of the data necessary for all optimization methods. This might not be sufficient for all modules since some techniques, e.g. mating restrictions and constraint handling, require the exchange of additional data. Therefore, the specification allows for optional data blocks after the first 'END' tag. A module which expects additional data can read on after the first 'END', whereas a simple module is not disturbed by data following after the first 'END'. A block of optional data has to start with a name. Providing a name for blocks of optional data allows to have several blocks of optional data and therefore make one module compatible with many other modules which require some specific data each. The exact specifications

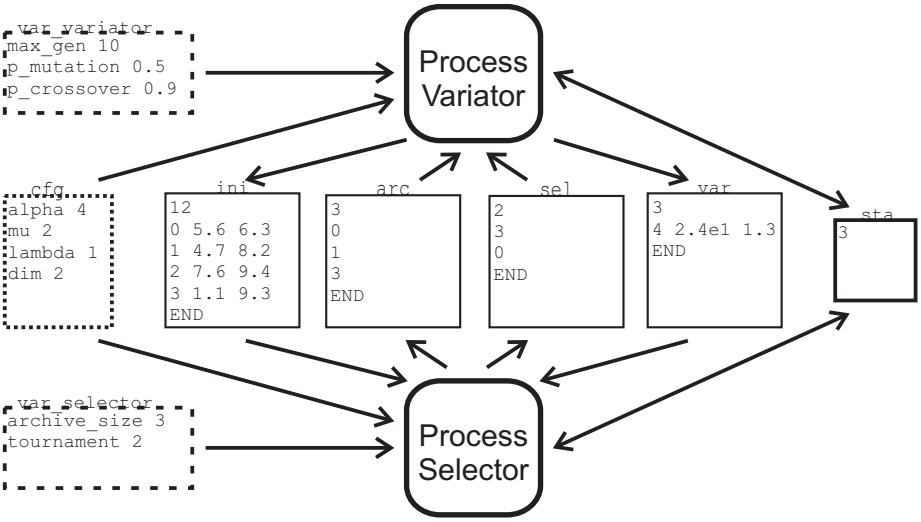

Figure 4: Communication between modules through text files. Four files for the data flow: The initial population in `ini`, the archive of the selector in `arc`, the sample of parent individuals in `sel` and the offspring in `var`. The `cfg` file contains the common parameters and `sta` contains the state variable. Additionally two examples for local parameter files are shown.

of the file formats are given in the appendix.

5.3 Parameters

Several parameters are necessary to specify the behavior of both modules. Following the principle of separation of concern, each module specifies its own parameter set (examples are shown in Fig. 4). As an exception, parameters that are common to both modules are given in a common parameter file. This prevents users from setting different values for the same parameter on the variation and the selection side. The set of common parameters consists of the number of objectives (*dim*) and the sizes of the three different collections of individuals that are passed between the two modules (see Fig. 3).

The author of a module must specify which α, μ and λ combinations and which *dim* values the module can handle. A module can be flexible in accepting different settings of these parameters or it can require specific values. To ensure reliable execution, each module must verify the correct setting of the common parameters.

Two parameters, however, are needed in the part of each module which implements control flow shown in Fig. 3: i) the filename base specifying the location of the data exchange files as well as the state file and ii) the polling interval specifying the time for which a module in idle state waits before reread-

ing the file. The values of these parameters need to be set before the variator and the selector can enter state 0 and state 1, respectively.

6 Experimental Results

The interface specification has been tested by implementing sample variators and selectors on various platforms.

In a first set of experiments, an interface has been written in the programming language C and extended with the simple multi-objective optimizer SEMO (selector) and the LOTZ problem (variator), see [6]. They have been tested on various platforms (Windows, LINUX, Solaris) where the two processes have been residing as well on different machines as on the same machine. Despite of the fact, that a shared file system has been used in the case of distributed processes, no communication error was ever detected.

In a second experiment, a large application written in Java was tested with the well known multiobjective optimizer SPEA2 [12] written in C++ using the library TEA [5]. The purpose of the optimization was the design space exploration of a network processor including architecture selection, binding of tasks and scheduling, see [10]. The interface worked reliably again, even if the application program and the optimizer ran on two different computing platforms, i.e., Windows and Solaris.

In a final set of experiments, the intention was to estimate the expected run-time overhead caused by the interface. Based on the cooperation between the two processes variator and selector, one can derive that the overhead caused by the interface for each generation can be estimated as

$$P + T_{comm} + (N + \lambda(1 + D) + \mu)K_{comm}$$

where P denotes the polling interval chosen as well in the variator as in the selector, N, λ and μ denote the size of the archive, sample and offspring data sets, respectively, and D denotes the number of objectives. The rationale behind this estimation is that the time overhead consists of three parts, namely the average time to wait for a process to recognize a relevant state change, the overhead caused by opening and closing all relevant files including the state file, and a part that is proportional to the number of tokens in the data files. Note that besides the polling for a state change, the two processes do not compete for the processor, as the variation and selection are executed sequentially, see Fig. 3. It is not considered that in the variator as well as in the selector we need to store and process the population. On the other hand, the corresponding time overhead can be expected to be much smaller than the time to communicate via a file-based interface.

The parameters of this estimation formula have been determined for a specific platform and a specific interface implementation and good agreement over a large range of polling times and archive sizes has been found. In order to be on the pessimistic side, we have chosen to use the interface written in Java.

The underlying platform for both processes was a Pentium Laptop (600 MHz) running LINUX and we obtained the parameters

$$T_{comm} = 10ms \qquad K_{comm} = 0.05ms/\text{token}$$

For example, if we take an optimization problem with two objectives $D = 2$, a polling interval of $P = 100ms$, a population size of $N = 500$, and a sample and offspring size of $\lambda = \mu = 250$, then we obtain 185 ms time overhead for each generation. For any practically relevant optimization application, this time is much smaller than the computation time within the population-based optimizer and the application program. Note that for each generation, at least the 250 new individuals must be evaluated in the variator.

Clearly, these values are very much dependent on many factors such as the platform, the programming language and other processes running on the system. Nevertheless, we can summarize that the overhead caused by the interface is negligible for any practically relevant application.

7 Summary

In this paper, we have proposed a platform and programming language independent interface for search algorithms (PISA) that uses a well-defined text file format for data exchange. By separating the selection procedure of an optimizer from the representation specific part, PISA allows to maintain collections of precompiled optimization algorithms and applications which can be arbitrarily combined. That means on the one hand that application engineers with little knowledge in the optimization domain can easily try different optimization strategies for the problem at hand; on the other hand, algorithm developers have the opportunity to test optimization techniques on various applications without the need to program the problem-specific parts. This concept even works on distributed files systems across different operating systems and can also be used to implement application servers using the file transfer protocol over the internet.

This flexibility certainly does not come for free. The data exchange via files increases the execution time, and the implementation of the interface requires some additional work. As to the first aspect, we have shown in Section 6 that the communication overhead can be neglected for practically relevant applications; this also holds for comparative studies, independent of the benchmark problems used, where we are mainly interested in relative run-times. Also concerning the implementation aspect, the overhead is small compared to the benefits of PISA. The interface is simple to realize, and most existing optimizers and applications can be adapted to the interface specification with only few modifications. Furthermore, the file format leaves room for extensions so that particular details such as diversity measures in decision space can be implemented on the basis of PISA.

Crucial, though, for the success of the proposed approach is the availability of optimization algorithms and applications compliant with the interface. To this

end, the authors maintain a Web site at `http://www.tik.ee.ethz.ch/pisa/` which contains example implementations for download.

Appendix

The formats of all files used in the interface are specified in the following. Note that the stated limits (e.g. largest integer) give minimal requirements for all modules. It is possible to state larger limits in the documentation of each module.

Common Parameter File (cfg)

All elements (parameter names and values) are separated by white space.

```
cfg := 'alpha' WS PosInt WS 'mu' WS PosInt WS 'lambda' WS PosInt
       WS 'dim' WS PosInt
```

State File (sta)

An integer i with $0 \leq i \leq 11$.

```
Statefile := Int
```

Selector Files (sel and arc)

The first element specifies the number of data elements following before the first 'END'. The data contains only white space separated indices. Optional data blocks start with a name followed by the number of data elements before the next 'END'.

```
SelectorFiles := PosInt WS SelData 'END' SelOptional*
SelOptional := Name WS PosInt WS SelData 'END'
SelData := (Int WS)*
```

Variator Files (ini and var)

The first element specifies the number of data elements m following before the first 'END'. The data consists of one index and dim objective values (floats) per individual. If n denotes the number of individuals: $m = (dim+1) \cdot n$. Optional data blocks start with a name followed by the number of data elements before the next 'END'.

```
VariatorFiles := PosInt WS VarData 'END' VarOptional*
VarOptional := Name WS PosInt WS VarData 'END'
VarData := (Int WS (Float WS)*)*
```

Names for optional data

Names for optional data consist of maximally 127 characters, digits and underscores.

```
Name := Char (Digit | Char)*
Char : 'a-z' | 'A-Z' | '_'
```

White space

```
WS := (Space | Newline | Tab)+
```

Integers

The largest integer allowed is equal to the largest positive value of a signed integer in a 32 bit system: $maxint = 32767$

```
Int := '0' | PosInt
PosInt: '1-9' Digits*
```

Floats

Floats are non-negative floating point numbers with optional exponents. The total number of digits before and after the decimal point can maximally be 10. The largest possible float is: $maxfloat = 1e37$. For the exponent value exp applies: $-37 \leq exp \leq 37$

```
Float := (Digit+ '.' Digit*) | ('.' Digit+ Exp?) | (Digit+ Exp)
Exp := ('E'|'e') ('+'? | '-'?) Digit+
```

Digit

```
Digit := '0-9'
```

Acknowledgment

This work has been supported by the Swiss National Science Foundation (SNF) under the ArOMA project 2100-057156.99/1 and the SEP program at ETH Zürich under the poly project TH-8/02-2.

References

[1] M. G. Arenas, B. Dolin, J. J. Merelo, P. A. Castillo, I. F. D. Viana, and M. Schoenauer. JEO: Java evolving objects. In W. B. Langdon et al., editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, page 991, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

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

[3] N. Bixby and E. Boyed. *Using the CPLEX callable library.* CPLEX Optimization Inc., Houston, 1996.

[4] C. A. Coello Coello, D. A. Van Veldhuizen, and G. B. Lamont. *Evolutionary Algorithms for Solving Multi-Objective Problems.* Kluwer, New York, 2002.

[5] M. Emmerich and R. Hosenberg. TEA - a C++ library for the design of evolutionary algorithms. Technical Report CI-106/01, SFB 531, Universität Dortmund, 2000.

[6] M. Laumanns, L. Thiele, E. Zitzler, E. Welzl, and K. Deb. Running time analysis of multi-objective evolutionary algorithms on a simple discrete optimization problem. In *Parallel Problem Solving From Nature — PPSN VII*, 2002.

[7] M. Laumanns, E. Zitzler, and L. Thiele. A unified model for multi-objective evolutionary algorithms with elitism. In *Congress on Evolutionary Computation (CEC 2000)*, volume 1, pages 46–53, Piscataway, NJ, 2000. IEEE Press.

[8] E. Lutton, P. Collet, and J. Louchet. Easea comparisons on test functions: Galib versus eo. In P. Collet, C. Fonlupt, J.-K. Hao, E. Lutton, and M. Schoenauer, editors, *Proceedings of the Fifth Conference on Artificial Evolution (EA-2001)*, volume 2310 of *LNCS*, pages 219–230, Le Creusot, France, 2001. Springer Verlag.

[9] K. Tan, T. H. Lee, D. Khoo, and E. Khor. A Multiobjective Evolutionary Algorithm Toolbox for Computer-Aided Multiobjective Optimization. *IEEE Transactions on Systems, Man, and Cybernetics—Part B: Cybernetics*, 31(4):537–556, August 2001.

[10] L. Thiele, S. Chakraborty, M. Gries, and S. Künzli. *Network Processor Design 2002: Design Principles and Practices*, chapter Design Space Exploration of Network Processor Architectures. Morgan Kaufmann, 2002.

[11] D. A. Van Veldhuizen. *Multiobjective Evolutionary Algorithms: Classifications, Analyses, and New Innovations*. PhD thesis, Graduate School of Engineering of the Air Force Institute of Technology, Air University, June 1999.

[12] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the strength pareto evolutionary algorithm for multiobjective optimization. In K. Giannakoglou, D. Tsahalis, J. Periaux, K. Papailiou, and T. Fogarty, editors, *Evolutionary Methods for Design, Optimisation, and Control*, pages 19–26, Barcelona, Spain, 2002. CIMNE.