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

Stefan Bleuler, Marco Laumanns, Lothar Thiele, and Eckart Zitzler

ETH Zürich, Computer Engineering and Networks Laboratory (TIK),
CH–8092 Zürich, Switzerland,
{bleuler,laumanns,thiele,zitzler}@tik.ee.ethz.ch,
http://www.tik.ee.ethz.ch/pisa/

Abstract. This paper introduces an interface specification (PISA) that allows to separate the problem-specific part of an optimizer from the problem-independent part. We propose a view of the general optimization scenario, where the problem representation together with the variation operators is seen as an integral part of the optimization problem and can hence be easily separated from the selection operators. Both parts are implemented as independent programs, that can be provided as ready-to-use packages and arbitrarily combined. This makes it possible to specify and implement representation-independent selection modules, which form the essence of modern multiobjective optimization algorithms. The variation operators, on the other hand, have to be defined in one module together with the optimization problem, facilitating a customized problem description. Besides the specification, the paper contains a correctness proof for the protocol and measured efficiency results.

1 Introduction

The interest in the field of multiobjective optimization mainly comes from two research directions:

- The application side, where engineers face difficult real-world problems and therefore would like to apply powerful optimization methods.
- The algorithmic side, where researchers aim to develop optimization algorithms that can be successfully applied to a wide range of problems.

Since modern optimization methods have become increasingly complex the work in both areas requires a considerable programming effort. Often code for optimization methods and test problems is available, but the usage of these implementations is restricted to one programming language. A thorough understanding of the code is needed in order to integrate it with own work. This raises the question whether it is possible to divide the implementation into an application part and an optimizer part reflecting the interests of the two groups mentioned. An ideal separation would provide ready-to-use modules on both sides and these modules would be freely combinable.

C.M. Fonseca et al. (Eds.): EMO 2003, LNCS 2632, pp. 494–508, 2003.

The application engineer would like to couple his implementation of the optimization problem to an existing optimizer. However, it is obviously not possible to provide a general optimizer which works well for all problems since many parts of an optimization method are highly problem specific. In many realistic application studies, the representation of candidate solutions is problem specific. Consequently, also the variation operators are very often problem dependent. Consider for example discrete optimization problems which involve network and graph representations combined with continuous variables and specific repair mechanisms. In addition, the neighborhood structure induced by the variation operator (either explicitly like in simulated annealing or tabu search or implicitly like in evolutionary algorithms) strongly influences the success of the optimization. As a consequence, the representation of candidate solutions and the definition of the variation operator comprise the major locations where problem specific and a priori knowledge can be inserted into the search process. Clearly, there are cases where standard representations such as binary strings and associated variation operators are adequate. For these situations, standard libraries are available to ease programming, e.g. [1,3]. In summary, it is the task of the application engineer to define appropriate representations and neighborhood structures.

In contrary, most optimizers in the multiobjective field work with a selection operator which is only based on objective values of the candidate solutions and are thus problem independent.[1] This allows to separate the selection mechanism from the problem-specific parts of the optimization method. As most of the research in the area of multiobjective optimization has focused on selection mechanisms these algorithms have become more and more complex. Freely combinable modules for selection algorithms on one side and applications with appropriate variation operators on the other side would thus be beneficial for application engineers as well as algorithm developers.

This paper proposes an approach to realize such a separation into an application-specific part and a problem-independent 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 selection method and try different variants, while an algorithm designer has the opportunity to test a selection method on various problems without additional programming effort (cf. Fig. 1). Certainly, this concept is not meant to replace programming libraries. It is a complementary approach that allows to build collections of selec-

[1] Nevertheless, PISA is extensible and allows for transmitting additional information needed to consider e.g. niching strategies in selection.

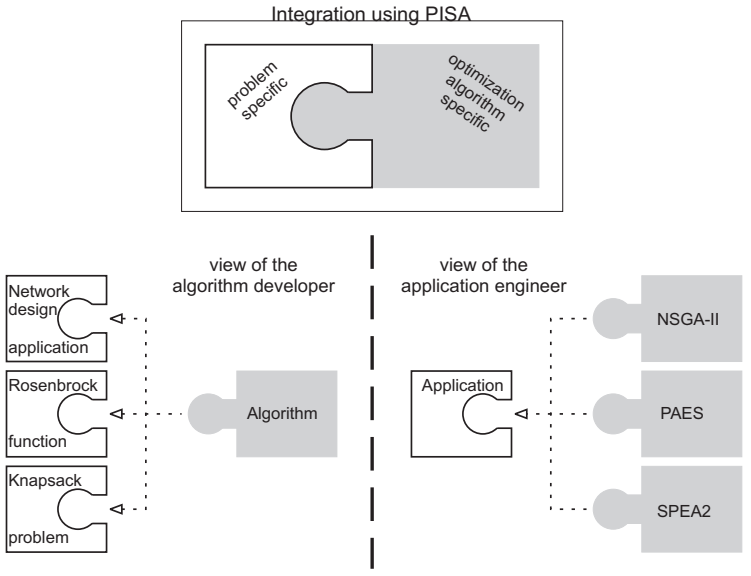

Fig. 1. Illustration of the concept underlying PISA. The applications on the left hand side and the multiobjective selection schemes on the right hand side are examples only and can be replaced arbitrarily

tion methods and applications including variation operators, all of them freely combinable across computing platforms.

2 Design Goals and Requirements

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

Separation of concerns. The algorithm-specific 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 should have a simple and comprehensible way of handling input and output data and setting parameters, but should hide all 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. The different components 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 should 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 Architecture

The proposed architecture is characterized by the following main features:

- Separation of selection on the one hand and variation and calculation of objective function values on the other.
- Communication via file system which allows for platform, programming language and operating system independence.
- Correct communication under weak assumptions about shared file system.
- Small communication overhead by avoiding to transmit decision space data.

We consider optimization methods that generate an initial set of candidate solutions and then proceed by an iterative cycle of evaluation, selection of promising candidates and variation. The selection is assumed to operate only in objective space. Nevertheless, the final specification of PISA is extensible and allows for transmitting additional information needed to consider e.g. niching strategies in selection.

Based on these assumptions, a 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. 2,

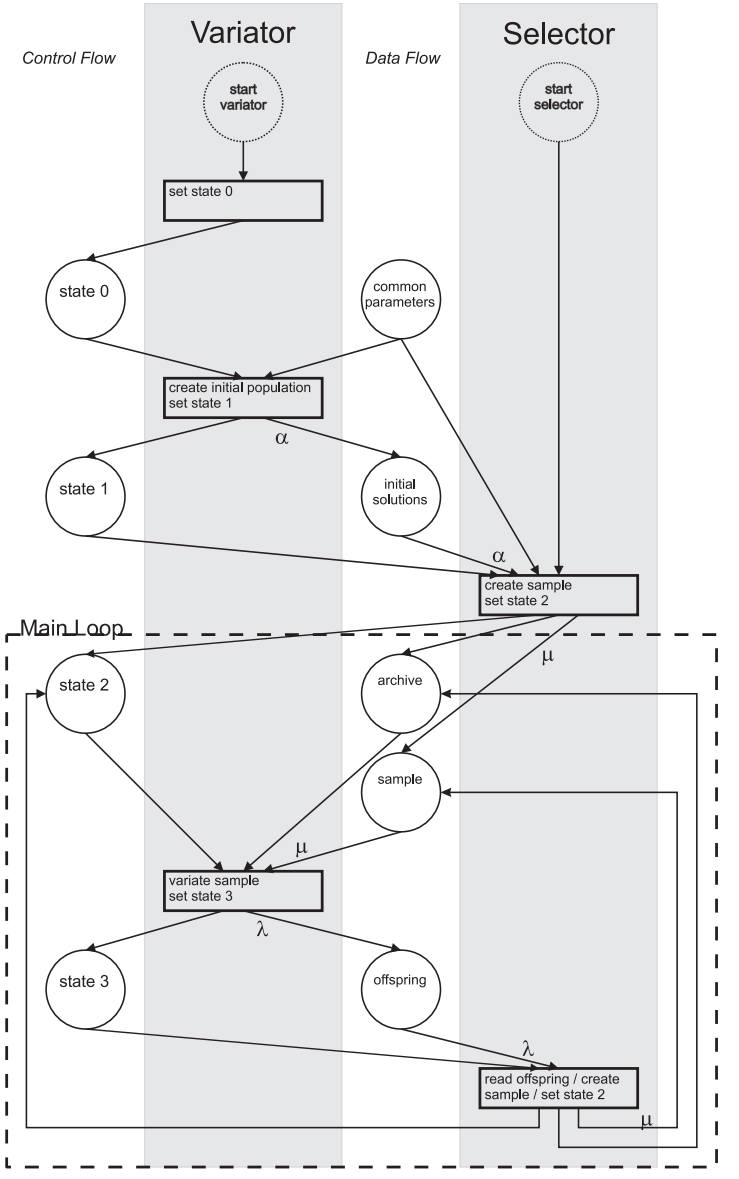

Fig. 2. The control flow and data flow specification of PISA using a Petri net. The transitions (rectangular boxes) represent the operations by the processes implementing the variator and the selector. The places in the middle represent the data flow and correspond to the data files which both processes 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

Table 1. Stop and reset states

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

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, consumes the input data and provides all outputs.

3.1 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 4.1.

The core of the optimization process consists of state 2 and state 3: In each iteration 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. 2. State 0 and state 1 trigger the initialization of the variator and the selector, respectively. In state 0 the variator reads the necessary parameters (the common parameters are shown in Fig. 2 and local parameters are not shown). For more information on parameters refer to Section 4.3. 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 above mentioned 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.

3.2 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 should 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 its 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 with 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. 2. 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 4.2).

As described in Section 3.1, a collection of parent individuals is passed from the selector to the variator and a collection of offspring individuals is returned. The actual representation of the individuals is stored on the variator 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. 2. 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 4.2 describes how the data exchange is implemented.

4 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.

4.1 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 net model (cf. Fig. 2), 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 number 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 manually edit the state file. The underlying assumptions about the file system and the correctness of this approach will be discussed in Section 4.4.

4.2 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. 2. The resulting set of files used for communication between the two modules and for parameters is shown in Fig. 3. 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. 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.

Additionally, the reading process is expected to replace a file's content with '0' after reading it and the writing process must check if no old data remains that would be overwritten. This represents the production and consumption of the data as shown in the Petri net in Fig. 2 and it prevents re-reading of old data which could happen if the writing of the new state can be seen earlier by the reading process then the writing of the data. For additional information on the requirements for the file system see Section 4.4.

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 objec-

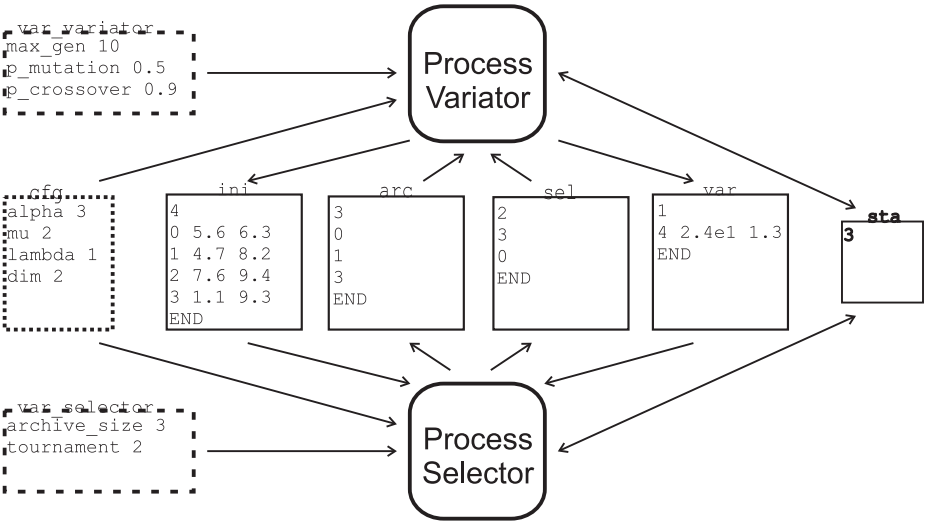

Fig. 3. Communication between modules through text files. Four files for the data flow: The initial population in `ini`, the archive of the selector in `arc`, the sample of parent individuals in `sel` and the offspring in `var`. The `cfg` file contains the common parameters and `sta` contains the state variable. Additionally two examples for local parameter files are shown

tive values for the files written by the variator and only one index for the files 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 every module 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 makes one module compatible with many other modules which require some specific data each. The exact specifications of the file formats are given in the appendix.

4.3 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. 3). 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. 2).

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. 2: 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 rereading 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, for example as command line arguments.

4.4 Correctness

It is not obvious that the proposed method for synchronization and data transfer works correctly. One problem arises for example from the fact that on a ordinary file system it cannot be assumed that the two successive write operations by one process to different files can be read in the same order by another process, i.e., changes in files can overtake each other. It is therefore necessary to state the necessary assumptions made about the file system and to show that the proposed system works correctly under these assumptions.

We assume that the file system is used by two processes $P1$ and $P2$ and has the following properties :

A) Writing of characters to a file is serial.
B) Writing of a character to a file is atomic.
C) A read by a process $P1$ from a file F that follows a write by $P1$ to F with no writes of F by another process $P2$ occurring between the write and the read by $P1$, always returns the data written by $P1$.
D) A read by a process $P1$ from a file F that follows a write by another process $P2$ to F after some sufficiently long time, returns the data written by $P1$ if there are no writes of F in between.

A possible underlying scenario is presented in Fig. 4. Two Processes $P1$ and $P2$ execute blocks of operations and communicate through a file system which can be modeled by two caches $C1$ and $C2$ and a memory M (see Figure 4). The time x needed to copy a file from a local memory to M is arbitrary but bounded. It may be different for each file access and for each distinct file.

There are two major properties that can be proven and which guarantee the correctness of the protocol with respect to the algorithm in Fig. 2.

The data transfer using the data files (archive, sample and offspring) is atomic. This property is a consequence of the properties A and B and the particular protocol used. The writer of a file puts a special `END` tag at the end of

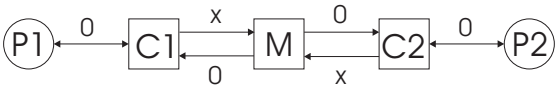

Fig. 4. Memory model for the file system. Two processes P1 and P2 communicate through the file system modeled by two local memories C1 and C2 and a global memory M. On some paths it takes zero time for the data to propagate from one element to the next. On others it takes a time x which is a positive bounded random variable

its data which can be recognized uniquely (see property B). As the writing of characters is serial, the reader of a file can determine when all data have been read. In addition, the reader starts reading if the file contains any data and it deletes all data after having read them. This way, there is a definite start and end time of the read process and all data are transmitted because of property A.

The sequence of operations according to Fig. 2 is guaranteed by the protocol. As shown in Fig. 2 two conditions are necessary for a process to resume operation: The presence of the respective state and the availability of the data. As a consequence of properties C and D the waiting process can only resume execution after the state has been changed by the other process. Property B then ensures that this state is unambiguously read. Access to the correct data is assured by the requirement that the reader deletes the data after reading it and the writer checks that the old data has been deleted before writing new data. Together with properties C and D this guarantees that no old data from the previous iteration can be read.

5 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 [2]. 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.

In a second experiment, a large application written in Java was tested with the well known multiobjective optimizer SPEA2 [5] written in C++ using the library TEA [1]. The purpose of the optimization was the design space exploration of a network processor including architecture selection, binding of tasks and scheduling, see [4]. 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 runtime 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. 2. 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

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

For example, if we take an optimization problem with two objectives $D = 2$, a polling interval of $P = 100\,\text{ms}$, 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.

6 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 components 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 5 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.

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. Emmerich and R. Hosenberg. TEA - a C++ library for the design of evolutionary algorithms. Technical Report CI-106/01, SFB 531, Universität Dortmund, 2000.
2. 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.
3. 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.
4. 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.
5. 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.

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'
```