InPUT: The Intelligent Parameter Utilization Tool

Felix Dobslaw
Dept. of Information, Technology and Media
Mid Sweden University
Östersund, Sweden
felix.dobslaw@miun.se

ABSTRACT

Computer experiments are part of the daily business for many researchers within the area of computational intelligence. However, there is no standard for either human or computer readable documentation of computer experiments. Such a standard could considerably improve the collaboration between experimental researchers, given it is intuitive to use. In response to this deficiency the *Intelligent Parameter Utilization Tool* (InPUT) is introduced. InPUT offers a general and programming language independent format for the definition of parameters and their ranges. It provides services to simplify the implementation of algorithms and can be used as a substitute for input mechanisms of existing frameworks. InPUT reduces code-complexity and increases the reusability of algorithm designs as well as the reproducibility of experiments. InPUT is available as open-source for Java and this will soon also be extended to `C++`, two of the predominant languages of choice for the development of evolutionary algorithms.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*software libraries, modules and interfaces*

Keywords

Intelligent Parameter Utilization Tool, InPUT, InPUT4j, Computer Experiments, Automated Algorithm Design

1. INTRODUCTION

The design and analysis of algorithms is at the heart of evolutionary computation (EC) and the whole of computational intelligence (CI). In contrast to classical research on deterministic algorithms computability theory and complexity theory can only be used to some extent in CI because algorithms involve uncertainty or fuzziness as a feature. Thus, a great deal of CI research heavily depends on experimentation and statistical analysis.

CI research is highly concerned with algorithms for system modeling and the solving of optimization problems. There are many open-source frameworks for different programming languages that offer the user a large set of fully implemented algorithms to tackle any given problems. The user has a significant number of choices and the selection of a suitable algorithm for the specific problem is a non-trivial task in itself. Thus, one of the main objective associated with CI is to obtain a better understanding of the circumstances under which an algorithm is superior to others given a problem and a quality measure. However, the discovery of an appropriate algorithm is not sufficient. Changes in the parameter ranges of a heuristic algorithm can have a significant impact on the experimental outcome [13]. This is true for static parametric choices and as much so, or even more so, for those adapted during execution. As a consequence, a badly parameterized appropriate algorithm could perform significantly worse than a well parameterized inappropriate algorithm. A whole line of research is dedicated to the investigation of parameter and parameter value impact on experimental results [6, 17]. Scientific problems that fall into this remit include the improvement of experimental results given a time budget or conducting comparisons between alternative algorithms. Open-source software exists that supports the experimenter in finding improved parameter settings or algorithm designs [4, 3]. These tools are here referred to as *tuners.*

1.1 Problem Statement

A hurdle for a more rigorous use of experimental design in CI is the absence of a standard format for the description of computer experiments shared among tuners, CI frameworks, and researchers. Figure 1 illustrates the missing link for a smooth and loosely coupled connection between the components. The vertical axis covers the code-independent components *algorithm* and *problem.* The emphasis of the horizontal axis is on the experimental and code-specific components with the *tuner* and the implementation of the algorithm as a *program.*

A common input/output format not only offers technical advantages. The absence of a standard format leads to a higher risk for misunderstandings in publications that can slow down the research work. Every experiment is conducted under certain conditions and is based on assumptions, some of which made explicit by the experimenter, and some not. In general, it is not possible to compare results from researcher A with those received from researcher B, as they are based on different assumptions (algorithm design, computer, implementation, etc.). Thus, experimental

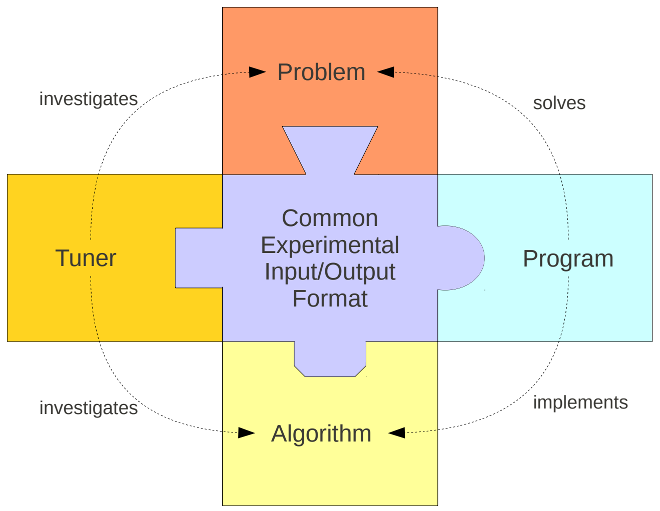


Figure 1: The missing piece of the puzzle for an increased usability, quality assurance, reproducibility, and reusability in computer experimental research.

research in CI requires the reimplementation of algorithms not available in source-code or the turning to a framework that contains an implementation. In any case, all assumptions such as parametric choices have to be identified in the original paper which can be a time consuming task, highly dependent on the paper structure, still leaving open the possibility that some assumptions are not explicitly addressed (the paper is not self-containing). This creates a dependency between the researcher and the author of the publication, which considerably reduces the chances of reproducing the experiments and their results. A standard notion for the description of algorithms and experimental setup would force the author to a concise description of the assumptions and the scope of the investigation.

Notions can be defined with the intention that they will be read by either humans or machines. A human readable standard would lead to a more accessible and therefore simplified reviewing of submissions, and an increased reproducibility of experimental results presented in publications if concise descriptors were associated with them. A computer readable standard would lead to an increased reusability of computer experiments and a further increased reproducibility of experimental results as designs can be imported into programs. It would assist researchers to concentrate on their research questions and could oblige authors to follow a documentation standard for papers to qualify for publication. The development of adapters for different programming languages can further utilize the use of experimental setups and algorithm designs for other causes, such as the automated export to LaTeX tables as authoring support or additional services that hide code-complexity, or automate routine jobs. A common format would be desirable for a number of reasons.

Design Goals

A solution for the center piece in Figure 1 has to be *simple*, *general*, and *open*. Algorithm descriptors have to be intuitive and easy to adjust. Intuitiveness also has to apply for the adapters that support the integration of the standard into programming languages. It would have to work with any programming language, and any existing framework, interchangeably. It should not be restricted to evolutionary algorithms use, but for any open environment enabling computer experiments. Descriptors and adapters should maximize cohesion and minimize coupling between software components, separating concerns. A solution should further contain descriptors that are readable and debatable by researchers. Its use has to be free and the access open.

	Criteria / **Frameworks**	Open Beagle	jMetal	ECJ	OPT4J	ParadisEO (EO)	JCLEC	PISA
1	input descriptors	√	-	√	√	√	√	√
2	usable outside EC	-	-	-	√/-	-	-	√
3	user defined params	√	-	√	√/-	√	√	√
4	hierarchical	√/-	-	-	-	-	-	-
5	XML	√	-	-	√	-	√	-
6	sep. of concern	-	-	-	-	-	-	√
7	compatibility	-	-	-	-	-	-	-
8	validation	√	-	-	-	-	-	-
9	implicit param model	-	-	-	√	-	√	√
10	meta parameter	√	√	√	-	√	-	-
11	multi-language	-	√	-	√	-	-	√

Table 1: A feature matrix that summarizes the compatibility and openness of relevant frameworks with respect to the configuration of computer experiments.

1.2 Contribution

The contribution of this paper is twofold. First, a conceptual solution to the posed problem of a missing standard for the documentation of computer experiments is presented. Second, the *Intelligent Parameter Utilization Tool*, in brief InPUT , is introduced. InPUT offers a general solution for the definition and processing of input data for computer experiments. It simplifies experimental aspects of CI research such as the definition, externalization and expansion of fixed parametric choices in source-code. Unlike work in [11, 22], InPUT does not suggest a meta-language format that requires a complex translation into programming languages. It complements existing frameworks by simplifying the parametrization of algorithms and can further be used as a tool for the experimental exploration of algorithm design spaces, creating a tangible value for researchers. InPUT is not to be considered as yet another EC framework. It does not impose the inheritance of classes or interfaces on the developer or even the use of a specific programming language.

2. RELATED WORK

2.1 Evolutionary Software Frameworks

There are frameworks for the development of evolutionary algorithms, predominantly written in Java and C++. A selection of relevant, open, well documented frameworks under active development with the objective of being generic is analyzed here [20, 23, 18, 10, 12, 15, 19, 8]. Table 1 summarizes their features with respect to customizability, interoperability and openness criteria regarding user input. The frameworks are not judged with respect to any other properties. The judgment is the result of a literature study together with an investigation of the software. The contestants differ in their project focus in terms of target audience, ranging from novices to EC experts, and coverage (genetic programming, multi-objective problems, etc.). Eleven criteria are compared:

1. Can parameters be defined using descriptors?
2. Is the format sufficiently general to be used outside EC?
3. Are user defined parameters supported?
4. Are hierarchical user defined parameters supported?
5. Does the format conform to modeling standards using XML (eXchangeable Modeling Language)?
6. Does the structure enforce a separation of concerns? For instance, a separation of problem and algorithm parameters.
7. Is the format compatible with any other framework?
8. Can parameter descriptors be checked for validity?
9. Does the framework impose a proprietary parameter model on the user?
10. Does the framework offer a *meta parameter*, a service component that encapsulates the programmatic parameter lookup and update?
11. Is the framework available for multiple languages?

Within the scope of this paper there is insufficient space for a detailed elaboration of the results. The table shows that the majority of frameworks provide input descriptors. It is only Open Beagle that supports a complex means of defining hierarchical parameters. Most frameworks define their own proprietary flat descriptor types and do not take account of XML technologies. It is again the case that it is only Open Beagle that offers an explicit schema descriptor allowing for a direct user input validation. PISA is the only framework that proposes a separation of concern for input data, namely the separation of problem and algorithm. It appears that only three frameworks offer a loose coupling between parameter descriptors and the framework components. OPT4J performs this by means of injection, while PISA imposes a seperation of concerns. None of the existent descriptor models can be directly compatible with any other[1]. As a consequence, an exchange and reuse of descriptor files for other frameworks is not possible.

2.2 New Experimentalism and Tuners

In [6], the author presents a methodology for experimental design, termed New Experimentalism. The methodology suggests an approach to experimental design by a sound conduct of computer experiments by using hypothesis testing and a statistical analysis of the results for EC research. Other pivotal methodological contributions to heuristic research include [5, 14, 7]. The integration of philosophy of science, classical experimental design and design of computer experiments has resulted in best practices and guidelines for experimental research in EC [7, 21]. Practical issues addressing the reuse of experimental configurations in contexts other than tuning, such as authoring support, or as a means for standardization in connection with EC frameworks, have not been discussed in that realm. Describing parameter ranges explicitly by design space descriptors allows for user input validation, and prepares the algorithm for a direct use in an experimental design. XML offers a standardized semi-structured grammar to define documents that can be validated. Neither of the two widely used tuners [4, 3] adapt XML for input definition. Both offer proprietary formats raising the bar for beginners.

[1]The fact that Paradiseo builds on EO does not qualify here.

3. METHOD

3.1 Modeling Computer Experiments

The class diagram in Figure 2 models the context of a computer experiment. A computer experiment is here composed of four inputs:

Instance: A problem instance to be solved. Examples here are real valued functions or instances of discrete optimization problems, such as the travelling salesman.

Program: An implementation of an algorithm of choice. For instance, a genetic algorithm (GA) written in Java.

Design: A complete and valid set of parameter-value pairs compatible with the implementation. A common parameter for GA is its population size.

Preferences: A complete and valid set of parameter-value pairs compatible with the implementation, connecting problem and algorithm. The representation of the genome, the fitness function, or the recombination operator for GA are examples.

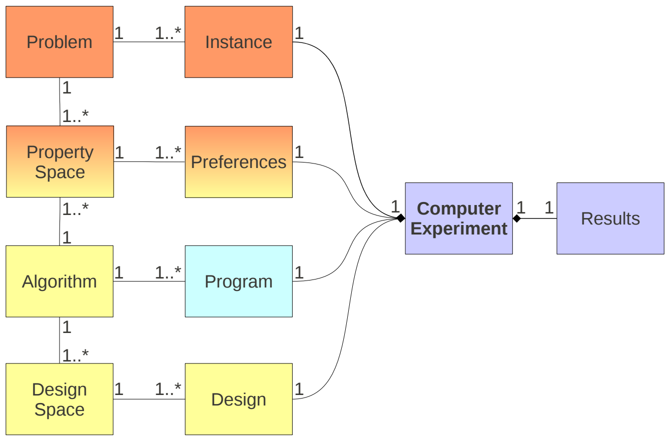


Figure 2: A class diagram making explicit the data that a computer experiment comprises.

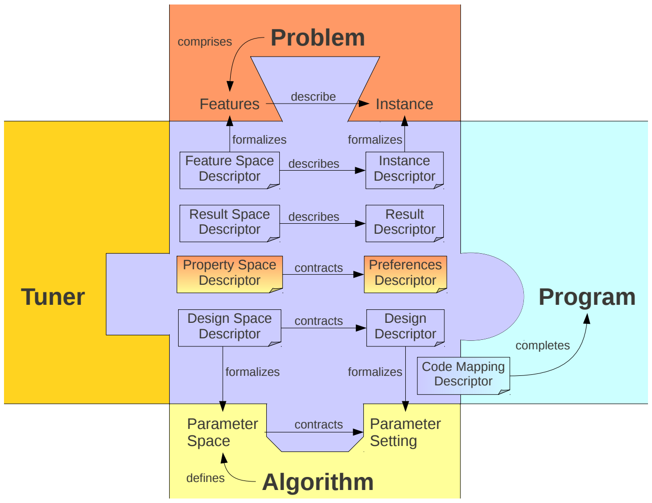
A computer experiment is further considered to possess one output, an arbitrarily formed result set. Output structures are not considered for a more in-depth analysis in this paper.

It is a common procedure to extract features from problem instances for statistical analysis and supervised learning purposes. One of the main differences between algorithm parameters and problem features is that the second do not define degrees of freedom for a user, they are merely a means to describe instances. Generally, a user chooses the problem instance to be solved. Algorithm parameters on the contrary expose options to the user. Figure 3a presents a general input/output format proposal for computer experiments. It comprises the following descriptors:

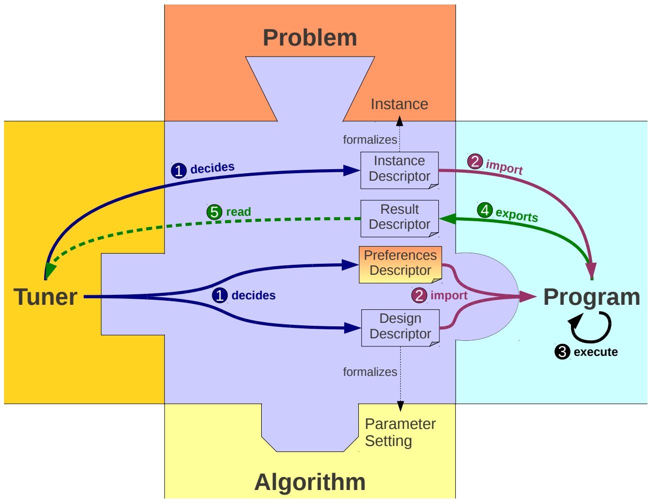
Algorithm Design Space: The (heterogenous) set of algorithm parameters together with their range definitions.

Algorithm Design: An exemplar of a design space for a given algorithm.

Problem Feature Space: The (heterogenous) set of problem features with optional range definitions.



(a) Descriptors ensure a minimal coupling between components.



(b) The recurring steps of a tuner under the use of input/output descriptors.

Figure 3: A conceptual proposal of a data input/output standard for the conduct of computer experiments.

Problem Instance: An exemplar of the feature descriptor, extracted from a problem instance.

Property Space: The (heterogenous) set of parameters that connect problem and algorithm including their range definitions.

Preferences: An exemplar of the property space descriptor for a given algorithm-problem combination.

Result Space: Describes the structure of result sets.

Result: An exemplar of a result that follows a result space structure.

Code Mapping: Maps general concepts from the design space and property space descriptors to code specific components of the program.

In a one user scenario well separated descriptors are not necessary. Proprietary, non-separated, or implicit settings in source-code imply a short cycle from an algorithm sketch to obtaining the first results. A mapping from code-independent to code-dependent concepts is not necessary, because no-one else reuses the experimental context. The separation of concerns introduces overhead. However, the proposed structure allows third parties to not only read and understand the configurations (making them *accessible*), but also to import them into their software (making them *reusable*). As a consequence the validation of a result from a third party becomes considerably easier.

Figure 3b illustrates the way a tuner and a program can interact in a sequential repetitive manner by exchanging descriptors. In 1, the tuner announces instance, preferences, and algorithm design to be tested and sends a signal once they are readily available. The setup is imported by the program in step 2, and the experiment is executed in step 3. The result is returned by the program in step 4 and the program signals its successful completion. Now, the tuner can, with or without the consideration of the feedback from the experimental result (step 5), decide about the next experiment to be executed or stop the investigation.

Type	Range
`boolean`	$\{0,1\}$
`integer`	$\mathbb{Z} \cap [-2^{31}, 2^{31}-1]$
`short`	$\mathbb{Z} \cap [-2^{15}, 2^{15}-1]$
`long`	$\mathbb{Z} \cap [-2^{63}, 2^{63}-1]$
`float`	$\mathbb{R} \cap [0,1]$ (32-bit `fp`)
`double`	$\mathbb{R} \cap [0,1]$ (64-bit `fp`)
`decimal`	$\mathbb{R}$

Table 2: The numerical parameter types supported by InPUT (`fp` stands for floating points).

3.2 InPUT

In an extension of the proposal above, and unlike existing approaches to modeling in EC [16], InPUT provides three grammar types, sufficiently general to describe all of the nine mentioned descriptors. They are presented in Extended Backus Naur Form (EBNF). In addition, InPUT offers an interface and reference implementation for their programmatic access. Examples for each grammar are given in the next section. It follows an overview regarding how parameters are modeled using InPUT .

3.2.1 Parameters

InPUT distinguishes between two parameter types: numerical and structural. Table 2 lists the supported numerical types with their default ranges. These ranges can be further restricted by the explicit setting of inclusive or exclusive extreme values. An example of a numerical parameter is the *population size* of a traditional evolutionary algorithm represented by a positive integer value range.

The fact that parameters can contain sub-parameters, even algorithms, increases the complexity of the modeling problem. The hyper-heuristic community deals with the creation and analysis of algorithms that combine heuristics in arbitrary ways [9]. With due regard to general demands, structural parameters are introduced. Structural parameters, as opposed to numerical ones, define a range of distinct complex choices that themselves can introduce parameters which have to be explicitly set by a user. An example is the *selection strategy* parameter for evolutionary algorithms. A possible choice here is *tournament selection*, introducing the two numerical sub-parameters *tournament size* and *amount of survivors*.

With structural parameters, the descriptor turns out to be a hierarchy of parameters or a parameter tree with the algorithm identifier as the root. The unambiguous identification of parameters is realized by a hierarchical notation with single dot delimiters. Examples for the parameters mentioned above are *"EA.PopSize"* for population size, *"EA.Selection"* for selection, and *"EA.Selection.Tournament.Size"* as well as *"EA.Selection.Tournament.Survivors"* for the tournament size and amount of survivors respectively.

Further, matrix or array types for both numerical and structural parameters are supported. Those can be of fixed or undefined size. An example of a valid type definition would be `decimal[2]`, which determines the parameter values to be a vector of fixed length containing two decimal entries. `boolean[][5][]` would then be a $n \times 5 \times m$ matrix of type boolean, with $n, m \in \mathbb{N}$ unspecified.

3.2.2 Grammars

InPUT defines three grammars. The design space grammar is a meta-structure that can be used to model the four descriptors to the left of Figure 3a. A *design space* is a code-independent aggregation of potentially heterogenous parameter (or property) ranges.

Grammar 1. The design space grammar:

$\langle\text{DesignSpace}\rangle \rightarrow \{\langle\text{Param}\rangle\}^{+}$
$\langle\text{Param}\rangle \rightarrow (\texttt{sParamId}\ \langle\text{SParam}\rangle) \mid \texttt{nParamId}$
$\langle\text{SParam}\rangle \rightarrow \{\langle\text{DesignSpace}\rangle\}\ \{\langle\text{SChoice}\rangle\}^{+}$
$\langle\text{SChoice}\rangle \rightarrow \texttt{sChoiceId} \mid (\texttt{sChoiceId}\ \langle\text{DesignSpace}\rangle)$

`nParamId` and `sParamId` stand for numerical and structural parameter identifier respectively. `sChoiceId` stands for structural choice identifier. Terminals start with small, non-terminals with capital letters.

The design grammar represents instances that can be validated given design spaces. Each instance is contracted by a grammar of type one. The instance, algorithm design, and preference descriptors from Figure 3a can be expressed by a type two grammar.

Grammar 2. The design grammar:

$\langle\text{Design}\rangle \rightarrow \{\langle\text{Value}\rangle\}^{+}$
$\langle\text{Value}\rangle \rightarrow \langle\text{SValue}\rangle \mid (\texttt{nParamId nValue})$
$\langle\text{SValue}\rangle \rightarrow (\texttt{sParamId sChoiceId}\ [\langle\text{Design}\rangle])$

Here, `nValue` stands for numerical value. XML makes descriptors machine-readable, but the code-independent descriptors require a means to become machine-interpretable by programs. The code mapping grammar is a dictionary which builds the connection between the design spaces and a program. It servers as a dictionary for the translation from concept to component. The code mapping descriptor from Figure 3a can be expressed by the code mapping grammar.

Grammar 3. The code mapping grammar:

$\langle\text{Mappings}\rangle \rightarrow \{\texttt{paramId componentId}\}$

Code mappings are only required for structural parameters. XML has a native support for primitive data types. Components in object oriented languages tend to be classes.

3.2.3 Integration

The definition of descriptors is not self-sufficient, there must be a means of importing them for different programming languages. It is possible that framework developers could work on interfacing the descriptors. However, a more general solution is desired that offers services to framework providers so that it is easy to incorporate the input mechanism. Adapters on a language basis would offer a more general solution. These adapters could offer a standard application programming interface (API) to the user, covering, amongst other things, descriptor import, export, validation, and assembly to describe computer experiments. A reference implementation is presented below after the presentation of the example descriptors.

4. EXAMPLE

The example descriptor in Figure 4 defines an algorithm design space for a real-coded Particle Swarm Optimization (PSO) algorithm using InPUT syntax. The algorithm is largely inspired by the tutorial in [1] and the standard components for real-coded PSO in ParadisEO. Figure 5 shows a valid

```xml
<Design id="1" ref="realPSO" ...>
  <NValue id="Seed" value="524521106532245"/>
  <NValue id="PopSize" value="39"/>
    <NValue id="VelMin" value="-0.2307"/>
    <NValue id="VelMax" value="1.1992"/>
    <NValue id="InitPosMin" value="-2.6058"/>
    <NValue id="InitPosMax" value="3.8540"/>
    <NValue id="InitVelMin" value="-2.4382"/>
    <NValue id="InitVelMax" value="0.7912"/>
  <SValue id="VelocityType" value="Constricted">
    <SValue id="Topology" value="Linear">
        <NValue id="Neighborhood" value="5"/>
    </SValue>
    <NValue id="Inertia" value="1.4249"/>
    <NValue id="LFactor1" value="0.8832"/>
    <NValue id="LFactor2" value="4.0326"/>
  </SValue>
  <SValue id="StopCriterion" value="Time">
    <NValue id="Ms" value="2435"/>
  </SValue>
</Design>
```

Figure 5: A valid random design for the real-coded PSO from Figure 4 generated using InPUT4j with a decimal value precision of 4 positions for illustration purposes.

instance of the design space in Figure 4, generated using the design randomizer of the reference implementation that can be used to generate arbitrarily large sets of random designs. Table 3 shows how the reference implementation exports this design space to a LaTeX table. An intact code mapping file with structural *parameter-to-class* and *choice-to-class* mappings as in

```
<Mapping id="StopCriterion" type="package.StopCriterion"/>
```

is all InPUT requires to create fully instantiated instances at runtime. These code mappings could appear different for different languages, but within one language a standard format should be established.

```xml
<DesignSpace id="realPSO" ...>
    <NParam id="Seed" type="decimal" />
    <NParam id="PopSize" type="integer" inclMin="2" inclMax="50"/>
    <NParam id="InitPosMin" type="decimal" inclMin="-4" inclMax="-1"/>
    <NParam id="InitPosMax" type="decimal" inclMin="1" inclMax="4"/>
    <NParam id="InitVelMin" type="decimal" inclMin="-3" inclMax="-0.5"/>
    <NParam id="InitVelMax" type="decimal" inclMin="0.5" inclMax="3"/>
    <NParam id="VelMin" type="decimal" inclMin="-4" inclMax="-0.5"/>
    <NParam id="VelMax" type="decimal" inclMin="0.5" inclMax="4"/>
    <SParam id="VelocityType">
        <SParam id="Topology">
            <NParam id="Neighborhood" type="integer" inclMin="2" inclMax="10"/>
            <SChoice id="Ring"/>
            <SChoice id="Linear"/>
        </SParam>
        <NParam id="Inertia" type="decimal" inclMin="0" inclMax="5"/>
        <NParam id="LFactor1" type="decimal" inclMin="0.1" inclMax="3"/>
        <NParam id="LFactor2" type="decimal" inclMin="0.1" inclMax="5"/>
        <SChoice id="Standard"/>
        <SChoice id="Constricted"/>
        <SChoice id="FixedInertia"/>
        <SChoice id="VariableInertia"/>
        <SChoice id="Extended">
            <NParam id="LFactor3" type="decimal" inclMin="0" inclMax="5"/>
        </SChoice>
    </SParam>
    <SParam id="StopCriterion">
        <SChoice id="Time">
            <NParam id="Ms" type="integer" inclMin="1000" exclMax="10000"/>
        </SChoice>
        <SChoice id="Generations">
            <NParam id="Amount" type="integer" inclMin="2" inclMax="100"/>
        </SChoice>
        <SChoice id="SteadyFit">
            <NParam id="MinGen" type="integer" inclMin="10" inclMax="50"/>
            <NParam id="Interval" type="integer" inclMin="1" inclMax="20"/>
        </SChoice>
    </SParam>
</DesignSpace>
```

Figure 4: The algorithm design space descriptor for a real-coded PSO. The numerical extreme values only serve the purpose of demonstration and no claim is made here that they are correct or meaningful.

Parameter	**Type**	**Range**
InitPosMax	`decimal`	[1, 4]
InitPosMin	`decimal`	[-4, -1]
InitVelMax	`decimal`	[0.5, 3]
InitVelMin	`decimal`	[-3, -0.5]
PopSize	`integer`	[2, 50]
Seed	`decimal`	]-∞, ∞[
StopCriterion	`structured`	{Time, Generations, SteadyFit}
Generations.Amount	`integer`	[2, 100]
SteadyFit.Interval	`integer`	[1, 20]
SteadyFit.MinGen	`integer`	[10, 50]
Time.Ms	`integer`	[1000, 10.000[
VelMax	`decimal`	[0.5, 4]
VelMin	`decimal`	[-4, -0.5]
VelocityType	`structured`	{Standard, Constricted, FixedInertia, VariableInertia, Extended}
Inertia	`decimal`	[0, 5]
LFactor1	`decimal`	[0.1, 3]
LFactor2	`decimal`	[0.1, 5]
Topology	`structured`	{Ring, Linear}
Neighborhood	`integer`	[2, 10]
Extended.LFactor3	`decimal`	[0, 5]

Table 3: The parameter ranges of the PSO algorithm from Figure 4 exported to a LaTeX table using InPUT4j.

5. IMPLEMENTATION

The reference implementation, InPUT4j, is written in Java. It can be downloaded together with the descriptors from [2]. A `C++` variant is under development. InPUT4j makes use of an XML parser to process the descriptors and their element trees. It uses plain Java reflection for the automatic object instantiation of user defined (proprietary) classes at runtime, unknown to InPUT4j at compile time. The interface allows their direct integration by softening Java's strong typing using generics. InPUT follows a minimal, object oriented, design. It offers an API and a command line interface.

5.1 Features

InPUT4j imports and validates design space, design, and code mapping descriptors, making their content accessible in Java. It offers direct access to parameters via a meta parameter, allowing for value queries and dynamic updates. The InPUT meta parameter encapsulates all parameter values and their ranges in a single variable to reduce code complexity and can comprehensively be used in formulas following the Java standard syntax. Only small code and descriptor adjustments are required to externalize fixed values in the source code as parameters, providing high experimental flexibility with little programming efforts. Snapshots of the present design can be issued at any time and exported as XML files. Algorithm design and design space descriptors can be exported to LaTeX tables for authoring support. InPUT supports exception handling and the logging of relevant events (value updates, errors, etc.). The creation and manipulation of algorithm spaces and designs can be conducted in support of any graphical XML editor of choice.

In addition, InPUT supports the automated creation of random parameter values or entire algorithm designs via its randomization services given a design space descriptor. It supports the random creation of arbitrarily sized arrays given a parameter type and array size information.

5.2 Code Mapping Contract

The injection of sub-parameter values into instances of their structural parent parameters is based on a simple contract between InPUT and the user thus ensuring a minimal coupling. On top of the setting of class identifiers, two assumptions can be made explicit in the code mapping descriptor. By default, it is assumed that classes follow the recommended Java interface nomenclature for setters and getters. For each sub-parameter, the parent class is expected to provide the public methods *"set + paramId"* and *"get + paramId"* (e.g., *"setSurvivors"* for tournament selection). The types are automatically derived from the design space descriptor at runtime. Getter and setter names can be customized in the code mapping descriptor. InPUT4j supports autoboxing between primitive types and their class equivalents. The default setup for class instantiation is that structured parameters use constructors with no formal parameters. This can also be customized. Thus, InPUT requires no change in existing code, given that it is modular in that two structural parameters of the same type share the same interface or superclass and use the same formal constructor parameters. For instance, the two selection procedures, *Tournament* and *Rank*, are considered to inherit a common interface of some user defined supertype, say *Selection*. If *Tournament* requires properties other than its sub-parameters to be set, those properties should generally also be set in the constructor for *Rank*. Thus, the mapping file would require the user to add the constructor signature in the *Selection* entry of the descriptor. How this is performed in practice is explained and exemplified on the InPUT project page [2].

6. DISCUSSION

The availability of frameworks in EC is of importance. Programming can result in unresolved bugs and a shared implementation reduces the likelihood that they remain as a basis for fraudulent results. For the programming language and code-independent documentation of experiments and parameter tuning, however, a more general solution is required. The fact that many EC frameworks coexist might suggest that none of them have yet achieved perfection. One potential reason is the complexity of the problem based on the perceived requirements of the end users. A user requests simplicity, generality, and an end-to-end solution. Simplicity and generality must achieve a compromise throughout multiple dimensions (parameter treatment, usability, API support, output format, etc.). To deliver this in its entirety is architecturally non-trivial and requires on top of that a lot of software development. The investigated frameworks are modular, but their modular approach is not carried forward into their launch strategy. The consequence are close solutions largely incompatible with one another. For instance, JCLEC and OPT4J both arrive with graphical user interfaces, launched within the scope of the project and not announced as a single component or service for reuse in other frameworks. Individual launches of components would require a thorough analysis of the interfaces and formats, serving as a driver for cooperation. Developers are able instead to narrow their scope to components, probably leading to better and more focused components. However, at some point the parts are required to be jointed and this should not be conducted by the user, but as a common effort by the community in bundles. It is possible to make some comparisons with the different distributions of integrated development environments such as Eclipse or the operating systems based on a Linux kernel, containing those components considered appropriate for a user profile. In that light, InPUT is a proposal for input treatment and one possible cornerstone for experimental documentation.

Returning to feature Table 1, InPUT presents a solution that satisfies, to a large extent, the majority of the posted criteria. It offers input descriptors that can be used for the description of algorithms and problems in EC, or any other area of research concerned with computer experiments. It allows for user defined parameters of any kind, offering three grammar types with a minimal syntax, which are made machine and human readable using XML. The validation of input therefore comes for free. The model for separation of concerns from Section 3.1 is taken up by InPUT in order to support the user in the definition of all input related data with respect to computer experiments. InPUT offers an implicit parameter model that does not impose any inheritance of concepts on the user. It offers a meta parameter with an API that allows programmers to interact with parameter descriptors programmatically. InPUT offers a conceptual framework that in principle can be adapted to any existing framework, given it is modular. It can therefore not be claimed that it is compatible to the current frameworks, but it requires a minimum of effort to make them all compliant to one another. In its current state, its full use is reduced to Java. However, the command line tool allows the creation and validation of descriptor files on any platform, because Java is platform independent. XML is also platform independent, so in principle everyone with access to an XML parser can comfortably access the InPUT descriptors programmatically. With respect to the design goals in Section 1.1 it is not possible to provide a complete evaluation. The design is created with the intention of being as simple, general and open as possible, which is non-trivial in the face of a very general challenge. InPUT4j is developed with its focus on a lightweight API for an intuitive use. Once adapted, InPUT would work with any programming language and existing framework and would not be restricted to EC. A maximized cohesion and minimized coupling between components would be achieved.

InPUT can be used to compare novel algorithms for benchmark or real world problems on entirely random design spaces, as it creates them by a single method call. Hundreds or thousands of random designs can be created in a few seconds on a common laptop for Monte Carlo simulations. Monte Carlo simulations can be used to collect data as a basis for variance analysis, where statistical indexes can assist in distinguishing those parameters which have a significant impact on the result from those which have a low impact, for instance by using *Analysis of Variance* (ANOVA).

Tuners can use the InPUT descriptors for their own configuration, making it easier to compare them on common grounds and problem sets in fulfillment of their task. This, in return, would enable the entering threshold for the use of parameter tuners to be lowered because of the familiar syntax. For research in dynamic control of parameter values InPUT offers a way to issue and record changes. The recorded data can then be used for a theoretical analysis or reinforcement learning.

7. CONCLUSIONS

A conceptual framework together with its implementation, termed InPUT , were introduced, addressing the lack of a standard for the documentation of computer experiments. InPUT simplifies the definition of computer experiments by proposing code-independent descriptors. InPUT fulfills the majority of compatibility criteria posted in Section 1.1 and Table 1 to a large extent, as discussed in Section 6. It provides services for simplified import and export to and from existing EC frameworks. It assists the experimenter to define a machine and human readable context for computer experiments. Data can be exported for publication support and experimental documentation. Most importantly, researchers as well as reviewers and readers must be released from time consuming implementation or information retrieval tasks that can be automated, which thus allows them to focus on the thorough execution of experiments and the analysis of the results. In that respect, InPUT could be an initial step to make frameworks and researchers speak and understand the same language.

Future Work

InPUT does not cover output formats for computer experiments. An investigation of existing formats and requirement assessment for experimental results are possibilities for future work. InPUT is open-source under the MIT license. Enthusiasts are welcome to contribute to its development, either by taking part in the features discussion, or by developing adapters for other popular programming languages for EC, such as Python or `C#`. Frequently, parameter ranges are restricted by value constraints including those in-between parameters (*inter-parameter constraints*). InPUT can be extended to express those constraints both for value validation and random design creation. Naming conventions for descriptors is a topic that has to be discussed and, in addition, the export to databases or spread-sheets for record keeping and statistical analysis purposes. Making descriptors available along with publications on home pages or preferably the inclusion in research databases is encouraged. InPUT and its grammars should not be taken as final, but, as a draft proposal.

8. ACKNOWLEDGMENTS

The author would like to thank Mr. Peter Edsbäcker for prolific discussions that contributed to the work on InPUT .

9. REFERENCES

[1] Implement a real-coded pso algorithm using paradiseo-eo, `http://paradiseo.gforge.inria.fr`, 2012.

[2] The intelligent parameter utilization tool, `http://TheInPUT.org`, 2012.

[3] Sequential model-based algorithm configuration, `http://cs.ubc.ca/labs/beta/Projects/SMAC`, 2012.

[4] Sequential parameter optimization tool, `http://gociop.de/research-projects/spot`, 2012.

[5] R. Barr, B. Golden, J. Kelly, M. Resende, and W. Stewart. Designing and reporting on computational experiments with heuristic methods. *Journal of Heuristics*, 1:9–32, 1995.

[6] T. Bartz-Beielstein. *New Experimentalism Applied to Evolutionary Computation*. PhD thesis, Universität Dortmund, Fachbereich Informatik, April 2005.

[7] T. Bartz-Beielstein and M. Preuss. Experimental research in evolutionary computation. In *Proceedings of the 2008 GECCO conference companion on Genetic and evolutionary computation*, pages 2517–2534. ACM, 2008.

[8] S. Bleuler, M. Laumanns, L. Thiele, and E. Zitzler. PISA - a platform and programming language independent interface for search algorithms. In *Evolutionary Multi-Criterion Optimization (EMO 2003)*, Lecture Notes in Computer Science, pages 494 – 508, Berlin, 2003. Springer.

[9] E. Burke, M. Hyde, G. Kendall, G. Ochoa, and R.Qu. Hyper-heuristics: A survey of the state of the art. *Technical Report, School of Computer Science and Information Technology, University of Nottingham*, pages 1–52, 2010.

[10] S. Cahon, N. Melab, and E.-G. Talbi. Paradiseo: A framework for the reusable design of parallel and distributed metaheuristics. *Journal of Heuristics*, 10:357–380, 2004.

[11] P. Collet, E. Lutton, M. Schoenauer, and J. Louchet. Take it easea. In *Parallel Problem Solving from Nature PPSN VI*, volume 1917 of *Lecture Notes in Computer Science*, pages 891–901. Springer Berlin / Heidelberg, 2000.

[12] J. J. Durillo and A. J. Nebro. jmetal: A java framework for multi-objective optimization. *Advances in Engineering Software*, 42(10):760 – 771, 2011.

[13] A. Eiben and S. Smit. Parameter tuning for configuring and analyzing evolutionary algorithms. *Swarm and Evolutionary Computation*, 1(1):19–31, 2011.

[14] O. Francois and C. Lavergne. Design of evolutionary algorithms-a statistical perspective. *IEEE Transactions on Evolutionary Computation*, 5(2):129–148, 2001.

[15] C. Gagne and M. Parizeau. Open beagle: A new versatile `C++` framework for evolutionary computations. In *Proceedings of the 2002 GECCO conference companion on Genetic and evolutionary computation*, pages 161–168, 2002.

[16] J. Guervos, P. Valdivieso, G. Lopez, and M. Arenas. Specifying evolutionary algorithms in xml. In *Computational Methods in Neural Modeling*, volume 2686 of *Lecture Notes in Computer Science*, pages 1042–1043. Springer Berlin / Heidelberg, 2003.

[17] F. Hutter. *Automated Configuration of Algorithms for Solving Hard Computational Problems*. PhD thesis, University of British Columbia, Department of Computer Science, October 2009.

[18] M. Keijzer, J. Merelo, G. Romero, and M. Schoenauer. Evolving objects: A general purpose evolutionary computation library. In *Artificial Evolution*, volume 2310 of *Lecture Notes in Computer Science*, pages 829–888. Springer Berlin / Heidelberg, 2002.

[19] M. Lukasiewycz, M. Glaß, F. Reimann, and J. Teich. Opt4J - A Modular Framework for Meta-heuristic Optimization. In *Proceedings of the 2011 GECCO conference companion on Genetic and evolutionary computation*, Dublin, Ireland, 2011.

[20] S. Luke. The ecj owner's manual. Technical report, George Mason University, 2010.

[21] M. Preuss. Reporting on experiments in evolutionary computation, 2007.

[22] D. Steve and C. J. Woodward. Esdl: a simple description language for population-based evolutionary computation. In *Proceedings of the 2011 GECCO conference companion on Genetic and evolutionary computation*, pages 1045–1052. ACM, 2011.

[23] S. Ventura, C. Romero, A. Zafra, J. A. Delgado, and C. Hervas. Jclec: A java framework for evolutionary computation. *Soft Computing*, 12(4):381–392, 2008.