University of British Columbia
Department of Computer Science
Technical Report TR-2010-14

Programming by Optimisation

Holger H. Hoos
University of British Columbia
Department of Computer Science

24 December 2010

Abstract

When creating software, developers often explore various ways of achieving certain tasks. Traditionally, these alternatives are eliminated or abandoned early in the process, based on the idea that the flexibility afforded by them would be difficult or impossible to exploit effectively by developers or users. Here, we challenge this view and advocate an approach that encourages developers to not only avoid premature commitment to certain design choices, but to actively develop promising alternatives for parts of the design. In this paradigm, dubbed *Programming by Optimisation (PbO)*, developers specify a potentially large design space of programs that accomplish a given tasks, from which then versions of the program optimised for various application contexts are obtained automatically; per-instance selectors and parallel portfolios of programs can be obtained from the same design space (*i.e.*, from exactly the same sources). We describe a simple, generic programming language extension that supports the specification of such design spaces and discuss ways in which specific programs that perform well in a given use context can be obtained from these specifications by using relatively simple source code transformations and powerful design optimisation methods. Using PbO, human experts can focus on the creative task of thinking about possible mechanisms for solving given problems or subproblems, while the tedious and boring task of determining what works best in a given use context is performed automatically, substituting human labour by computation. We believe that PbO provides an attractive way of creating software whose performance can be effectively adapted to a wide range of use contexts; it also enables the principled empirical investigation of the impact of design choices on performance, of the interaction between design choices and of the suitability of design choices in specific use contexts.

1 Introduction

In computer programs and the algorithms they are based on, just as in everyday's life, there are frequently different ways of getting a task done. While sometimes, certain choices are clearly preferable over others, it is often unclear which of several design decisions will ultimately give the best results. Such design choices can, and often do occur at many levels, from high-level architectural aspects of a software system to low-level implementation details. They are often made based on considerations of maintainability, extensibility

and performance of the system or program under development, and it is this latter aspect that we focus on in this work. In doing so, we further consider only sets of semantically equivalent design choices, *i.e.*, situations in which only the performance of the program depends on the decisions made for each aspect of the program for which one or more candidate designs are available, while the program's correctness and functionality is not affected by those choices. This premise differs fundamentally from that of program synthesis, where the primary goal is to come up with a design that satisfies a given functional specification.

We note that even for relatively small pieces of software, such design choices are considered and made at several points, and the effects of these choices on overall performance are often not independent. Furthermore, there are typically components with one or more parameters whose settings can have a substantial impact on their performance on various types of data.

In the past, such design choices have typically been made to a large extent by the software developer (or development team). In some cases, the choice between various alternatives is made at development stages preceding the generation of actual code, and in others, design decisions have far-reaching effects on many other choices (as is the case, for example, when deciding on higher-level architectural aspects of a system or on specific data structures that are widely used within a larger piece of software). But often, one of several design alternernatives is chosen at or after the implementation stage, and those choices, while not severely constraining other parts of the system, may have a substantial impact on overall performance.

Occasionally, decisions of this latter type are deferred to a post-implementation stage, and sometimes they are left to the user of a system, by exposing them as parameters.[1] More often, however, they are hard-coded, either by means of constants within a program or module, or by retaining some pieces of code, while abandoning alternative ones. Especially when implementing heuristic mechanisms, these design choices are ususally made based on intuition, experience and perhaps a limited amount of *ad-hoc* experimentation with various alternatives.[2]

In our experience in the area of designing high-performance heuristic solvers for various hard combinatorial problems, building software in this manner leads to suboptimal results in terms of performance and adaptability to different use contexts. We therefore advocate an approach in which many design choices are deliberately left open, by means of retaining alternative realisations of components or mechanisms, and by exposing a large number of parameters (see, *e.g.*, Hutter et al., 2007a; Hoos, 2008; KhudaBukhsh et al., 2009). These choices are then made by means of running a *meta-algorithmic optimisation procedure*, which optimises the empirical performance achieved in a given use context. In the case of heuristic solvers for $\mathcal{NP}$-hard problems, this context is characterised by a set of benchmark instances deemed typical for the intended application; more generally, use contexts correspond to a set of benchmark inputs representative of those encountered in a particular situation in which a given program is used.

We call this approach *Programming by Optimisation (PbO)*; it is fundamentally based on the idea to avoid premature commitment to certain design choices, and to actively develop promising alternatives for parts of the design.[3] Thus, rather than building a single program for a given purpose, software developers specify a rich and potentially large design space of programs. From this specification, programs that perform well in a given use context are generated automatically, by means of powerful optimisation techniques. This approach allows human experts to focus on the creative task of thinking about possible mechanisms for

[1] It may be noted that many users, especially those without deeper insights into the program or system under consideration, tend to keep these parameters at their default values.

[2] We note that the use of heuristic techniques does not imply that the resulting programs do not have any provable performance guarantees, but often results in empirical performance far better than the bounds guaranteed by rigorous theoretical analysis.

[3] The fact that the acronym PbO is identical to the chemical formula for lead(II) oxide is not entirely coincidental. Lead(II) oxide is the key component of lead glass, also known as lead crystal – which, because of its attractive optical properties, has been used extensively in the production of art glass and glassware for at least 1000 years; it is also employed for the imitation of precious stones. Through its use in lead glass and lead glazes, the substance also has various industrial and technological applications.

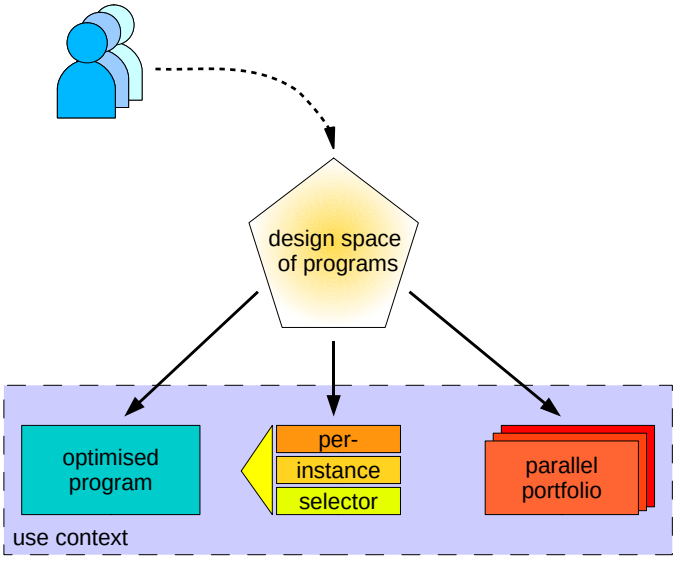

Figure 1: Programming by Optimisation (PbO): Developers (or teams of developers) specify a potentially large design space of programs that solve a a given problem, from which then versions of the program optimised for various application contexts are obtained automatically; per-instance selectors and parallel portfolios of programs can be obtained from the same design space (*i.e.*, from exactly the same sources).

solving given problems or subproblems, while the tedious and boring task of determining what works best in a given use context is performed automatically, substituting human labour by computation. Furthermore, more complex designs, such as per-instance selectors (see, *e.g.*, Leyton-Brown et al., 2003; Guerri and Milano, 2004; Xu et al., 2008) and parallel portfolios (see, *e.g.*, Huberman et al., 1997; Gomes and Selman, 2001), can be generated automatically from the same design space specification (and sources), as illustrated in Figure 1.

We note that this approach relies on appropriate support for specifying design spaces and on powerful procedures for searching those spaces for performance optimising designs, along with the computational resources required for carrying out that optimisation process. In the following, we briefly discuss each of these requirements along with specific proposals on how they can be met; but before delving into the particulars of supporting programmers in following the PbO paradigm, we will explain how we expect PbO to change the way in which we create and use software.

2 How PbO changes software development and deployment

We first employed, unwittingly, the key idea behind PbO in the context of collaborative work on SAT-based software verification (Hutter et al., 2007a). The use of off-the-shelf solvers for the propositional satisfiability problem (SAT) has become one of the standard approaches for formally verifying hard- and software. In this project, the idea was to produce a SAT solver that would be particularly well-suited for dealing with the SAT instances produced by a specific static checker, CALYSTO (Babić and Hu, 2007). In initial stages of the work (carried out by Domagoj Babić and Alan Hu), a new SAT solver dubbed SPEAR had been developed, which included a wide range of techniques developed and demonstrated to be useful by

the SAT community. Because it was unclear which combination of techniques would prove most effective for solving the SAT instances produced by CALYSTO, the initial version of SPEAR could be configured quite flexibly, via parameters exposed to the user. On the other hand, finding settings for these parameters that would result in good performance of the solver on the instances of interest proved to be very challenging even for its primary designer, Domagoj Babić. Therefore, it was decided to use the automated algorithm configuration procedure ParamILS, which had just been developed by Hutter et al. (2007b), to accomplish this task.

What happened next can be seen as the genesis of the PbO paradigm: *After seeing how much better the configurations of* SPEAR *found by ParamILS performed than his manually determined default settings, Domagoj decided to expose additional design choices.* Some of these had been previously hardwired into the solver; others had been implemented, tested and then abandoned; and yet others were newly implemented alternatives to existing mechanisms within SPEAR. This ultimately lead to a version of SPEAR that could be configured via 26 parameters, which jointly gave rise to 8.34×10^{17} configurations of the solver. *ParamILS turned out to be able to achieve speedups of a factor of over 500 for the software verification instances produced by* CALYSTO *compared to the default configuration, which had been manually determined with considerable effort by its designer.* This automatically optimised configuration of SPEAR was entered into and won the QF_BV category of the 2007 Satisfiability Modulo Theories (SMT) Competition. *Moreover, when automatically optimising the same highly parametric version of* SPEAR *for solving SAT-encoded hardware verification instances, substantial speedups over the (then) state-of-the-art solver MiniSAT 2.0 were achieved, and several interesting observations could be made when comparing the configurations specifically optimised for the SAT-encoded hardware and software verification tasks, respectively (Hutter et al., 2007a).*

This early example already displays the two key elements of the PbO approach:

- The specification of large and rich combinatorial design spaces of programs that solve a given problem, by means of avoiding premature commitment to certain design choices and active development of promising alternatives for parts of the design.
- The automated generation of programs that perform well in a given use context from this specification, by means of optimisation techniques that can realise the performance potential inherent in the given design space.

Naturally, these elements can be realised to various degrees, and are present, at least to some extent, in practices that are already used widely within computing science and beyond. Our primary goal in this work is to develop and advocate an approach to software development and, indeed, the solution of computational problems, that explicitly recognises these elements, as well as to provide conceptual support and tools that facilitate advanced forms of programming by optimisation.

Our example also illustrates, at least to some extent, four distinct levels of PbO in software development:

Level 0: Settings of the parameters exposed by an existing piece of software are optimised for a given use context (characterised, for example, by a set of typical input data); this is also known as *parameter tuning*.

Level 1: The design space represented by an existing piece of software is extended by exposing design choices hardwired into code; such choices include certain *magic constants* (*i.e.*, literals changing which could affect performance but not the correct function of the program), *hidden parameters* (*i.e.*, named constants that could take different values without compromising the correctness of the program or variables that are set to constant values, but not exposed as externally accessible parameters)

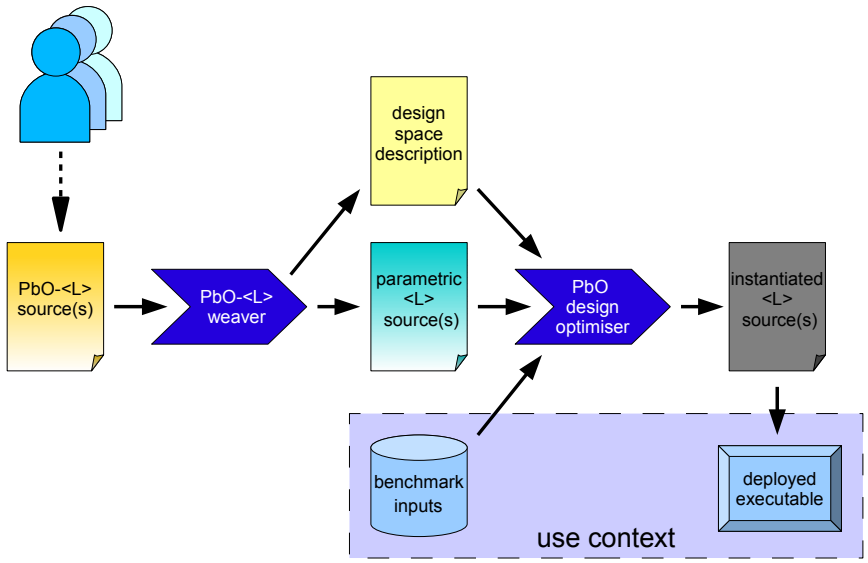

Figure 2: Software development in the PbO paradigm: Developers produce a PbO-enhanced source code in their language of choice, ⟨L⟩, from which the PbO weaver produces parametric source code in pure ⟨L⟩, as well as a description of the design space; next, the PbO design optimiser uses this along with benchmark input data to produce a fully instantiated version of the source that has been optimised specifically for deployment in the given use context.

and *abandoned design alternatives* (*i.e.*, pieces of code that could be used in addition or instead of active code without compromising correctness, but that are no longer reachable during execution of the current version of the program).

Level 2: Design choices considered during the normal course of the software development process are actively kept and exposed.

Level 3: The software development process is structured and carried out in way that actively strives to provide design choices and alternatives in many places of a project.

We also envision a further level, which we believe will be worth striving for:

Level 4: The software development process is centred on the idea of providing design choices and alternatives in all parts of a project that may benefit from them; no design choices that cannot be justified compellingly are made prematurely.

While levels 0 and 1 deal with existing software and essentially involve adding to the development process one or more phases that are based on PbO principles, levels 2–4 integrate PbO tightly into the creation of software. We note that especially levels 3 and 4 lend themselves to a team approach, where various team members contribute alternative designs for functionally equivalent components.

Of course, it is possible to apply the PbO paradigm using existing methods and tools, particularly, since research on automated algorithm configuration and parameter tuning techniques has yielded a number of

very powerful techniques (these will be discussed in more details in Section 4). Nevertheless, we believe that, especially at the highest levels of PbO-based software development, there is substantial benefit to be gained from using dedicated tools. Figure 2 illustrates key aspects of the process of developing software using the dedicated PbO support described in more detail in Sections 3 and 4 of this report.

1. Developers write the source code for a program in a language that has been generically extended with constructs that explicitly declare alternative blocks of code and parameters that are to be exposed on the command line; for programming language $\langle \mathsf{L} \rangle$ we call the thus extended language PbO-$\langle \mathsf{L} \rangle$ (hence, PbO-C and PbO-Java would be the PbO-enhanced versions of C and Java, respectively). This enriched source specifies a design space of programs rather than a single program.

2. A tool we call the *PbO weaver* takes this PbO-$\langle \mathsf{L} \rangle$ specification and transforms it into a program written entirely in language $\langle \mathsf{L} \rangle$ that exposes the design choices specified in the original source as parameters;[4] it also produces a description of the respective design space (and various pieces of additional information about the parameters and design alternatives declared by the programmer).

3. A second tool, which we call the *PbO design optimiser*, produces from the parametric $\langle \mathsf{L} \rangle$-source a fully instantiated $\langle \mathsf{L} \rangle$-source, *i.e.*, a version of the program in which all design choices have been made in a way that results in desirable performance characteristics on benchmark inputs characteristic of a given use context. This version can then be deployed – in the case of compiled languages $\langle \mathsf{L} \rangle$, such as C or C++, after compilation.

The PbO-based approach to software development can provide substantially increased flexibility to software developers, providers and users. In particular, it makes it possible to quickly and automatically customise software for optimised performance in different use contexts; this optimisation can be carried out automatically, by the software developer, by the provider, or even by the user (we note that the latter can be done a the level of a parametric executable and does not require sources to be made available to the provider or user). Even more interestingly, PbO provides a generic way for creating software that periodically and automatically adapts itself for optimised performance as the use context changes over time. This process can be set up to take place entirely client-side or could be delivered as an internet service. In the latter case, input data from the actual application is collected client-side and transmitted to the service provider, where the PbO design optimiser is run to produce a new software configuration; this configuration is then transmitted back to the client and deployed there.

As mentioned in Section 1, another attractive feature of PbO lies in the ability to generate multiple solvers for a given problem, which are automatically combined into a portfolio (*i.e.*, run concurrently on the same input) or into a per-instance selector (*i.e.*, a mechanism that selects one of them to be run on a given problem instance based on characteristics of that instance) that is optimised for a given use context. These complex solvers are generated from the same design space specification – *i.e.*, from the same PbO-$\langle \mathsf{L} \rangle$ source – that gives rise to a single, optimised solver. Therefore, PbO offers generic and automated ways of taking advantage of the fact that large design spaces will typically contain programs that work well on different kinds of input data; it also offers a generic and automated way of generating parallel programs from inherently sequential sources. Of course, to construct portfolios and per-instance selectors, the PbO weaver needs to produce a suitably modified parametric source for the component solvers as well as for an *execution controller* that launches, monitors and terminates the component solvers as needed. Furthermore, the PbO optimiser needs to produce fully instantiated sources (or parameter settings) for all component solvers; when building a per-instance selector, it is also provided with a *feature extractor* that computes

[4] The term *weaver* has been chosen, because its intuitive meaning nicely reflects the type of program transformation affected here, but with a deliberate nod to the aspect-oriented programming paradigm (Kiczales et al., 1997), where it is used in a similar fashion.

from given input data the features based on which the component solver to be run is determined. This process will be described in more detail in Section 4.4.

3 Design space specification

To facilitate the specification of design spaces, we propose two basic mechanisms: one that provides dedicated support for exposing parameters and the other for specifying alternative blocks of code. Both mechanisms are realised through generic programming language extensions, as explained in the following, where the syntax we introduce here serves primarily illustrative purposes. Furthermore, without loss of generality, we assume that programs can be called with command-line arguments.

Of course, both mechanisms can be realised without dedicated programming language support. In particular, basically all programming languages provide support for parameters exposed to the user. However, we believe that a PbO-based software development process benefits from dedicated language constructs that make it easy to explicitly specify, in a way that is independent of the programming language used, the parts of a program representing design choices deliberately left open. The use of such dedicated language constructs also facilitates program transformations that hardwire into the source code certain design choices made during design optimisation – this process, which we call (partial) instantiation, will be discussed in more detail later in this section; furthermore, they provide a better basis for extending and enhancing widely used development platforms to support the use of PbO (we briefly comment on such extensions in Section 6).

3.1 Support for exposing parameters

We propose the following light-weight mechanism for declaring parameters that are to be exposed as command-line arguments:

```
##PARAM(<type> <name>[=<value>])
```

`<type>` indicates the type of the parameter and can take one of the following values:

```
bool = boolean
int = integer
float = real
char = character
str = string
```

where the two values shown in each row are synonyms. Furthermore, to support ranges and sets of allowable values, complex type specifiers of the following forms can be used:

int[a,b] = int[a..b]
int{i_1,i_2,$\cdots$,i_n}
float[a,b]
float{x_1,x_2,$\cdots$,x_n}
char{c_1,c_2,$\cdots$,c_n}
str{s_1,s_2,$\cdots$,s_n}

where a, b, i_k, x_k, c_k, s_k are valid literals of the respective types and bracket symbols, `,` and `..` are used

as shown.[5] `<name>` indicates the name of the parameter and needs to be a legal variable name in the programming language under consideration. `<value>` is an optional default value assigned to the parameter when the program is called; if a different value is assigned on the command line or configuration file, that value overrides the default. For boolean parameters, values can be specified as `false` and `true`, `f` and `t` (all case-sensitve), or as `0` and `1`. Character values are delimited by single and string values by double quotes. An error is generated when the value does not have the correct type.

Examples:

```
##PARAM(int numIterations)
```

– integer parameter `numIterations` gets exposed; there is no default value, and an error will be generated when calling the executable without assigning a value to this parameter.

```
##PARAM(int numIterations=10)
```

– integer parameter `numIterations` gets exposed, and its default value is set to 10.

```
##PARAM(int[0..1000] numIterations=10)
```

– integer parameter `numIterations` with possible values between 0 and 1000 gets exposed, and its default value is set to 10.

Parameter declarations can occur at any place in the source code, but we suggest that they should be located consistently either at the beginning of the source for the main procedure or object of the program, or just before their first occurrence (*i.e.*, the place where the parameter value is first accessed). We also suggest to use one separate line for each parameter declaration.

Throughout the program, parameter values are accessed using the following syntax:

```
##<name>
```

Example:

```
##numIterations
```

– the value of parameter `numIterations` (read-only)

Parameters can be read arbitrarily often and at arbitrary places throughout the program, wherever it would be legal for a literal representing the parameter value to occur. Access to parameters declared via this mechanism is read-only, *i.e.*, their values cannot be changed other than via a command-line argument or an entry in the configuration file.

Resolution by the PbO weaver

To process source code using this mechanism, the PbO weaver collects all parameter declarations from a software project. It checks for validity of parameter names, types and for agreements between types and

[5] Further extensions of this syntax – *e.g.*, to include hints on potential discretisation of real-valued parameters or preferred values, can be easily imagined; however, it might be preferable to specify this type of information, possibly along with more complex constraints on allowable values, in a separate place.

default values (where those have been specifies). It also ensures that all declarations use different parameter names.

In its *default mode*, the weaver generates source code for

- creating a distinct global variable for each parameter;
- parsing and processing command-line arguments and configuration files;
- generating error messages if invalid values are being assigned to parameters via the command line or configuration file, or if no value is assigned to a default-less parameter;
- accessing parameter values where needed.

This code is integrated into the given source(s), replacing or removing the original parameter references and declarations (`##<name>` and `##PARAM` statements) as needed.

In *instantiation mode*, the weaver replaces some or all exposed parameters with literal values. This is done by replacing the parameter wherever its value is read with the respective literal. Instantiated parameters are thus 'hard-wired' into the source code of the program under consideration and cannot be changed. Attempts to set them via the command line or configuration file produce the same error message as an attempt to set an undeclared parameters. The parameters to be instantiated and their respective values are given as an input to the weaver via its user interface (GUI or command line / specific instantiation file). Any uninstantiated parameters are treated as in default mode and thus exposed to be set at run time via the command line or configuration file. Instantiating parameters can be useful for producing marginally faster code, but more importantly, can be used to eliminate access to certain design parameters, which might be desirable for certain types of releases to end users (demo versions, restricted versions, *etc.*).

The weaver's output consists of the transformed source code files as well as a report listing all parameters along with their types and (where specified) default values; instantiated parameters are shown separately in the report. For each parameter, a list of occurrences (file names and line numbers) is also provided. Finally, the weaver produces a template for calling the parameterised executable as well as for a configuration file.

The realisation of weavers for languages that do not provide convenient support for processing command-line arguments can be simplified considerably by using packages that provide this type of functionality. For C and C++, several such packages exist, and of these we found `opt 3.19` (`http://public.lanl.gov/jt/Software`) to be particularly suitable.

3.2 Support for specifying alternative block of code

Alternative blocks of code are the primary mechanism for specifying design alternatives, where in this context, a block represents any fragment of code that constitutes a meaningful unit of replacement (in particular, successive lines of code constitute such units). To facilitate the specification of alternative blocks of code, we propose a mechanism based on so-called choice points and choices. A *choice* is a set of interchangeable fragments of code that represent design alternatives (where one alternative might be an empty fragment); those fragments are called *instances* of the choice. A *choice point* is a location in a program at which a choice is available. During execution, an instance of a choice is called *active* if it has been explicitly selected as such (via a command-line argument, configuration file or instantiation by the weaver).

Choices are declared using the following syntax:

```
##BEGIN CHOICE <name>[=<id>])
<code>
##END CHOICE <name>
```

`<name>` is the name of the choice and needs to be a valid variable name in the programming language under consideration; it also needs to be different from any parameter exposed via the mechanism described previously. `<id>` is an identifier given to the instance of the choice represented by code fragment `code`; this identifier can be an arbitrary sequence of letters and digits.

Examples:

```
##BEGIN CHOICE preProcessing
<block 1>
##END CHOICE preProcessing
```

– the code in `<block 1>` is marked up as a choice named `preProcessing`, and will only be executed if `preProcessing` is active at run time.

```
##BEGIN CHOICE preProcessing=standard
<block 1s>
##END CHOICE preProcessing

##BEGIN CHOICE preProcessing=enhanced
<block 1e>
##END CHOICE preProcessing
```

– a choice named `preProcessing` with two alternative instances, named `standard` and `enhanced`, which correspond to the code fragments `<block 1s>` and `<block 1e>`, respectively.

As a shorthand for declaring choices with multiple instances, we propose the syntax demonstrated in the following example:

```
##BEGIN CHOICE preProcessing=standard
<block 1s>
##CHOICE preProcessing=enhanced
<block 1e>
##END CHOICE preProcessing
```

– semantically equivalent shorthand form of the previous example.

The same choice name (and instances) can appear in multiple places within a program. At each of these choice points, the fragments of code specified for the respective choice declared at that point are available. Choices of this kind are called *distributed choices*.

Example:

```
##BEGIN CHOICE preProcessing
<block 1>
##END CHOICE preProcessing
...
##BEGIN CHOICE preProcessing
<block 2>
##END CHOICE preProcessing
```

– two occurrences (choice points) of the same choice; the code in `<block 1>` and in `<block 2>` (which may be different, of course), will only be executed if choice `preProcessing` is active at run time.

For distributed choices, the set of instances available at each choice point may differ, as in the following example:

```
##BEGIN CHOICE preProcessing=standard
<block 1s>
##CHOICE preProcessing=enhanced
<block 1e>
##END CHOICE preProcessing
...
##BEGIN CHOICE preProcessing=enhanced
<block 2e>
##END CHOICE preProcessing
```

– if instance `standard` of choice `preProcessing` is active at run time, at the first choice point, `<block 1s>` is executed, while at the second choice point, any empty choice is made (since no code was specified for choice `preProcessing=enhanced`).

Choices can be *nested*, as in the following example:

```
##BEGIN CHOICE stepType=1
<block A>
  ##BEGIN CHOICE stepPostOptimisation
  <block B>
  ##END CHOICE stepPostOptimisation
<block C>
##BEGIN CHOICE stepType
```

– `<block A>` and `<block C>` are executed if choice `stepType=1` is active; `<block B>` is executed (between `<block A>` and `<block C>`) if `stepType=1` *and* `stepPostOptimisation` are active at the same time.

Choices nested within instances of other choices are only available if all the enclosing instances are active (in the example above, choice `stepPostOptimisation` and therefore, `<block B>` is only available if choice `stepType=1` is active). Such nested choices are therefore said to be *conditional* with respect to the enclosing (higher-level) choices.

Resolution by the PbO weaver

To handle choices, the PbO weave, in its *default mode*, checks the validity of choice names and instance identifiers; it then introduces new, exposed parameters that allow for the control of choices at run time and transforms the choice declarations as required to facilitate this control.

In *instantiation mode*, the weaver replaces the respective choice declarations with the code fragment corresponding to a specific instance for that choice. Instantiated choices are thus 'hard-wired' into the source code of the program under consideration and cannot be controlled at run time, exactly like instantiated parameters. The choices to be instantiated and their respective instances are given as an input to the weaver via its user interface (GUI or command line / specific instantiation file). Any uninstantiated choices are treated as in default mode and thus exposed to be controlled at run time via the command line or configuration file. Instantiating choices can be useful for producing leaner and faster code, as well as for eliminating certain design choices, as might be desirable for certain types of releases to end users (demo versions, restricted versions, *etc.*) and even during development (*e.g.*, for the removal of poor or obsolete design choices, or for purposes of code clean-up). Similarly, in *elimination mode*, the weaver removes certain instances of choices, while the remaining instances are exposed as in default mode.

The weaver's output consists of the transformed source code files as well as a report listing all choices along with the identifiers of their instances; instantiated choices are shown separately in the report. For each choice, a list of choice points (file names and line numbers) is also provided, with an indication of the choice instances available at each choice points. Distributed and nested choices are listed separately; for the choice points of nested choices, all enclosing choice instances are also indicated. Finally, the weaver produces a template for calling the parameterised executable as well as for a configuration file, to be used by the PbO design optimiser (or other meta-algorithmic optimisation procedures).

The parts of the weaver that support choices and choice points make use of its mechanisms for handling exposed parameters, by creating a distinct, string-valued parameter for each choice. The additional program transformations required in this context are carried out in a rather straightforward way for languages such as C, C++, Java and Python, with the exception of choices involving code that cannot be encapsulated in conditional statements (such as declarations for objects or data structures). To support such choices in default mode, the weaver would have to resort to language-specific mechanisms; in many cases where such mechanisms are difficult to devise are where their use leads to unacceptable inefficiencies in the parametric program produced by the weaver, it may be preferable to use the weaver's instantiation mode for the respective choices within the design optimisation process.

4 Meta-algorithmic optimisation

Once a design space has been specified, meta-algorithmic optimisation procedures are used to automatically find a solver with desirable performance charactistics within in. In the simplest case, these procedures determine a single, fully instantiated solver whose performance (as measured according to a user-defined metric, *e.g.*, average run time) has been optimised for a given set of inputs. Since choices can be exposed as parameters (and are handled in this way by the PbO weaver described in the previous section), the problem solved by these meta-algorithmic optimisation procedures corresponds to the well-known *algorithm configuration problem* (sometimes also called *parameter tuning problem*) and the procedures themselves are referred to as *configurators*. In the algorithm configuration problem, given a target target algorithm A that can be configured via a set of exposed parameters, a set of benchmark problem instances I and a performance metric m, the objective is to find a parameter configuration of A that yields optimised performance on I, as measured by m (see, *e.g.*, Hutter et al., 2007b, 2009b; Hoos, 2011).

4.1 Standard stochastic and numerical optimisation methods

In principle, algorithm configuration can be seen as a stochastic optimisation problem (where the stochasticity stems from the performance variation observed over a set of input data or from randomisation of the computation performed on them) and solved using standard stochastic optimisation procedures (see, *e.g.*, Spall, 2003). However, we expect that in order to be reasonably effective, such procedures would need to be augmented with mechanisms for dealing with sets of inputs and capped runs. The issue of input sets is important, because evaluating many solver configurations on all inputs from a given set can incur a substantial (sometimes prohibitive) computational burden; this burden can and typically should be avoided, based on the observation that poor performance often manifests itself across a wide range of inputs. Furthermore, there are situations in which candidate configurations can be discarded without even completing runs that exceed a certain time bound, considering the performance measured for other configurations; such runs can be *capped*, *i.e.*, terminated when that time bound is reached or exceeded (Hutter et al., 2009b).

The same holds for numerical optimisation methods, which in principle could be used for solving configuration problems in which all parameters are real-valued and in which there are no parameter dependencies, such as conditional parameters. A secondary challenge stems from the fact that gradients (and higher-order derivates) for the function to be optimised are typically not available in the context of algorithm configuration. However, there are prominent and relatively recent gradient-free methods that could still be used in the restricted context of optimising real-valued, unconditional parameters, including the covariance matrix adaptation evolution strategy (CMA-ES) by Hansen and Ostermeier (2001) and the mesh adaptive direct search (MADS) algorithms by Audet and Orban (2006). Similarly, it is possible to use gradient-based numerical optimisation procedures – in particular, quasi-Newton methods, such as the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm (see, *e.g.*, Nocedal and Wright, 2006) – in conjunction with suitable methods for estimating or approximating gradient information.

However, the specification of alternative blocks of code, which is central to the way design spaces are constructed in PbO, necessarily leads to categorical (and hence discrete) parameters, and nested alternatives give rise to conditional parameters. Therefore, general methods for algorithm configuration used in the context of PbO need to support categorical and conditional parameters; they should also, for the reasons mentioned previously, support capping of runs.

4.2 Racing, model-free search and sequential model-based optimisation

There are three classes of methods that have been specifically designed for carrying out algorithm configuration tasks: *Racing procedures* iteratively evaluate target algorithm configurations on problem instances from a given set and use statistical hypothesis tests to eliminate candidate configurations that are significantly outperformed by other configurations; *model-free search procedures* use suitably adapted search techniques – in particular, stochastic local search methods, such as iterated local search – to explore potentially vast configuration spaces; and *sequential model-based optimisation (SMBO) methods* build a response surface model that relates parameter settings to performance, and use this model to iteratively identify promising settings. In the following, we will briefly discuss each of these classes of configuration methods, and point out specific procedures suitable for carrying out the configuration tasks that arise in the context of PbO; a more detailed survey can be found in a forthcoming book chapter by Hoos (2011) and in the primary sources referenced in the following.

Racing

Racing procedures have been originally introduced for solving model selection problems in machine learning (Maron and Moore, 1994). Transferred to the problem of of selecting a solver for a given problem from a set of candidates, where each candidate may correspond to a configuration of a parameterised algorithm, the key idea is to sequentially evaluate the candidates on a series of benchmark instances and eliminate solvers as soon as they have fallen too far behind the current incumbent, *i.e.*, the candidate with the overall best performance at a given stage of the race.

The F-Race algorithm by Birattari et al. (2002) uses a non-parametric statistical testing procedure (the rank-based Friedman test with appropriate *post hoc* tests) as the basis for deciding which configurations to eliminate in any given step of the race. The size of the configuration spaces to which this procedure can be applied effectively is limited by the fact that in the initial steps, all given configurations have to be evaluated. While F-Race can deal with categorical parameters, it requires discretisation (or sampling) of continuous parameters and does not support conditional parameters. The more recent Iterative F-Race (I/F-Race) procedure overcomes these limitations (Balaprakash et al., 2007; Birattari et al., 2010). I/F-Race is based on an iterative process, where in the first stage of each iteration, configurations are sampled from a probabilistic model M that is designed to represent 'promising' configurations, while in the second stage, a standard F-Race is performed on the resulting sample, and the configurations surviving this race are used to define or update the model M used in the following iteration (see Birattari et al., 2010 for details). Results in the literature demonstrate that I/F-Race can be used to effectively solve difficult algorithm configuration tasks with up to 12 parameters (Birattari et al., 2010), and there is some indication that it can handle substantially more complex situations with up to 26 parameters, many of which are categorical and conditional (Stützle, 2010).

The recent Random Online Adaptive Racing (ROAR) procedure does not use a statistical test to determine whether a candidate configuration is competitive with the current incumbent, but rather makes this decision using an aggressive heuristic mechanism (Hutter et al., 2011). Furthermore, in each step of the race, only two candidates are evaluated: the current incumbent and another configuration sampled uniformly at random from the given configuration space. ROAR supports categorical and continuous parameters, as well as conditional parameters. Despite its rather simplistic nature, it has been shown to work well for complex configuration tasks involving up to 76 parameters. However, because it currently does not support capping, ROAR is not competitive with the best model-free search techniques on configuration tasks that involve run-time minimisation and long run-times of the given target solver; it also fails to reach the performance of the best model-based search procedure, SMAC, on large configuration spaces (Hutter et al., 2011).

Model-free search

As of this writing, model-free search techniques, and notably the FocusedILS procedure by Hutter et al. (2009b), represent the state of the art in solving algorithm configuration problems of the kind arising in the context of PbO. Along with BasicILS, another member of the ParamILS family of algorithm configuration procedures (Hutter et al., 2007b, 2009b), it is the only method to support categorical and conditional parameters as well as capping. At the core of the ParamILS framework lies *Iterated Local Search (ILS)*, a well-known and versatile stochastic local search method that has been applied with great success to a wide range of difficult combinatorial problems (see, *e.g.*, Lourenço et al., 2002; Hoos and Stützle, 2004). ILS iteratively performs phases of simple local search, designed to rapidly reach or approach a locally optimal solution of a given problem instance, interspersed with so-called perturbation phases, whose purpose is to effectively escape from local optima. Starting from a local optimum x, in each iteration one perturbation

phase is performed, followed by a local search phase, with the aim of reaching (or approaching) a new local optimum x'. Then, a so-called acceptance criterion is used to decide whether to continue the search process from x' or whether to revert to the previous local optimum, x. Using this mechanism, ILS aims to solve a given problem instance by effectively exploring the space of its locally optimal solutions. ParamILS performs iterated local search in the space of configurations of a given parametric algorithm.

While BasicILS, the simplest variant of ParamILS, evaluates candidate configurations based on a fixed number of target algorithm runs, the more sophisticated FocusedILS procedure uses a heuristic mechanism to perform runs on a candidate configuration only as long as that configuration appears promising compared to the current incumbent, and therefore avoids wasting computational effort on configurations that can easily be recognised as performing poorly. (We note that the idea of focussing the computational effort spent on evaluating configurations on candidates that have already shown promising performance is exactly the same as that underlying the concept of racing.) The use of FocusedILS for configuring highly parametric solvers has lead to substantial improvements in the state of the art in solving SAT (Hutter et al., 2007a; KhudaBukhsh et al., 2009; Tompkins and Hoos, 2010), mixed integer programming (Hutter et al., 2010a), university timetabling (Chiarandini et al., 2008; Fawcett et al., 2009) and planning problems (Vallati et al., 2010).

One shortcoming of FocusedILS lies in the fact that, in its current version, it requires all parameters to be discretised. While in principle, finding reasonable discretisations (*i.e.*, ones whose use does not cause major losses in the performance of the configurations found by FocusedILS) could be difficult, in the applications considered so far, generic approaches, such as even or geometric subdivisions of a given interval, appear to give good results. Where this is not the case, multiple runs of the configuration procedure could be used to iteratively refine the domains of real-valued parameters. The same approach could be used to extend the ranges of allowable values for parameters in cases where parameter values in an optimised configuration lie at the boundary of their respective ranges. Nevertheless, the development of ParamILS variants that natively deal with real-valued parameters and support dynamic extensions of parameter domains remains an interesting direction for future work.

Ansótegui et al. (2009) recently developed a model-free search procedure for algorithm configuration based on a gender-based genetic algorithm. Their GGA configuration procedure supports categorical, ordinal and real-valued parameters; it also allows its user to express independencies between parameter effects by means of so-called variable trees – a concept that could be of particular interest in the context of PbO, where it might be possible to extract or infer such independencies (or approximate independencies) from the given design space specification. Although there is some evidence that GGA can solve some moderately difficult configuration problems more effectively than FocusedILS without capping (Ansótegui et al., 2009), it appears to be unable to reach the performance of FocusedILS version 2.3 with aggressive capping on the most challenging configurations problems (Hutter et al., 2010b, 2011). Unfortunately, GGA also offers less flexibility than FocusedILS in terms of the performance metric to be optimised, and is not publically available.

Sequential model-based optimisation

The key idea behind model-based search is to use the information gained from candidate solutions (here: parameter configurations) evaluated so far to build and maintain a model of the search space, based on which candidates are chosen to be evaluated in the future. Sequential model-based optimisation (SMBO) procedures for algorithm configuration utilise so-called *response surface models*, which capture directly the dependency of target algorithm performance on parameter settings, to determine promising configurations at any stage of an iterative model-based search procedure. This generic approach can be seen as a special

case of sequential analysis – a broader area within statistics that also comprises sequential hypothesis testing and so-called multi-armed bandits.

The setting considered in almost all work on sequential model-based optimisation procedures up to this day is known as the *black-box optimisation problem*: Given an unknown function f and a space of possible inputs X, find an input $x \in X$ that optimises f based on measurements of f on a series of inputs. The function to be optimised, f, may be deterministic or stochastic; in the latter case, measurements are subject to noise, and formally, the values of f are random variables. In the context of PbO, black-box optimisation methods, such as EGO (Jones et al., 1998), SKO (Huang et al., 2006), SPO (Bartz-Beielstein et al., 2005; Bartz-Beielstein, 2006; Bartz-Beielstein et al., 2008), SPO$^+$ (Hutter et al., 2009a) and TB-SPO (Hutter et al., 2010c), have the same shortcomings as standard numerical optimisation procedures: they do not support categorical and conditional parameters, and the do not provide effective mechanisms for dealing with sets of inputs and with capping of runs.

Two of these shortcomings have been overcome in recent work by (Hutter et al., 2011), who introduced a procedure dubbed *Sequential Model-based Algorithm Configuration (SMAC)* that can handle categorical parameters while explicitly exploiting the fact that performance is evaluated on a set of inputs. There is some evidence that SMAC can, at least on some challenging configuration benchmarks, reach and sometimes exceed the performance of FocusedILS (Hutter et al., 2011). We are convinced that, in the context of PbO, advanced SMBO methods such as SMAC hold great promise, particularly in cases where the parameter response of a given target algorithm is reasonably regular and performance evaluations are very costly.

4.3 Performance metrics

The performance metric used to assess candidate designs is an important component of the meta-algorithmic optimisation process. The definition of this metric typically has two components: One that measures the performance of a specific design on a single input, and another one that aggregates these per-input measures over the given set of inputs. Both are to some degree determined by the overall design objective, which in many cases is to minimise expected run-time or to maximise expected solution quality on a given class of inputs. It should be noted that when using such metrics, the optimised design might represent a tradeoff between strong performance on some inputs at the cost of weaker performance on others. Interestingly, the use of aggregate measures based on robust statistics, *e.g.*, median run-time, can lead to detrimental performance on large parts of a given input set, precisely because the objective to be optimised is oblivious to such effects. Therefore, in order to obtain designs that achieve robust performance, it is important to use a performance metric that is sensitive to poor performance on parts of the given input set.

We expect that in challenging applications of PbO it will be impossible or impractical to perform meta-algorithmic optimisation on inputs of the size that is ultimately of interest; in other cases, it may be desirable to obtain a single design that performs well on a range of input sizes. In our past work on algorithm configuration, we often observed that performance differences achieved for small, easier problem instances were amplified for larger, more difficult ones (see, *e.g.*, Hutter et al., 2007a, 2010a). We hypothesise that, for most of the heuristic mechanisms that contribute to good performance of solvers for $\mathcal{NP}$-hard problems on diverse set of benchmarks, improvements achieved on small instances tend to increase with instance size. We further note that certain aggregate performance measures, such as simple or penalised averages, are implicitly sensitive to scaling effects; for example, when minimising average run-time over a set of inputs of varying size, where run-time tends to increase (perhaps steeply) with input size, the performance measurements will tend to be dominated by the run-time on the largest instances. However, in many cases, it will be desirable to use more explicit mechanisms to achieve good scaling behaviour. Such mechanisms could be obtained

- by means of a generate-and-test approach, where many high-performance designs are produced, using the methods outlined above, and then tested (potentially using evaluation procedures too computationally expensive to be run frequently during the configuration process) for desirable scaling behaviour;
- by using carefully constructed sets of inputs over a range of sizes in conjunction with configuration procedures that take into account input size when selecting the inputs used for evaluating candidate configurations;
- by developing performance metrics that explicitly capture and quantify trends in scaling of per-input performance with input size, to be used during the configuration process; or
- by using performance models, such as the ones obtained in the context of sequential model-based optimisation, to capture and assess the dependence of performance on input size (we note that this last approach can only be expected to work well if the models are reasonably accurate in a global sense).

It is currently unclear which of these approaches (alone or in various combinations) will be most effective; because of the importance of finding designs whose performance scales well, research on this subject is of great interest in the context of the PbO paradigm.

Another challenge arises from target algorithm runs in which no solution was produced (in particular, timeouts encountered when minimising run-time). A standard approach for dealing with such censored runs is to use a performance metric that applies a fixed penalty to them; an example for such a performance metric is penalised average run-time (Hutter et al., 2009b). Refinements of this approach could include penalties that reflect the importance of the instance in question in a given use context.

In many cases, more than one performance measure is relevant in a given use context. For example, when applying algorithms that produce anytime solutions to optimisation problems, such as solvers based on stochastic local search, users often care about both, solution quality and run-time; a similar situation can arise in the context of designs involving repeated (or parallel) execution of algorithms with error, where the error is easy to detect and multiple independent runs are used to reduce the error probability (incomplete algorithms for $\mathcal{NP}$-hard problems like SAT can be seen in this light). In principle, these situations can be handled in the same way as other multi-objective optimisation problems. In some cases, the design objectives can be combined into a single aggregate performance measure (for example, in the form of a weighted sum), which can then be optimised using the meta-algorithmic optimisation techniques discussed previously. Sometimes, this approach is difficult or impossible – for example, because users have not decided how to trade off two performance objectives against each other; in such cases, there is a need for design techniques that maintain and iteratively improve sets of non-dominated solutions in an attempt to construct a set of Pareto-optimal designs. Such techniques allow users to explore tradeoffs between the given objectives permitted by feasible designs. While there is a large body of literature on multi-objective optimisation, we expect that, considering the particular features and requirements of meta-algorithmic optimisation tasks discussed earlier in this section, the development of specific multi-objective meta-algorithmic optimisation procedures is warranted.

We note that the multi-objective meta-algorithmic optimisation of a given target algorithm should not be confused with the meta-algorithmic optimisation of a multi-objective target algorithm. Considering the importance of multi-objective optimisation problems in various practical contexts, such as scheduling and resource allocation, the latter task is also of substantial interest. As demonstrated in recent work by López-Ibáñez and Stützle (2010), in which I/F-Race was used to automatically configure an ant colony optimisation procedure for the bi-objective TSP, this task can be accomplished using existing (single-objective) algorithm

configuration procedures in combination with standard performance metrics for multi-objective optimisers, such as hypervolume and the binary ϵ-indicator (Zitzler et al., 2003).

Finally, picking up a theme briefly discussed in previous work (Hoos, 2008), we note that it might be interesting to use meta-algorithmic optimisation procedures to simplify a design without incurring inacceptable performance losses (where what constitutes an acceptable loss is specified by the algorithm designer). In the context of PbO, such simplifications could be achieved via empty choices (*i.e.*, choices one of whose instances is empty, *cf.* Section 3) or by using methods that measure the complexity of choice instances (for example, in terms of description complexity or nesting of choices and parameters). Another interesting aspect of simplification in the context of PbO is the automated identification and elimination of choice instances that appear to contribute little or nothing to designs that are of interest in a given range of use contexts. This could be approached, for example, based on response surface models, as used in configuration procedures based on sequential model-based optimisation, or by means of ablation experiments. Future work in this area could play an important role in ensuring the design spaces and automatically optimised designs arising in the context of PbO do not become overly complicated.

4.4 Beyond configuration

One of the attractive features of PbO lies in the possibility to not only automatically produce optimised solver solver designs for a range of use contexts from a single design space specification, but also more complex types of solvers – in particular, algorithm portfolios (*i.e.*, sets of solver designs that run concurrently) and per-instance algorithm selectors (*i.e.*, sets of solvers from which one is selected at run time, based on characteristic of the specific problem instance to be solved).

Algorithm portfolios

In previous work, we have already discussed automated design techniques for algorithm portfolios (Hoos, 2008, p. 8), and to the best of our knowledge, no significant progress in terms of actual methods for automated portfolio design has been made since. In the context of PbO, the component solvers that make up the portfolio are obtained from the same design space; to produce an actual portfolio solver from those, the PbO design optimiser would furthermore generate code that coordinates their execution. In the simplest case, where the component solvers run independently in parallel on the same input data, this *execution manager* would start each component solver and process the results from these runs after the component solvers have terminated. When applied to a decision problem such as SAT, as soon as the first component solver found a solution (in the case of SAT, a model or a proof of unsatisfiability) for the given instance, the execution manager would terminate all remaining component solvers and return the result from the one run that completed. (Any other situation in which there is no need to continue the computation after any component has completed running would be handled analogously.) For optimisation problems, the execution manager would monitor the incumbent solutions produced by each component solver and pass on only those corresponding to the best candidate solution found by *any* component solver. [6] In situations where the component clusters are run on a cluster without a shared file system, the execution manager would also distribute the solver executables and input data.

The development of meta-algorithmic optimisation procedures that construct a suitable set of components solver from a given design space specification remains an interesting avenue for future work. We note

[6] In principle, this can be done in a generic way, as long as any solver design in the given design space uses a standardised method for recording changes in incumbent solutions, such as writing them to the standard output or to a database.

that in principle, this could be achieved by using automated configuration procedures on a configuration space defined over multiple independent copies of all solver parameters. Another approach consists of adding component solvers iteratively, where solvers are only configured immediately after they have been added (while component solvers added earlier remain fixed); since this corresponds to a greedy construction method that might not be able to find globally optimal solutions, it could be preferable to interleave this process with phases in which the current configurations of all component solvers can be modified. Finally, we note that the availability of reasonably accurate response surface models in this context could lead to substantial savings in computational effort, by means of replacing costly target solver runs by much cheaper model evaluations.

Per-instance algorithm selectors

The idea of automatically constructing a per-instance algorithm selector from a single parametric design has recently been explored by Xu et al. (2010) and, independently, by Kadioglu et al. (2010). In both cases, a given feature extractor is used to compute a vector of features from the instance to be solved. The method by Kadioglu et al. (2010), dubbed ISAC, uses a combination of clustering based on the feature values and automatic algorithm configuration to produce an algorithm selector. The Hydra procedure by Xu et al. (2010), on the other hand, iteratively adds solvers to the set available to the per-instance selector; in each iteration, an additional solver is configured automatically to maximally improve the performance obtained when building a selector using the thus extended set of solvers. In principle, both, Hydra and ISAC, can make use of arbitrary feature extractors and algorithm configuration procedures. Whereas ISAC produces a single selector, based on a number of components automatically determined by the G-means algorithm of Hamerly and Elkan (2004), Hydra builds a series of selectors – the longer it runs, the more component solvers are available for selection, and the better the expected performance of the resulting selector. Furthermore, Hydra can make use of arbitrary selector builders; Xu et al. (2010) used a procedure based on the regression-based performance predictor underlying the well-known SATzilla approach (Nudelman et al., 2004; Xu et al., 2008), but many alternatives are possible. As of this writing, a direct performance comparison between Hydra and ISAC has not been performed and would be very difficult to carry out, based on the fact that ISAC is not publically available, and some of the data from the study by Kadioglu et al. (2010) appears to have been irretrievably lost (Sellmann, 2010). While there are theoretical reasons to believe that at least under some circumstances, ISAC's performance could be severely compromised by misleading information obtained from the underlying clustering mechanism, it is not clear to which extent such situations may arise in practice.

Procedures for building per-instance algorithm selectors fit very naturally into the context of PbO. To construct per-instance selectors from a given design space specification, the PbO design optimiser would use a procedure such as Hydra (or ISAC) to obtain a suitable set of component solvers. It would furthermore produce an execution manager that first calls the feature extractor also provided by the user (this is specific to the problem to be solved, but does not depend on the target solver design), and then select the component algorithm to be run based on the resulting instance features. More complex selection mechanisms, such as the multi-phase procedure underlying the latest version of SATzilla (Xu et al., 2009) can, in principle, be used in the context of PbO in a similar fashion.

Adaptive mechanisms and logging of run-time information

Rather than making certain choices before running a pogram on given input data, one might want to make those decisions at run time, depending on information collected during earlier phases of the run. This is

the idea underlying *reactive search procedures*, *on-line algorithm control methods* and *adaptive operator selection techniques* (see, *e.g.*, Battiti et al., 2008; Carchrae and Beck, 2005; Da Costa et al., 2008); it also provides the motivation for so-called *hyper-heuristics* (see, *e.g.*, Cowling et al., 2002; Burke et al., 2003), *generalised local search machines* (see, *e.g.*, Hoos and Stützle, 2004; Hoos, 2009) and *dynamic algorithm portfolio approaches* (see, *e.g.*, Gagliolo and Schmidhuber, 2006).

Since these mechanisms typically have parameters or heuristic components for which various instantiations can be easily imagined, PbO provides a direct way of defining design spaces that strongly involve such adaptive mechanisms. In fact, because the way in which such mechanisms control the behaviour of a target solver is often even less intuitive than is the case with other design choices, we believe that automatic design optimisation, as performed in the context of PbO, will play an even more crucial role in realising the performance potential inherent in adaptive solver designs.

Another way in which PbO could facilitate the use of adaptive mechanisms involves their automatic introduction in the design optimisation stage. This would require either splicing a generic adaptive mechanism (which could involve parameters and design choices that would be added to those of the user-specified design space and subject to the same meta-algorithmic optimisation process) into the code produced by the PbO weaving routines described in Section 3, or the generation of an execution controller that controls certain parameter settings or choices at execution time. We believe that the second mechanism is preferable, since it maintains a clearer conceptual separation between solvers and automatically added adaptive mechanisms. In either case, information collected during the actual run is required as input for the adaptive mechanism. In the case of solvers for optimisation problems, up-to-date information about incumbent solutions are often provided in a generic way (*e.g.*, via writing them to standard output, to a file or to a database), and a variety of potentially very useful features could be computed from the time series of incumbent quality thus obtained. However, we believe that in many cases, it would be useful to have access to additional information about the state of the computation.

One way of supporting developers in exposing such information to be collected during the run of a program is via an additional mechanism in the generic PbO programming language extension. Specifically, analogous to the syntax introduced in Section 3, we propose the construct

```
##LOG(<type> <name>=<value>)
```

where `<value>` is an expression whose value is logged (*i.e.*, collected) at this point of the computation, `<name>` is a user-defined name used to identify the information being collected, and `<type>` indicates the data type of the expression being evaluated. (Altough types could in most cases be inferred during static program analysis, we see some benefits in forcing the programmer to provide them explicitly; for example, explicit type information simplifies the design of the PbO weaver and can provide the basis for additional type checking.) Specifying information to be logged via this mechanism, rather than using target language constructs, enables transparent support for different logging mechanisms: Depending on settings of the PbO weaver, time-stamped values logged in this way could be written to one or more files (or to the standard output) or to a database; they could also be sent directly to an execution controller, using other mechanisms, such as remote procedure calls or network sockets.

5 Use of computational resources

It might seem that, due to its use of compute-intensive meta-algorithmic optimisation procedures, programming by optimisation always requires large amounts of computational resources. However, while meta-

algorithmic optimisation in a large design space can certainly become computationally expensive, it often yields good results at relatively modest computational cost (see, *e.g.*, Hutter et al., 2007a, 2010a; Vallati et al., 2010).

At the same time, the programming by optimisation paradigm can effectively utilise compute cycles, and in particular, parallel computation, for finding better solutions to given computational problems in two ways: At design time, for finding performance-optimised solvers in potentially very large design spaces, and at run time, by means of automatically constructed parallel solvers.

Meta-algorithmic optimisation techniques, be they based on stochastic local search (such as ParamILS), sequential model-based optimisation (such as SMAC) or on racing methods (such as I/F-Race), are all anytime procedures, in that they tend to produce increasingly better results as they are run for longer and longer times. When searching very large and complex design spaces, it is unlikely that any meta-algorithmic optimisation procedure can find optimal solutions within any reasonable amount of compute time. Therefore, at least when using procedures that do not prematurely stagnate, we can expect that longer running times will lead to better results in the form of higher-performing target algorithms.

But nowadays, increases in computational power are mostly achieved in the form of parallel processing capabilities (as in multi-core CPUs, symmetrical multi-processor architectures and large compute clusters) rather than by increases in processor speed. It is therefore particularly interesting to note that all types of meta-algorithmic optimisation methods mentioned above can easily be adapted to make effective use of parallel computation. This can be done in two ways: by evaluating multiple designs in parallel or by concurrently evaluating one design on several inputs. The former type of parallelisation can be achieved easily by performing multiple independent runs of the meta-algorithmic optimisation procedure, particularly if it is based on a stochastic local search algorithm, as in the case of ParamILS. Similarly, the initial design phase in procedures such as ParamILS, during which multiple designs are evaluated, can be parallelised in a straight-forward manner, and the same holds for neighbourhood evaluations in ParamILS as well as for the intensification procedure used in SMAC. The concurrent evaluation of one design on several inputs appears somewhat less attractive, since it is prone to wasting compute cycles in situations where the design being evaluated turns out to be poor. (Note that such situations can often be detected based on evaluations on few inputs.)

Besides these design-time uses, parallel computating resources can also be exploited by the algorithms being designed using the PbO paradigm. Firstly, PbO can be applied to design spaces of parallel algorithms and optimise performance metrics that capture performance in a multi-core or multi-processor computing environment. The use of PbO in this way appears justified and attractive, because in the construction of parallel algorithms at least as many design choices are encountered as in the design of sequential solvers.

Secondly, as explained in Section 4.4, the PbO paradigm makes it possible to automatically construct parallel solvers for a given problem, in the form of portfolio solvers, from design spaces of sequential solvers. Analogous to financial portfolios, such portfolio solvers could be optimised to achieve maximal average performance over a given set of inputs, while keeping the performance variation bounded; alternatively, they could be optimised for minimal performance variation with bounded average performance.

Finally, PbO can leverage computational resources to deal with situations where important features of the problem instances to be solved may change over time. In such situations, it may be desirable to automaticelly adapt the solver design, so that at any time, a solver is used that is well-suited for the problem instances currently to be solved (this idea is closely related to the concept of *lifelong learning* – see, *e.g.*, Thrun, 1996). PbO can be used to achieve this goal by automatically generating new solvers according to the problem instances deemed to be representative of the current use context. This process takes place after the initial design phase, in the actual application context, and does not involve human designers; it can be

carried out by a deployed system that involves a highly parametric solver, a meta-algorithmic optimisation procedure and a mechanism for deciding which of the problem instances encountered during the lifetime of the system are to be used when assessing the performance of candidate solver designs.

In the simplest form, this process could make use of spare compute cycles (on sequential or parallel computing resources) for simply running the meta-algorithmic optimisation procedure occasionally or continuously. In this context, assuming that instance characteristics do not change too abruptly, it would make sense to start the meta-algorithmic optimisation process from the current solver configuration. Alternatively, one could evaluate multiple starting points (as currently done, *e.g.*, in ParamILS), where each solver constructed up to this point would provide one initial candidate design.

A more advanced variant would use the set of solvers available at any time as the basis for assembling a per-instance algorithm selector. In this context, an extractor for instance features would also be part of the deployed system. This feature extractor would be used used to analyse each instance to be solved, and the resulting feature vector would provide the basis for selecting the solver to be run on that instance (see, *e.g.*, Xu et al., 2009).

6 Conclusions

We believe that the PbO paradigm discussed in this report offers numerous benefits to software developers and users; chiefly amongst these are better performance of the programs created in this way, easier and more effective adaptation to different (and changing) use contexts, as well as better use of human capabilities and skills throughout the development process.

These advantages have already become apparent in work that clearly exhibits key elements of the approach advocated by PbO (see, *e.g.*, Hutter et al., 2007a; KhudaBukhsh et al., 2009; Fawcett et al., 2009; Hutter et al., 2010a; Tompkins and Hoos, 2010; Vallati et al., 2010, as well as Whaley et al., 2001; Cooper et al., 1999; Pan and Eigenmann, 2006); however, we believe that even greater benefits can be realised through higher levels of PbO-based software development (as outlined in Section 1) and dedicated support in the form of the language extensions and tools described in this report. A first version of a PbO weaver for PbO-C (the PbO extension of the C programming language) has been implemented by the author and is avaliable to others upon request. PbO design optimisation can be achieved by means of readily available automated algorithm configuration procedures, such as ParamILS (Hutter et al., 2007b, 2009b); considering the active research efforts currently underway in this area, particularly in the context of racing and sequential model-based optimisation procedures (see, *e.g.*, Birattari et al., 2010; Hutter et al., 2011), we expect even better performing procedures to be available soon. Similarly, meta-algorithmic procedures that can effectively produce per-instance algorithm selectors from a single, highly parametric design are now available (see, *e.g.*, Xu et al., 2010), and we expect substantial further progress in this area in the near future. Research on automated procedures for producing parallel portfolios from a given design space specification is currently underway and expected to produce useful results very soon. We plan to integrate these (and possibly other) meta-algorithmic optimisation procedures into a single PbO design optimiser that facilitates their use in the context of PbO-based software development.

Our *High Performance Algorithm Laboratory (HAL)* environment (Nell et al., 2011), which has been designed to support the computer-aided design and the empirical analysis of high-performance algorithms by means of a collection of ready-to-use, state-of-the-art analysis and design procedures, provides an ideal platform for realising and operating an integrated PbO design optimiser; we therefore plan to strongly integrate PbO support into HAL. Furthermore, we believe that extensions and enhancements of widely used

development platforms – in particular, the Eclipse IDE (`www.eclipse.org`) – will provide useful support for PbO-based software development. Besides syntax highlighting and folding for PbO constructs, we envision tools that support developers in tracking and navigating parameters, choices (in particular, distributed choices) and logs declared in PbO-sources, and in using PbO weavers and optimisers in their various modes.

We anticipate two major concerns being raised regarding PbO-based software development. The first of these pertains to correctness and debugging. At first glance, it might seem that when dealing with the large, combinatorial spaces of programs that are key to the PbO paradigm, the occurrence of bugs would be amplified to the point where testing and debugging becomes a major burden, if not completely infeasible. However, this concern is mitigated by the fact that design alternatives for individual mechanisms and components can be tested individually (which effectively reduces the combinatorial set of programs to be checked to a set that grows linearly with the number of choices and choice instances). While there is a potential for error conditions that only arise in particular instantiations of multiple design choices, we believe that these are quite rare and can be mostly avoided by following sound practices regarding proper encapsulation of program code and data structures. Furthermore, as recently observed by Hutter et al. (2010c), the use of design optimisation tools such as automated algorithm configurators makes it possible to find previously unknown bugs in widely used solvers developed using traditional methods.

A second concern follows from the observation that optimisation of software for a narrowly defined use context can lead to brittle performance. This issue arises prominently in the area of machine learning, which offers a number of techniques to address it. We believe that a combination of judicious practices for constructing the input data sets used in the optimisation process, appropriately defined optimisation objectives and suitable methods for assessing the performance of candidate designs can be used to effectively avoid brittle performance and poor generalisation beyond narrowly defined classes of input data.

On the other hand, we expect that the use of PbO will not only result in the previously mentioned practical advantages, but also facilitate scientific insights into the efficacy of algorithms and their components, as well as into the empirical complexity of computational problems. For example, to measure the extent to which a particular instance of a design choice contributes to overall performance, one would simply remove that instance (or instruct the weaver to ignore it) and compare the performance obtained in one or more use contexts when optimising within this reduced design space with that obtained from the original design space. Further analysis of the differences between the two designs thus obtained can shed insight into the degree to which other design choices can compensate for the effects of eliminating that choice instance from the design space. Comparisons between designs that have been optimised for different use contexts can reveal interactions between characteristics of the inputs of a program and the mechanisms that should be used to achieve good performance on those inputs. We note that for all of these types of empirical investigations, it might be interesting to consider design spaces not necessarily constructed to achieve the best possible performance, but rather to study certain mechanisms or data structures. Finally, when analysing the empirical complexity of computational problems, it is often of crucial importance to solve instances of those problems as efficiently as possible. We believe that the use of PbO strongly facilitates the construction of the solvers required in this context. Furthermore, because PbO enables empirical investigations into the interaction between problem instance characteristics and the efficacy of certain solver components, it can help to provide insights into what renders certain problems hard to solve.

Efforts that are concepually related to the ideas underlying PbO can be traced back more than 30 years, to the work of Rice (1976), but only much more recently have the powerful optimisation and machine learning techniques as well as the computational environments required to effectively carry out the automated design optimisation at the heart of the PbO approach become readily available. While much work remains to be done to demonstrate the utility of the PbO paradigm described in this report, we believe that it has the

potential to change the way in which we build and use software. We expect the approach to be particularly impactful in the context of solving $\mathcal{NP}$-hard problems, where general insights into empirically effective solution methods are very limited; at the same time, we are convinced that it will prove to be useful on a much broader scale.

PbO can be seen as a logical extension of existing work on parameter tuning, automated algorithm configuration and automated algorithm selection, as well as of the more general approach of computer-aided algorithm design (Hoos, 2008); it is also complementary to work on algorithm portfolios and self-adaptation, which can benefit from the use of PbO and be leveraged in PbO-based software development. Further work on PbO will not only integrate techniques and considerations from a wide range of reseach areas, including artficial intelligence, empirical algorithmics, machine learning, numerical optimisation, programming languages, software engineering and statistics, but at the same time pose interesting challenges for those areas.

In closing, we note that the key idea underlying PbO appears to be in stark contrast with established practices, which reflect a general tendency to avoid parameters, to eliminate and restrict degrees of freedom. This tendency appears to be well aligned with the fact that the fundamental notion of algorithm leaves no room for ambiguity. PbO, in some sense, turns this on its head, by asking algorithm designers and software developers to actively seek design alternatives and encouraging highly parametric designs – if it turns out to be as widely successful as we believe it will be, programming by optimisation may therefore even change the way we think about computation.

Acknowledgements: Some of the ideas discussed in this document have their roots in joint work and discussions with Frank Hutter, Chris Fawcett, James Styles, Kevin Leyton-Brown and Catherine Yelick. The author gratefully acknowlegdes helpful comments by Chris Fawcett and James Styles on earlier versions of this document.

References

Ansótegui, C., Sellmann, M., and Tierney, K. (2009). A gender-based genetic algorithm for the automatic configuration of algorithms. In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming (CP 2009)*, pages 142–157. Springer Verlag.

Audet, C. and Orban, D. (2006). Finding optimal algorithmic parameters using the mesh adaptive direct search algorithm. *SIAM Journal on Optimization*, 17(3):642–664.

Babić, D. and Hu, A. J. (2007). Structural Abstraction of Software Verification Conditions. In *Computer Aided Verification: 19th International Conference, CAV 2007*, volume 4590 of *LNCS*, pages 366–378. Springer Verlag.

Balaprakash, P., Birattari, M., and Stützle, T. (2007). Improvement strategies for the F-Race algorithm: Sampling design and iterative refinement. In *Proc. 4th Intl. Workshop on Hybrid Metaheuristics (HM 2007), LNCS 4771*, pages 108–122. Springer-Verlag.

Bartz-Beielstein, T. (2006). *Experimental Research in Evolutionary Computation: The New Experimentalism*. Natural Computing Series. Springer Verlag, Berlin, Germany.

Bartz-Beielstein, T., Lasarczyk, C., and Preuß, M. (2005). Sequential parameter optimization. In McKay, B. et al., editors, *Proceedings 2005 Congress on Evolutionary Computation (CEC'05), Edinburgh, Scotland*, volume 1, pages 773–780. IEEE Press.

Bartz-Beielstein, T., Lasarczyk, C., and Preuss, M. (2008). Sequential parameter optimization toolbox. Manual version 0.5, September 2008, available at http://www.gm.fh-koeln.de/imperia/md/content/personen/lehrende/bartz_beielstein_thomas/ spotdoc.pdf.

Battiti, R., Brunato, M., and Mascia, F. (2008). *Reactive Search and Intelligent Optimization*. Operations Research/Computer Science Interfaces. Springer Verlag.

Birattari, M., Stützle, T., Paquete, L., and Varrentrapp, K. (2002). A racing algorithm for configuring metaheuristics. In *GECCO '02: Proc. of the Genetic and Evolutionary Computation Conference*, pages 11–18.

Birattari, M., Yuan, Z., Balaprakash, P., and Stützle, T. (2010). F-Race and Iterated F-Race: An overview. In *Experimental Methods for the Analysis of Optimization Algorithms*, pages 311–336. Springer-Verlag.

Burke, E., Kendall, G., Newall, J., Hart, E., Ross, P., and Schulenburg, S. (2003). Hyper-heuristics: an emerging direction in modern search technology.

Carchrae, T. and Beck, J. (2005). Applying machine learning to low knowledge control of optimization algorithms. *Computational Intelligence*, 21(4):373–387.

Chiarandini, M., Fawcett, C., and Hoos, H. (2008). A modular multiphase heuristic solver for post enrollment course timetabling (extended abstract). In *Proceedings of the 7th International Conference on the Practice and Theory of Automated Timetabling (PATAT 2008)*. 6 pages.

Cooper, K. D., Schielke, P. J., and Subramanian, D. (1999). Optimizing for reduced code space using genetic algorithms. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems*, LCTES '99, pages 1–9.

Cowling, P., Kendall, G., and Soubeiga, E. (2002). Hyperheuristics: A tool for rapid prototyping in scheduling and optimisation. In *Applications of Evolutionary Computing*, volume 2279 of *Lecture Notes in Computer Science*, pages 1–10. Springer Verlag.

Da Costa, L., Fialho, Á., Schoenauer, M., and Sebag, M. (2008). Adaptive Operator Selection with Dynamic Multi-Armed Bandits. In *Proc. 10th Annual Conf. on Genetic and Evolutionary Computation (GECCO'08)*, pages 913–920.

Fawcett, C., Hoos, H., and Chiarandini, M. (2009). An automatically configured modular algorithm for post enrollment course timetabling. Technical Report TR-2009-15, University of British Columbia, Department of Computer Science.

Gagliolo, M. and Schmidhuber, J. (2006). Dynamic algorithm portfolios. In *Proc. 9th International Symposium on Artificial Intelligence and Mathematics (AI-MATH-06)*.

Gomes, C. P. and Selman, B. (2001). Algorithm portfolios. *Artificial Intelligence*, 126(1-2):43–62.

Guerri, A. and Milano, M. (2004). Learning techniques for automatic algorithm portfolio selection. In *Proc. 16th European Conf. on Artificial Intelligence (ECAI 2004)*, pages 475–479.

Hamerly, G. and Elkan, C. (2004). Learning the k in k-means. In *Advances in Neural Information Processing Systems 16 (Proc. NIPS 2003)*, Cambridge, MA. MIT Press.

Hansen, N. and Ostermeier, A. (2001). Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9(2):159–195.

Hoos, H. (2008). Computer-aided design of high-performance algorithms. Technical Report TR-2008-16, University of British Columbia, Department of Computer Science.

Hoos, H. (2009). Computer-aided algorithm design using generalised local search machines and related design patterns. Technical Report TR-2009-26, University of British Columbia, Department of Computer Science.

Hoos, H. H. (2011). Automated algorithm configuration and parameter tuning. In Hamadi, Y. and Saubion, F., editors, *Autonomous Search*. Springer-Verlag. To appear.

Hoos, H. H. and Stützle, T. (2004). *Stochastic Local Search—Foundations and Applications*. Morgan Kaufmann Publishers, USA.

Huang, D., Allen, T. T., Notz, W. I., and Zeng, N. (2006). Global optimization of stochastic black-box systems via sequential kriging meta-models. *Journal of Global Optimization*, 34(3):441–466.

Huberman, B., Lukose, R., and Hogg, T. (1997). An economics approach to hard computational problems. *Science*, 265:51–54.

Hutter, F., Babić, D., Hoos, H. H., and Hu, A. J. (2007a). Boosting verification by automatic tuning of decision procedures. In *Proc. Formal Methods in Computer-Aided Design (FMCAD'07)*, pages 27–34. IEEE Computer Society Press.

Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2010a). Automated configuration of mixed integer programming solvers. In *Proc. 7th Intl. Conference on the Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2010)*, pages 186–202.

Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2010b). Sequential model-based optimization for general algorithm configuration (extended version). Technical Report TR-2010-10, University of British Columbia, Department of Computer Science.

Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2011). Extending sequential model-based optimization to general algorithm configuration. To appear in: *Proc. 5th Intl. Conference on Learning and Intelligent Optimization (LION 5)*.

Hutter, F., Hoos, H. H., Leyton-Brown, K., and Murphy, K. P. (2009a). An experimental investigation of model-based parameter optimisation: SPO and beyond. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation (GECCO'09)*, pages 271–278. ACM.

Hutter, F., Hoos, H. H., Leyton-Brown, K., and Murphy, K. P. (2010c). Time-bounded sequential parameter optimization. In *Proceedings of the 4th International Conference on Learning and Intelligent Optimization (LION 4)*, pages 281–298. Springer-Verlag.

Hutter, F., Hoos, H. H., Leyton-Brown, K., and Stützle, T. (2009b). ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306.

Hutter, F., Hoos, H. H., and Stützle, T. (2007b). Automatic algorithm configuration based on local search. In *Proceedings of the 22nd National Conference on Artificial Intelligence (AAAI-07)*, pages 1152–1157.

Jones, D. R., Schonlau, M., and Welch, W. J. (1998). Efficient global optimization of expensive black box functions. *Journal of Global Optimization*, 13:455–492.

Kadioglu, S., Malitsky, Y., Sellmann, M., and Tierney, K. (2010). Isac – instance-specific algorithm configuration. In *Proc. 19th European Conference on Artificial Intelligence (ECAI 2010)*, pages 751–756.

KhudaBukhsh, A., Xu, L., Hoos, H., and Leyton-Brown, K. (2009). SATenstein: Automatically building local search sat solvers from components. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI-09)*, pages 517–524.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In *Proc. 11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag.

Leyton-Brown, K., Nudelman, E., Andrew, G., McFadden, J., and Shoham, Y. (2003). A portfolio approach to algorithm selection. In *Principles and Practice of Constraint Programming (CP 2003), LNCS 2833*, pages 899–903.

López-Ibáñez, M. and Stützle, T. (2010). Automatic configuration of multi-objective aco algorithms. In *Proceedings of the 7th International Conference on Swarm Intelligence*, ANTS'10, pages 95–106. Springer-Verlag.

Lourenço, H. R., Martin, O., and Stützle, T. (2002). Iterated local search. In Glover, F. and Kochenberger, G., editors, *Handbook of Metaheuristics*, pages 321–353. Kluwer Academic Publishers, Norwell, MA, USA.

Maron, O. and Moore, A. W. (1994). Hoeffding races: Accelerating model selection search for classification and function approximation. In *In Advances in neural information processing systems 6*, pages 59–66. Morgan Kaufmann.

Nell, C. W., Fawcett, C., Hoos, H. H., and Leyton-Brown, K. (2011). HAL: A framework for the automated design and analysis of high-performance algorithms. To appear in: *Proc. 5th Intl. Conference on Learning and Intelligent Optimization (LION 5)*.

Nocedal, J. and Wright, S. J. (2006). *Numerical Optimization*. Springer-Verlag, 2nd edition.

Nudelman, E., Leyton-Brown, K., Devkar, A., Shoham, Y., and Hoos, H. H. (2004). Understanding random SAT: Beyond the clauses-to-variables ratio. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP 2004)*, pages 438–452.

Pan, Z. and Eigenmann, R. (2006). Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 319–332, Washington, DC, USA. IEEE Computer Society.

Rice, J. R. (1976). The algorithm selection problem. *Advances in Computers*, 15:65–118.

Sellmann, M. (2010). Personal communication. December 2010.

Spall, J. (2003). *Introduction to Stochastic Search and Optimization*. John Wiley & Sons, Inc., New York, NY, USA.

Stützle, T. (2010). Personal communication. December 2010.

Thrun, S. (1996). *Explanation-Based Neural Network Learning: A Lifelong Learning Approach*. Kluwer Academic Publishers, Boston, MA.

Tompkins, D. A. and Hoos, H. H. (2010). Dynamic Scoring Functions with Variable Expressions: New SLS Methods for Solving SAT. In *Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing (SAT 2010)*, pages 278–292. Springer-Verlag.

Vallati, M., Fawcett, C., Gerevini, A., Hoos, H. H., and Saetti, A. (2010). A framework for automatic generation of efficient domain-specific planners from generic parametrized planners. Submitted for publication.

Whaley, R. C., Petitet, A., and Dongarra, J. J. (2001). Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35.

Xu, L., Hoos, H. H., and Leyton-Brown, K. (2010). Hydra: Automatically configuring algorithms for portfolio-based selection. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI-10)*, pages 210–216.

Xu, L., Hutter, F., Hoos, H., and Leyton-Brown, K. (2009). SATzilla2009: An Automatic Algorithm Portfolio for SAT. Solver description, SAT Competition 2009.

Xu, L., Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2008). SATzilla: portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606.

Zitzler, E., Thiele, L., Laumanns, M., Fonseca, C., and da Fonseca, V. (2003). Performance assessment of multiobjective optimizers: an analysis and review. *IEEE Transactions on Evolutionary Computation*, 7(2):117–132.