Appendix:
Generic PbO programming language extension

As explained in the main text, we propose three fundamental mechanisms to be covered by a generic PbO programming language extension; these provide light-weight support for exposing parameters, for specifying alternative blocks of code and for flexibly exposing information collected at run-time. While the precise syntax and realization of the mechanisms may undergo changes in the future, we believe that what is described in the following forms a solid basis for design space specifications in the context of PbO-based software development. After outlining the three mechanisms that form the generic language extension and illustrating their use by means of brief examples, we explain the design objectives underlying our proposal and discuss potential drawbacks.

A.1 Support for exposing parameters

We propose the following light-weight construct for declaring parameters that are to be exposed as command-line arguments (where the part in square brackets is optional):

```
##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 literals 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, ···,i_n}
float[a,b]
float{x_1,x_2, ···,x_n}
char{c_1,c_2, ···,c_n}
str{s_1,s_2, ···,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.[1] *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`,

[1]Further extensions of this syntax – *e.g.*, to include hints on potential discretization of real-valued parameters or preferred values – can be easily imagined; however, we believe that it would be preferable to specify this type of information, possibly along with more complex constraints on allowable values, in a separate input file to the PbO weaver.

or as `0` and `1`. Character and string values are delimited by single or double quotes. An error is generated when the specified default 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, function, object or class 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*

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.

Example:

```
##numIterations
```

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

Resolution by the PbO weaver

To process source code using these constructs for declaring and accessing parameters, the PbO weaver collects all parameter declarations from a software project. It checks for validity of parameter names and types, as well as for agreements between types and default values (where those have been specifies). It also ensures that all declarations use different parameter names.

In its *parametric 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 parameter. 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 parametric mode and thus exposed to be set at run time via the command line or configuration file.

A.2 Support for specifying alternative blocks of code

The mechanism used for the specification of design alternatives in the form of alternative blocks of code is based on so-called choices and choice points. A *choice* is a set of interchangeable blocks of code that represent design alternatives (where one alternative might be an empty block); those blocks 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 (where the part in square brackets is optional):

```
##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 (optional) identifier given to the instance of the choice represented by code fragment *code* (which can be empty); this identifier can be an arbitrary sequence of letters and digits.

Examples:

```
##BEGIN CHOICE preProcessing
block1
##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`
$block_s$
`##END CHOICE preProcessing`

`##BEGIN CHOICE preProcessing=enhanced`
$block_e$
`##END CHOICE preProcessing`

– a choice named `preProcessing` with two alternative instances, named `standard` and `enhanced`, which correspond to the code fragments $block_s$ and $block_e$, respectively.

As a shorthand for declaring choices with multiple instances, the syntax demonstrated in the following example can be used:

`##BEGIN CHOICE preProcessing=standard`
$block_s$
`##CHOICE preProcessing=enhanced`
$block_e$
`##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`

$\ldots$

`##BEGIN CHOICE preProcessing`
$block_2$
`##END CHOICE preProcessing`

– two occurrences (choice points) of the same choice; the code fragments $block_1$ and $block_2$ (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`
$block_s$
`##CHOICE preProcessing=enhanced`
$block_e$
`##END CHOICE preProcessing`

$\ldots$

```
##BEGIN CHOICE preProcessing=enhanced
```
$block_f$
```
##END CHOICE preProcessing
```

– if instance `standard` of choice `preProcessing` is active at run time, at the first choice point, *$block_s$* is executed, while at the second choice point, an empty choice is made (since no code was specified for choice `preProcessing=standard`).

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

```
##BEGIN CHOICE stepType=1
block1
  ##BEGIN CHOICE stepPostOptimization
  block2
  ##END CHOICE stepPostOptimization
block3
##END CHOICE stepType
```

– *$block_1$* and *$block_3$* are executed if choice `stepType=1` is active; *$block_2$* is executed (between *$block_1$* and *$block_3$*) if `stepType=1` *and* `stepPostOptimization` 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 `stepPostOptimization`, and therefore, *$block_2$*, 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 weaver, in its *parametric 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 parametric mode and thus exposed to be controlled at run time via the command line or configuration file.

A.3 Support for exposing information collected at run-time

To provide a light-weight mechanism for exposing information collected during run-time – which could serve, for example, as input to execution controllers that adaptively control certain parameter settings at run-time – 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 needs to be a legal variable name in the programming language under consideration, and *type* indicates the data type of the expression being evaluated and is one of the type indicators described in Section A.1.[2]

Example:

```
##LOG(int timeSinceLastImpr = (currIter - lastImprIter))
```

– the value of the expression `currIter - lastImprIter` (referring to two ordinary variables in the target languague) is logged under the name `timeSinceLastImpr` as program execution reaches the location where this `##LOG` statement appears.

Resolution by the PbO weaver

The weaver replaces the `##LOG` statements with calls to routines that perform type checking and write the value of the given expression to a destination determined by weaver settings. The weaver can also be instructed, via further settings, to ignore individual logging statements (these are identified by their names), or to disable logging alltogether. In both cases, the respective `##LOG` statements are simply excised from the source. We note that logging statements intended to provide information to adaptive control mechanisms that only apply to certain design alternatives would tyically be found within instances of the respective choices, and would therefore only be activated along with those choices.

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 can be written to one or more files (in particular, to the standard output); they could also be written to a database or sent directly to an execution controller, using other mechanisms, such as remote procedure calls or network sockets.

A.4 Design objectives and potential drawbacks

The main design objective behind the generic language extensions discussed in the previous sections was to render PbO design space specifications as straightforward as possible. This concerns in particular level 1, which we believe is likely to be the entry point for many developers exploring the use of PbO. In this context, the mechanisms for exposing parameters and design alternatives need to be as intuitive and light-weight as possible, and the respective constructs have to be designed with this in mind. By allowing parameter declarations to be made anywhere in the code, and in particular, just before the first actual use of the respective parameter, it becomes possible to expose a magical constant (or hidden parameter) by means of an extremely simple and, more importantly, entirely local modification of an existing source. Likewise, we judged it important to keep the overhead involved in declaring alternative blocks of code as low as possible. We believe that these design decisions, although motivated strongly by the needs of level 1 PbO-based software development, also benefit higher levels of PbO.

A second, rather obvious design objective was to avoid clashes with target language constructs, and to provide a clean and obvious separation between the constructs of the PbO language extension and the remainder of the sources. This facilitates the construction of PbO weavers, which do not have to parse target language syntax. (This is a useful property also found in commonly used preprocessors.) The ability to parse the PbO constructs easily and independently of the target language also facilitates the implementation of special

[2]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.

support for them in software development environments, such as syntax highlighting, folding and convenient affordances for managing parameters, choices and logs.

A third goal was to allow the design space of a program to be explored without the overhead of repeated compilation during the design optimization process, as well as the instantiation of some or all design choices. The first part matters, because compilation, even when restricted to certain modules and using caching of compiled code, often takes enough time to seriously slow down automated design optimization. The second part is important, because instantiation leads to faster and leaner executables, and furthermore, because in certain contexts, it might not be desirable to release a version of a program that gives the user access to the full design space (examples include demo versions, restricted versions, as well as situations where optimization for a given use context is a paid service).

One potential drawback of the generic language extension presented here stems from the fact that the names of parameters, choices and logs are global to a software project. This raises the possibility of naming clashes, particularly when separately developed parts of a PbO-based project are integrated or merged, and may be seen to be at odds with established practices for modularization of name spaces. The decision to use global names was made because they offer the most light-weight way of dealing with the fact that design choices often cut across the structure of a program (and in particular, across module boundaries). Furthermore, we expect that even at the highest levels of PbO, there will be orders of magnitude fewer design choices (and logs) than other named entities (such as variables, functions or objects), and that therefore, segregation of name spaces is much less important for the former than for the latter. Finally, naming clashes are easily detected by the PbO weaver, and resolving them (*e.g.*, when assembling parts of a system previously developed or used in different contexts) will be easy. The need for such resolution can be further reduced by adopting canonic naming schemes for parameters, choices and logs used in libraries and other parts of a project that are designed to be reused; for example, all names introduced by PbO constructs could be prefixed with the name of the library or module to which they belong.