User Guide

mARGOt heel

Draft version

January 23, 2020

Contents

1 Introduction

The main goal of mARGOt is to provide a dynamic autotuning framework, to enhance an application with an adaptation layer. The mARGOt user guide, provides an high level description of the autotuning framework and an integration process example. This document describes mARGOt heel, which supports the autotuning framework by abstracting extra-functional concerns from implementation details. Therefore, we recommend to go through the mARGOt user manual before reading this document.

The mARGOt heel main goal is to expose to the end-user a single, simple, and consistent interface that hides as much as possible the mARGOt implementation details, named high-level interface. In particular, the end-user should state extra-functional concerns in configuration files, then mARGOt heel generates a C++ library that implements the high-level interface.

This document guides the end-user through the mARGOt integration using the high-level interface. At first, we introduce the target application model and the high-level interface, highlighting the relation with configuration files. Then, we specify the syntax and semantic of the content of each configuration files. Eventually, we show the integration process from the building system point of view, focusing on CMake.

2 Application model and high-Level interface

Section 5 of the mARGOt user manual shows an example on how the end-user can integrate the mARGOt autotuner in the target application, using directly the mARGOt API and objects. While it provides great flexibility, this low-level integration has two major flaws: it requires a deep knowledge of the internals data structure of mARGOt and it requires a fair amount of code to be inserted in the target application. To overcame these limitations, mARGOt heel hides all the implementation details behind few simple functions, meant to be used as wrappers of the target region of code to tune.

The idea is to model an application as composed of several indipendent kernels, named “blocks”. If we want to observe the performance of a block at runtime, it must be stateless; i.e. the block perfomance must depend only on the current software-knobs configuration and on features of the current input. The following is a simple application model example. For simplicity it has a single block named *foo*, but it is trivial to generalize the example with more than one block.

```
main
{
        loop
        {
                output = foo(IN input, IN knobs)
        }
}
```

For each block of code, mARGOt heel generates the following functions meant to wrap the block of code:

- **update**: set the current software-knob configuration.
- **start_monitors**: begin the measurement of all the monitors that requires a starting point (e.g. time monitor).
- **stop_monitors**: end the measurement of all the monitors that requires a stopping point (e.g. time monitor).
- **push_custom_monitor_values**: insert a new value in a custom monitor (e.g. quality monitor).
- **log**: print runtime information on file and/or standard output, according to compile time flags.

Moreover, if we want to initialize all the data structures in a given time, or if we want to log runtime information on a file, we can explicilty use the global initialization function, named **init**. If we consider the previous example, the following is the logical integration of mARGOt in the target application, considering all the exposed function.

```
main
{
        init
        loop
        {
                update(IN input,OUT knobs)
                start_monitors
                output = foo(IN input,IN knobs)
                stop_monitors
                compute custom metrics
                push_custom_monitor_values
                log
        }
}
```

The actual C++ signature of each function depends on extra-functional concerns. In particular, these are the convention used by mARGOt heel to generate the actual C++ functions with an explicit focus on their semantic:

Init function

This function initialize the mARGOt internal structure and it writes the header of the log file (if enabled). This function is optional if we disable the logging on file and if the monitor constructors parameters are known at compile time. Note that if the call to this function is omitted, the mARGOt internal structures of a block are initialized the first time that the application calls a related function.

The pseudo signature of this function is the following:

```
void margot::init(<monitor_ctor_params>, log_filename_prefix = "");
```

Differently from all the other functions that compose the high-level interface, the init function is global and it considers all the block of code. The function parameters are all the monitor constructor parameters in all the blocks managed by mARGOt, that are not known at compile time. The order of the parameters inside a monitor constructor are preserved. We force a lexicographical order between monitors and blocks. For example, the constructor parameters of the monitor "time" in the block "bar" are before the constructor parameters of the monitor "throughput" in the block "foo". The last parameter is a variable that set a prefix to the log filename. The name of the log file for a given block is "<prefix>margot.<block>.log". This option is useful if the application is composed of several processes, for example if we use MPI.

Update function

This function may update the software-knobs configuration according to the application requirements, the observed performance (if any), and the input features (if any). This function is optional if we want to use mARGOt only to profile the application.

The pseudo signature of this function is the following:

```
bool margot::<block>::update(const <features>, <software_knobs>&);
```

It returns a boolean that states whether the software-knobs configuration has changed, returning false if the configuration is the same of the previous one. mARGOt heel instantiate this function for each block of code inside a namespace named after the block name. The function parameters are the input features (if any) and the software knobs. The software knobs are the output parameters updated by the function. In the function declaration, the features are before the knobs. Both, the features and the knobs are lexicographically sorted.

Start_monitors function

This function call the start method of all the known monitors shipped with the mARGOt framework that are stated in the application requirements. This function is optional if we don't use known monitors.

The pseudo signature of this function is the following:

```
void margot::<block>::start_monitors(<monitor_start_params>);
```

mARGOt heel instantiate this function for each block of code inside a namespace named after the block name, even if there are no known monitors stated in the application requirements. The parameters of this function are all the parameters that a monitor may require to start a measure. The order of the parameters inside the start method of a monitor are preserved. We force a lexicographical order between monitors. In the current version of the framework, no known monitor requires start parameters.

Stop_monitors function

This function call the stop method of all the known monitors shipped with the mARGOt framework that are stated in the application requirements. This function is optional if we don't use known monitors.

The pseudo signature of this function is the following:

```
void margot::<block>::stop_monitors(<monitor_stop_params>);
```

mARGOt heel instantiate this function for each block of code inside a namespace named after the block name, even if there are no known monitors stated in the application requirements. The parameters of this function are all the parameters that a monitor may require to stop a measure. The order of the parameters inside the stop method of a monitor are preserved. We force a lexicographical order between monitors. In the current version of the framework, only the throughput monitor requires a stop parameter.

Push_custom_monitor_values function

This function aims at providing an easy way to the application developer for measuring a custom metric, such as quality. In the past we suggested a more strict object-oriented approach where the user inherit from a base monitor class and he/she must implement the "start" and "stop" methods to gather the measure. However, we noticed that it is way easier and less cumbersome to just use the base monitor and to push the computed value. Therefore, we introduced this function that takes as input the value of the custom metric and insert it in the related monitor. This function is separated by the stop_monitor function to avoid to measure the time taken to compute the custom metric.

The pseudo signature of this function is the following:

```
void margot::<block>::push_custom_monitor_values(<custom_measurement>);
```

mARGOt heel instantiate this function for each block of code inside a namespace named after the block name, even if there are no custom monitors. The parameters of this function are the stop parameters of the custom monitor, we should be the value to insert in the monitor. We force a lexicographical order between monitors.

Log function

This function display information about the extra-functional properties of the execution. In particular, it display information about the monitored values, the software-knobs configuration, the current input features, and the constraint values. According to configure-time variable, it can log the information on the standard output and/or on a file. In the second case it uses the csv format to store information. The idea is to generate a log file for each block of code handled by margot.

The pseudo signature of this function is the following:

```
void margot::<block>::log(void);
```

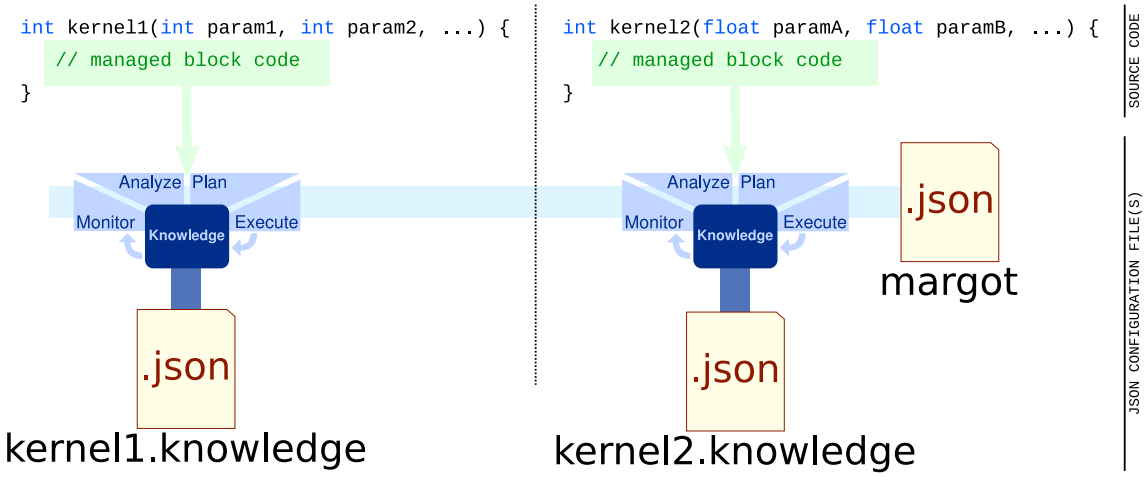

Figure 1: Graphical representation of the relationship between configuration files and extra-functional concerns, in an application example.

3 Configuration overview

The mARGOt autotuning framework implements a Monitor-Analyze-Plan-Execute loop based on application knowledge (MAPE-K). The application Knowledge describes the application behavior, by varying the selected software-knobs configuration and according to the input features. In the mARGOt context, this knowledge is defined as a list of points, named Operating Points. Each Operating Point relate the perfomance reached by the application using a given software-knobs configuration, in the context of the given input features. The application knowledge might be derived at design-time, using a well-known algorithm to perform a Design Space Exploration, or at runtime, using a dedicated component (Section **??**). The extra-functional concerns define the Monitor-Analyze-Plan-Execute, by specifying all the information required to instantiate a MAPE loop tailored for the target application.

From the integration point of view, we need to create at least one configuration file, written in json format, to enable the high-level interface generation. In particular, all the extra-functional concerns of all the blocks that compose the application, must be written in a single file. Section 5 describes in detatils its semantic and systax. Then, if we have gathered the application knowledge at design time, we need to write one or more configartion files that describes the Operating Point list for each block managed by mARGOt. Section 4 describes in details the semantic and syntax of those files.

To clarify the concept, Figure 1 shows an example of integration. In particular, we suppose that the application is composed of two block of code managed by mARGOt, contained in two functions, named “kernel1” and “kernel2”. The two block of codes represent independent phases of the application, therefore they have completely different extra-functional requirements. This leads to the instantiations of two different MAPE loops, based on knowledge, that drive the selection of two different set of software-knobs. To

```
{
  "foo":
  [
    {
      "features": { "resolution": 34 },
      "knobs": { "threads": 3, "algorithm": "one" },
      "metrics": { "throughput": [23.0, 2.0], "quality": 32 }
    },
    {
      "features": { "resolution": 34 },
      "knobs": { "threads": 4, "algorithm": "two" },
      "metrics": { "throughput": [2.0, 0.5], "quality": 100 }
    }
  ]
}
```

Figure 2: Example of a knowledge configuration file for a block named “foo”.

generate this interface, it is required to write a configuration file (named “margot.json” in this example) with all the extra-functional concerns. Then, we create a configuration file for each block, with the application knowledge (named “kernelX.knowledge.json” in this example).

4 Knowledge configuration file

The configuration file that describes the application knowledge must be a different file with respect to the one that describes extra-functional concerns, to ensure separation of concerns. The structure of the configuration file matches the application knowledge representation of mARGOt. In particular, it is a list of Operating Points (OPs), where each OP has up to three fields:

1. The feature section (if any)
2. The software-knobs section
3. The metric section

The feature section states the value for each input feature field. Since mARGOt use input features as indipedent cluster of Operating Point, a best practice would be to ensure that all the input feature clusters contain OPs with the same configurations of software-knobs.

Each section of the OP is basically a key-value pair, where the key is the name of field and the value is a number, which represents its average value. In the software-knob section, the field can be a string. In the metric section, we can use two values to define the mean and standard deviation of the related distribution. These two exceptions are enabled by the the extra-functional concerns stated in the mARGOt configuration file, explained in details in Section 5.

```
{
  "name": "bar",
  "version": "4.0",
  "blocks":
  [
    { "name": "block1", ... },
    { "name": "block2", ... }
  ]
}
```

Figure 3: The top-level description of the apllication extra-functional cencerns.

Figure 2 shows an example of the syntax of the knowledge file, that defines an Operating Point list for a block named “foo”, by defining a root element with the same name that will contain the OP list. A configuration file that represents the application knowledge may contain more than one OP list, if the application is composed of more than one block of code. However, this behaviour is not recommended to ensure separation of concerns. The application knowledge of a block of code can be splitted in more than file, to limit its size. In this example, we have just a single input feature named “resolution”, defined in the element with tag **features**. If we don’t have any input feature, the **features** element can be omitted. We have two software-knobs, named “threads” and “algorithm”, defined in the element with tag **knobs**. The values of the second knob are strings. Finally, we have two metrics, named “throughput” and “quality”, defined in the element with tag **metrics**. For the metric “quality”, the average value is enough to describe the reached performance. It may happen when the metric is determistic. Instead, the metric “throughput” is a distribution defined by an average value and the standard deviation. While the OPs that belong to the same block managed by mARGOt have the same structure, the actual values changes to define the application knowledge.

5 Adaptation file specification

This section describes in details the syntax and semantic of the configuration file that states the extra-functional concerns. In particular, it follows at top-down approach, where the root element represents the application and it contains all the block descriptions. Figure 3 shows an example of a configuration file skeleton. The root element must define the application name and version, using the tag **name** and **version**. The **blocks** element contains the list of all the block managed by margot. In this example, we have a block named “block1” and a second one named “block2”. The name of the block is represented by the element **name** and it must be a valid C++ identifier. The remainder of the section will expand the block description, defining all its sub-elements.

5.1 How to define monitors

All the monitors that belong to a block are defined as a list contained in the **monitors** element. Each monitor is defined by the following elements:

name : the monitor's name, it must be a valid C++ identifier and it must be unique.

type : the monitor's type. Appendix A provides the list of monitors shipped with mARGOt. If the type is arithmetic type, it will create a custom monitor of that type.

log : the list of statistical properties that we want to display in the log. Each element of the list must assume one of the following values: **average**, **standard_deviation**, **max**, and **min**.

constructor : the list of parameters that the monitor's constructor is expecting. The order of the parameters must match the constructor signature. If we want to override the default parameters, we need to specify all of them. Appendix A provides more details on constructor parameters for the known monitors. Custom monitors have a single constructor parameter: the observation window size.

start : the list of parameters for starting a mesure. In the current version, no monitors requires any parameter to start a measure.

stop : the list of parameters for stopping a measure. In the current version, only the throughput monitor and the custom monitor requires a stop parameter, which are the amount of data elaborated and the value to push in the monitor, respectively.

All the parameters specified in the *constructor*, *start*, and *stop* fields can be an immediate or a variable. An immediate parameter is a string that it is used "as it is", therefore it is set a compile time. A variable parameter is a pair of strings that represents a C++ variable that will be exposed to the user through as parameter of the high-level interface, therefore it can change at runtime. In particular, the first value of the variable parameter is its C++ type, while the second is its name. Therefore, the latter must be a valid C++ identfier. Moreover, to ensure the uniqueness of the variable parameters exposed in the high-level interface, the actual name of the parameter exposed through the high-level interface is "<monitor_name>_<parameter_name>".

Figure 4 shows an example that defines an error monitor and a time monitor. The error monitor will store 22 elements of type *floats*, to display the average in the logs. Each observation is pushed in the monitor using a *float* variable named "error". This variable will be exposed to the user as parameter of the **push_custom_monitor_values** high-level function. According to generation convention, the name of the parameter will be "*error_monitor_error*". The second monitor observe the elapsed time between the **start_monitors** and **stop_monitors** high-level functions. Since all its parameters are immediate, they will be compile-time constant in the generated code.

```
"monitors":
[
  {
    "name": "error_monitor",
    "type": "float",
    "log": [ "average" ],
    "constructor": [ 22 ]
    "stop": [ {"error":"float"} ]
  },
  {
    "name": "time_monitor",
    "type": "time",
    "log": [ "average" ],
    "constructor": [ "margot::TimeUnit::MICROSECONDS", 1 ]
  }
]
```

Figure 4: This is an example on how we can define of two monitors

5.2 How to define software-knobs

All the block software-knobs are defined as a list contained in the **knobs** element. Each software-knob is defined by the following elements:

name : the knob name, it must be a valid C++ identifier.

type : the C++ type of the knob. It can be an arithmetic type or a string. If the type is integral, it will be aliased with a fixed size. For example, “int” will be automatically aliased to “int32_t” in certain systems. If the type is a string, then it will be considered as an enumeration. Therefore, it must be specified in the configuration file all the possible values that the knob can take, using the **values** field.

values : the list of all the possible values that the knob may take. This field is mandatory if we plan to learn the application knowledge at runtime or if we the know type is a string.

range : it’s a list composed of two or three values (*min*, *max*, and *step*) that generates all the values in the range [*min*; *max*) with step *step*. The parameter *step* is optional and its default value is 1. It replaces the **values** field.

The software-knobs are part of the application geometry, along with the definition of the metrics of interest. They are optional, meaning that we can omit this element in a block defition. However, in this case we are not able to tune the application, but we can use the mARGOt facilities to monitor its performance. If we define the software-knobs, we need to define also the metrics of interest.

Figure 5 shows an example that defines two software-knobs, named “threads” and “algorithm”. Since the first one represents the number of threads used in an application, its type is integral. Moreover, if we assume that the target platforms has 32 hardware thread,

```
"knobs":
[
  {
    "name": "threads",
    "type": "int",
    "range": [ 1, 33, 1 ]
  },
  {
    "name": "algorithm",
    "type": "string",
    "values": [ "one", "two", "three" ]
  }
]
```

Figure 5: This is an example on how we can define of two software-knobs

its range spans from 1 thread to 32, with step 1. The second software-knob represents the name of the algorithm to use in the elaboration. Therefore, it defines the enumeration of all the possible values that the knobs can have.

5.3 How to define metrics of interest

All the metrics of interest are defined as a list contained in the **metrics** element. Each metric is defined by the following elements:

name : the metric name, it must be a valid C++ identifier.

type : the C++ type of the knob. It must be an arithmetic type. If the type is integral, it will be aliased with a fixed size. For example, “int” will be automatically aliased to “int32_t” in certain systems.

distribution : tells whether this metric is determistic or not. Which means that if the value is true, we need a mean and a standard deviation to define a value.

observed_by : relate this metric to a monitor. This field is required if we need to adapt reactively of if we need to learn the application knowledge at runtime.

reactive_inertia : the size of circular buffer that adjust the expected metric value, according to the runtime observation. Larger values lead to a slower reaction. If the value is zero, it means that we don’t want to adapt reactively at runtime. This field is optional and its default value is zero.

prediction_plugin : the name of the plugin in charge of predicting this metric. This field is meaningful only if we learn the application knowledge at runtime.

prediction_parameters : a list of key-value parameters that we would like to forward to the prediction plugin. This field is meaningful only if we learn the application knowledge at runtime.

```
"metrics":
[
  {
    "name": "throughput",
    "type": "float",
    "distribution": "yes",
    "observed_by": "throughput_monitor",
    "reactive_inertia": 5,
    "prediction_plugin": "hth",
    "prediction_parameters":
    [
      {"param1": "value1"},
      {"param2": "value2"}
    ]
  },
  {
    "name": "quality",
    "type": "float",
    "observed_by": "quality_monitor",
    "prediction_plugin": "hth"
  }
]
```

Figure 6: This is an example on how we can define of two software-knobs

The metrics of interest are part of the application geometry, along with the definition of the software-knobs and input features. They are optional, meaning that we can omit this element in a block defition. However, in this case we are not able to tune the application, but we can use the mARGOt facilities to monitor its performance. If we define the metrics of interest, we need to define also the software-knobs.

Figure 6 shows an example that defines two metrics, named “throughput” and “quality”. In this example we consider the throughput as distribution since the measure depends on several factors that are unknown at priori and that depends on the system evolution. Therefore, we want to react on those changes, but with some inertia to lower the effect of noise. To predict this metric we use the plugin “hth”, where we define two additional parameters to improve the prediction. Moreover, in this example we consider the error determistic, therefore we can describe its values using only the mean. Since computing the error might be expensive to perform at runtime, we don’t adapt reactively. Indeed, we use the measurement only to learn its behaviour at runtime, using the plugin “hth”.

5.4 How to define input features

The input features are defined by two elements. The **feature_distance** element which represent how mARGOt should compute the distance between two input features. This element can have only two values: *euclidean* or *normalized*. The **features** element stores the list of all the fields that compose the input features. Each field is defined by the following elements:

```
"feature_distance":"euclidean",
"features":
[
  {
    "name": "f1",
    "type": "int",
    "comparison": "le"
  }
]
```

Figure 7: This is an example on how we can define input features

name : the knob's name, it must be a valid C++ identifier.

type : the C++ type of the knob, it must be an arithmetic type.

comparison : the type of constraint that we would like to impose for the cluster selection. It is possible to use one of the following values:

- le to express the "less or equal than" constraint
- ge to express the "greater or equal than" constraint
- - if we don't care about imposing a constrant. This is the default value.

The input features are part of the application geometry, along with the definition of the metrics of interest and software-knobs. Differently from the former sections, the input feature definition is not mandatory. However, if we define input features, we need to define also the metrics of interest and software-knobs.

Figure 7 shows an example that defines the input features as composed by a single integer field named "f1". We impose a constraint on the selection of the input feature cluster. In particolar, we state that the feature *f1* of a cluster must be lower or equal than the feature of the current input to be selected. Moreover, we stated that we want to select the closest one, among the eligble ones, computing the euclidean distance.

5.5 How to learn the application knowledge at runtime

If we want to learn the application knowledge at runtime, we need to connect to an MQTT broker and provide information about DoE and clustering (if any). All these information are contained in the **agora** element, where we can specify the following information as inner elements:

borker_url : the string containing the url of the target broker in the form <*protocol*>://<*url*>:<*port*>. The protocol can be either *tcp* or *ssl*; and the port usually is 1883 for tcp or 8883 for ssl. Both, the protocol and port can be omitted. The default value depends on the MQTT driver implementation.

broker_username : the username used to log in the MQTT broker, leave empty if not required.

broker_password : the password used to log in the MQTT broker, leave empty if not required.

broker_qos : the amount of effort spent by MQTT to check if the receiver has received the sent message. It is an integral value in range [0,2], where with the value 2 MQTT spend most effort.

broker_ca : the path to the broker certificate, leave empty if not required.

client_cert : the path to the client certificate, leave empty if not required.

client_key : the path to the key of the client certificate, leave empty if not required.

clustering_plugin : the name of the plugin in charge of clustering the observed input features. This field is meaningful only if we have input features.

clustering_parameters : a list of key-value parameters that we would like to forward to the clustering plugin. This field is meaningful only if we have input features.

doe_plugin : the name of the plugin in that defines the DoE to explore at runtime.

doe_parameters : a list of key-value parameters that we would like to forward to the DoE plugin.

Beside the information required to connect with the MQTT broker, all the other information are forwarded to the remote component in charge of orchestrating the DSE, without validating them. If the connection with broker fails, the generated code rise an *std::runtime_error*, since the specification of the **agora** component is required only if we want to learn the application knoweldge and it can't happen without a MQTT broker.

Figure 9 shows an example where mARGOt will connect locally to the MQTT broker, with authentication. Since we are communicating locally, encription is not required. Moreover, we forward two parameters to the clustering pluging, while we forward a single parameter.

5.6 How to define the optimization problem

The element that contains the list of optimization problem that defines the application requirements is **extra-functional_requirements**. Each optimization problem is defined by a name (using the element **name**), the objective function (named rank in mARGOt context), and the list of constraint.

The optimization function is defined by the element **maximize** or **minimize**, according to the rank direction. The only child element of the rank direction is how the user would combine the different rank fields, using either the element **geometric_mean** or **linear_mean**. The child of the latter element is a list, composed of rank field. Each rank field is defined by a key-value element. The key is the name of the related metric or software-knob. The value is its coefficient in the formula combination.

The list of constraint is represented in the **subject_to** element, where the order in which they appear in the configuration file set the priority between them. In particular the constraint on the top has higher priority of the one in the bottom. Each constraint is defined by the following fields:

```
"agora":
{
  "borker_url": "127.0.0.1:1883",
  "broker_username": "margot",
  "broker_password": "margot",
  "broker_qos": 2,
  "broker_ca": "",
  "client_cert": "",
  "client_key": "",
  "clustering_plugin": "clusty",
  "clustering_parameters":
  [
    {"algorithm": "kmeans"},
    {"number_centroids": 5}
  ],
  "doe_plugin": "jhondoe",
  "doe_parameters":
  [
    {"constraint": "knob1 + knob2 < 40"}
  ]
}
```

Figure 8: This is an example on how we can define the information required to learn the application knowledge at runtime

subject : the name of the target metric or software-knob.

comparison : the kind of comparison used in this contraint. The semantic is that <*field*> must be <*operator* than *value*.

value : the initial value of the constraint. It can be updated at runtime.

confidence : the number of times that we need to consider the standard deviation of the Operating Point. If we want to consider only the mean value, we can set the confidence to zero.

The specification of the optimization problem happens during the initialization of the data structures. If we want to update the value of a constraint, we can do it by setting the value of the related goal. In particular, the related goal is automatically generated and its name is <*name*>_*constraint*_<*index constraint*>. For example the following instruction set the goal on the second constraint of the optimization problem named “problem”, in the block named “elaboration”.

```
margot::elaboration::context().goals.problem_constraint_1.set(25);
```

Moreover, if we define more than one optimization problem, the application is in charge of selecting the starting one. By default it will be the last problem defined in the configuration file.

```
"extra-functional_requirements":
[
  {
    "name":"default",
    "maximize":
    {
      "geometric_mean":
      [
        {"quality": 2},
        {"throughput": 5}
      ]
    },
    "subject_to":
    [
      {
        "subject":"throughput",
        "comparison": "ge",
        "value": 25.0,
        "confidence": 3
      }
    ]
  }
]
```

Figure 9: This is an example on how we can define the information required to learn the application knowledge at runtime

Figure 9 shows an example where we define a single optimization problem named "default". In terms of mARGOt representation, the target problem is the following:

$$\begin{array}{rll} \max & E(quality)^2 \times E(throughput)^5 & \\ \text{s.t.} & C_0 : E(throughput) - 3 \times \sigma_{throughput} & \leq \quad 25 \end{array} \tag{1}$$

where the mean and standard deviation of the *quality* and *throughput* may change if we change software-knobs configuration.

6 Integration Process

The mARGOt heel high-level interface extends the mARGOt API with a set of high-level interface that aims at lowering as much as possible the integration effort. Following this trend, we provide facilities to minimize the integration effort from the compilation process point of view, if the end-user uses *CMake* as building system. The first section states the requirements in order to build mARGOt and mARGOt heel. Then, we start explaining how to inegrate mARGOt in a target application with the easy way, which relies on CMake. Finally, we explain the integration process that can be ported in a generic build system, when we want to take the hard way.

6.1 Build requirements

The mARGOt autotuning framework uses standard C++11 features. However, it uses the Paho MQTT C++ client library to communicate with the broker (https://github.com/eclipse/paho.mqtt.cpp), which in turns requires the OpenSSL development files. Moreover, mARGOt heel requires a compiler that support *std::filesystem* (C++17) and the boost library to parse a json configuration file. We strongly suggest to handle these dependencies externally, for obvious reasons, but we included for convenience two scripts to download and compile OpenSSL and PahoMQTT.

6.2 CMake-based integration

The main idea of the integration is to hide as much as possible implementation details. Therefore, if the building system of the target application is CMake, the integration procedure follows these steps:

1. Import mARGOt heel
2. Call a CMake function to configure the high-level interface generation
3. Link the high-level interface to the executable

We use modern CMake facilities to automatically resolve all the dependence chains and to generate the high-level interface at compile time, rather than configure time. In this way, the high-level interface is generate if, and only if, we update the configuration files or if we update mARGOt heel. The benefit of this choice is twofold: on the main hand, we lower

the compilation time during the application development. On the other hand, we are sure to always use the most updated high-level interface.

6.2.1 Import mARGOt heel

How to perform this step depends on whether we want to use mARGOt as an application model or if it is shared among other applications. In the first case, we expect the user to clone the mARGOt repository in the source folder of the application. In this case, to import mARGOt, it is enough to use the following code:

```
add_subdirectory( path/to/margot )
```

This procedure has the advantage that the mARGOt compilation process is tied to the compilation process of the application. For example, if the user would like to compile the application in DEBUG mode, also mARGOt will compiled in DEBUG mode.

If we plan to integrate mARGOt in several application, it is better to proceed with this two-step procedure. At first we have to compile and install mARGOt, using the classic CMake procedure:

```
$ git clone https://gitlab.com/margot_project/core.git
$ cd clone
$ mkdir build
$ cd build
$ cmake -DCMAKE_INSTALL_PREFIX:PATH=/install/path ..
$ make
$ make install
```

If we omit the install prefix, it will use the project directory. Please notice that the CMake finders of mARGOt are stored in the folder *<prefix>/lib/cmake/margot*. In the second step, we add to the CMakeLists.txt of each application, the lines to import the mARGOt heel high-level generator, for example:

```
set(margot_heel_generator_DIR "/install/path/lib/cmake/margot")
find_package(margot_heel_generator REQUIRED)
```

This procedure has the advantage that mARGOt can be downloaded and compiled only once. All the application import and share its implementation.

6.2.2 Configure the high-level interface generation

The next step is to configure the high-level interface generation. In order to do so, we need to write the mARGOt configuration file, along with the ones for the operating points. Then, it is enough to call the following function, that takes as input parameters the list of configuration files, where the mARGOt configuration must be the first one.

```
margot_heel_generate_interface( "path/to/margot.json" "path/to/ops.json" )
```

From this moment, we can use the target *margot::margot_heel_interface* that represents the high-level interface and its dependencies, such as the mARGOt framework.

The actual source files of the library are created in the build directory, as defined in *CMAKE_CURRENT_BINARY_DIR*.

6.2.3 Link the target library

The last step to integrate mARGOt heel in the building system, is to link the target *margot::margot_heel_interface* to the executable targets. In this way, CMake automatically set includes directories, linking flags, and build dependencies.

6.3 Source code integration

To access the high-level interface of mARGOt, it is only required to include the *margot/margot.hpp* header file.

A Monitor implementation details

This appendix lists, for each monitor, all the parameters of its constructor, start measure method and stop measure method. These information might be used when a monitor is declared in the XML configuration file.

A.1 Collector Monitor

Constructor parameters

1. Default constructor
    - Size of the observation window (default = 1)
    - Minimum number of observations (default = 1)
2. Custom connection constructor
    - Name of the topic of interest (string)
    - Address of the MQTT broker (string)
    - Size of the observation window (default = 1)
    - Minimum number of observations (default = 1)

Start method parameters

N/A

Stop method parameters

N/A

A.2 Energy Monitor

Constructor parameters

1. Default constructor
    - Size of the observation window (default = 1)
    - Minimum number of observations (default = 1)
2. Custom domain constructor
    - Domain of interest
        (a) margot::EnergyMonitor::Domain::Cores
        (b) margot::EnergyMonitor::Domain::Uncores
        (c) margot::EnergyMonitor::Domain::Ram
        (d) margot::EnergyMonitor::Domain::Package

- Size of the observation window (default = 1)
- Minimum number of observations (default = 1)

Start method parameters

N/A

Stop method parameters

N/A

A.3 Frequency Monitor

Constructor parameters

1. Default constructor
    - Size of the observation window (default = 1)
    - Minimum number of observations (default = 1)

Start method parameters

N/A

Stop method parameters

N/A

A.4 Odroid Energy Monitor

Constructor parameters

1. Default constructor
    - Size of the observation window (default = 1)
    - Minimum number of observations (default = 1)
2. Custom period time
    - Period of polling (unsigned integer)
    - Size of the observation window (default = 1)
    - Minimum number of observations (default = 1)

Start method parameters

N/A

Stop method parameters

N/A

A.5 Odroid Power Monitor

Constructor parameters

1. Default constructor
    - Size of the observation window (default = 1)
    - Minimum number of observations (default = 1)

Start method parameters

N/A

Stop method parameters

N/A

A.6 Papi Monitor

Constructor parameters

1. Trivial Default constructor

    N/A

2. Custom event constructor
    - PAPI event of interest
        (a) margot::PapiEvent::L1_MISS
        (b) margot::PapiEvent::L2_MISS
        (c) margot::PapiEvent::L3_MISS
        (d) margot::PapiEvent::INS_COMPLETED
        (e) margot::PapiEvent::NUM_BRANCH
        (f) margot::PapiEvent::NUM_LOAD
        (g) margot::PapiEvent::NUM_STORE
        (h) margot::PapiEvent::CYC_NO_ISSUE
        (i) margot::PapiEvent::CYC_TOT
    - Size of the observation window (default = 1)
    - Minimum number of observations (default = 1)

Start method parameters

N/A

Stop method parameters

N/A

A.7 Process CPU Usage Monitor

Constructor parameters

1. Default constructor
    - Size of the observation window (default = 1)
    - Minimum number of observations (default = 1)
2. Custom CPU time counter constructor
    - CPU time counter
        (a) margot::CounterType::SoftwareCounter
        (b) margot::CounterType::HardwareCounter
    - Size of the observation window (default = 1)
    - Minimum number of observations (default = 1)

Start method parameters

N/A

Stop method parameters

N/A

A.8 System CPU Usage Monitor

Constructor parameters

1. Default constructor
    - Size of the observation window (default = 1)
    - Minimum number of observations (default = 1)

Start method parameters

N/A

Stop method parameters

N/A

A.9 Temperature Monitor

Constructor parameters

1. Default constructor
    - Size of the observation window (default = 1)
    - Minimum number of observations (default = 1)

Start method parameters

N/A

Stop method parameters

N/A

A.10 Throughput Monitor

Constructor parameters

1. Default constructor
    - Size of the observation window (default = 1)
    - Minimum number of observations (default = 1)

Start method parameters

N/A

Stop method parameters

1. Amount of data processed (float)

A.11 Time Monitor

Constructor parameters

1. Default constructor
    - Size of the observation window (default = 1)
    - Minimum number of observations (default = 1)
2. Custom time unit constructor

- Time resolution
  (a) margot::TimeUnit::NANOSECONDS
  (b) margot::TimeUnit::MICROSECONDS
  (c) margot::TimeUnit::MILLISECONDS
  (d) margot::TimeUnit::SECONDS
- Size of the observation window (default = 1)
- Minimum number of observations (default = 1)

Start method parameters

N/A

Stop method parameters

N/A