EDACC
User Guide
version 0.1

Zusammenfassung

We present the main capabilities of EDACC and describe how to use EDACC for managing solvers and instances, create experiments with them, launch them on different computer clusters, monitor them and then analyze the results.

disclaimer!

Inhaltsverzeichnis

1 User guide zum User Guide

Bevor Ihr etwas in diesem user guide was reinschreiben wollt/sollt bitte diesen Kapitel durchlesen. Alle Autoren sollten versuchen die folgenden Richlinien zu folgen, sodass die Arbeit ein homogenes Erscheinen bekommt auch wenn viele Autoren dran arbeiten.

1.1 Darstellungskonventionen

Das ist ein User Guide, folglich sollten die wichtigen Informationen für den Benutzer sehr schnell auffindbar sein. Dazu verhelfen folgende Konzepte:

Index

Indexierung aller wichtigen Wörter; insbesondere Schlüsselwörter. Der Index und die pdf-hyperreferenzen sollen ein schnelles suchen ermöglichen. Setzen von index wörter mit `\index{wort}`.

seitlicher Hinweis auf

Um wichtige Dinge schnell zu finden sollen auch seitliche Hinweise helfen. diese werden mit `\marginlabel{schlüsselwort}` gesetzt.

Example 1:

Auf Beispiele weist man am besten mit `\marginlabel{\Eexample}` hin.

Verweise

→ Section 2

Wenn man auf eine andere stelle im Text refenrenzueren will, was man so oft wie möglich machen sollte dann erscheint die Referenz auch im linken Rand mit: `\seealso{label}`. Wie zum Beispiel: Ein Überblick über diesen user-guide gibts in Kapitel 2 .

Hinweise

! →

Falls etwas doch sehr wichtig sein sollt, im Regel Besonderheiten auf die man achten sollte so kann man den Leser mit `\attention` darauf aufmerksam machen!

Glosareinträge

Wichtige Terme sollten am besten auch eine Definition haben, oder wenigstens eine Erklärung was damit gemeint ist. Das kann man am besten mit `\glossary{name={Glossareintrag}, description={Beschreibung}}` .

1.2 Inhaltliche Konventionen

Audienz identifizeren

Ich gehe davon aus, dass die meisten Leute die EDACC verwenden werden Informatiker sein werden, oder eine Unterart davon. Folglich können wir davon ausgehen, dass sie mit den gängisten Begriffe vertraut sind (Ein Glossareintrag zu diesen würde tortzdem nicht schaden). Der Benutzer wird immer als **user** im Text angesprochen. Ich gehe auch davon aus, dass die meisten users diese Hilfe als Nachschlagewerk verwenden werden. Folglich soll auch der Inhalt task-orientiert sein. Das bedeutet dass man nicht das System an sich versucht zu beschreiben sondern die Aufgaben beschreibt und dadurch eher die Systembeschreibung entsteht. Es ist ungefähr das Gegenteil von dem Paper, wo nur die abstrakten Konzepte des Systems beschrieben worden sind. Also wenn Eure Doku zu sehr nach paper klingt seid ihr auf dem falschen Weg.

Verwendungsart des user-guides

System definieren

Das System wenn es als Ganzes erwähnt wird sollte mit **EDACC** erwähnt werden. Allerdings sollte man vermeiden das Obersystem in Beschreibungen zu verwenden. Besser ist es wenn man sich immer auf dessen Teilkomponenten bezieht: **DB, GUI, client, WF**. **Monitor** wird nur als ein Teil des **WF** betrachtet.

Workflow angelehnte Beschreibung

Die Beschreibung der einzelenen Aufgaben sollte an dem typischen Workflow von EDACC orientiert sein. Die allgeinen Konzepte und Probleme werden in der Anleitung sehr gut beschrieben sein. Im Hauptteil des userguides sollen die Aufgaben beschrieben werden die man mit EDACC lösen kann. Dabei achtet man auf:

Task-Beschreibung

1. Identifizierung der Aufgaben (z.B:"Solver hinzufügen")
2. Aufteilung der Aufgabe in Unteraufgaben: (z.B: "Solver binaries verwalten und die Parameter spezifizieren")
3. Jede Aufgabe wird in Schritten beschrieben die durchnummeriert werden. (z.B: 1. name des solvers eintragen 2. Author eintragen 3. Version (name und verison müssen eindeutig sein))
4. Fallunterscheidungen sollen explizit beschrieben werden wenn der Benutzer eine Entscheidung treffen muss (if-then-else Formulierungen). z.B: (Falls ein Parameter als Boolean markiert wird so ...)
5. Reichlich refenrenzieren und verweisen auf Glossar, Index und andere Stellen im Text wo (verwandte / benötigte) (Aufgaben / Konzepte) beschrieben werden.

2 Outline

Here we will have an overview of this user guide specifying where the user can find what!

3 Introduction

Specification of the problems algorithm engineers are often confronted with!

Algorithm engineering:

When designing and implementing algorithms one is at the end of the process confronted with the problem of evaluatating the implementation on the targeted problem set. As the authors of *EDACC* are familiar with algorithms for the satisfiability problem we will take this sort of algorithms as examples. After designing and implementing a SAT solver we would like to see how this performs on a set of instance problems (let us suppose that our solver is an implementation of a stochastic one i. e.,the result of the solver on the same instance will be a random variable).

Normally we would start our solver on each instance and record the runtime or some quality measure. This is a sequential process and could be easily performed with the help of a simple shell script. But there are some questions that have to be answered before starting this evaluation process.

1. How long is the solver allowed to compute on one instance?
2. In the case of randomized solvers, how often do we call the solver on each problem set?
3. Do we limit the resources used by the solver (i. e.,maximum of memory, maximum stack size)?

What is the task?
What are the tools?
What would we like to achieve?

Description of two example scenarios:

Example 2:

See how the performance of a deterministic solver differs from other solvers!

Example 3:

3.1 What is EDACC

Describe the main tasks of EDACC, what is it for, what problems can be solved with EDACC?

3.2 System Requirements

3.3 Getting started

Components of EDACC

DB: Here we are going to describe the data that EDACC uses.

UI: How to work with EDACC.

Client:

→ Chapter 6

For more details about the requirements of the client and the command line see 6

4 User Interface

5 Parameter search space specification [Draft]

5.1 Definitions

A parameter is an input variable of a program and is defined by a name, a domain, a prefix (which can also be empty) and an order number. Additionally there are two booleans. "space" that indicates whether to put a space between the prefix and the value. "attach to previous" indicates whether there should be a space between the previous parameter (according to order) and this one. "attach to previous" can be useful for parameters that look like "-prefix v1[,v2,v3]". ",v2" and ",v3" can be modeled as parameters that attach to the preceding "-prefix" parameter.

Definition 5.1:
A solver configuration is a list of parameters and their assigned values.

Definition 5.2:
A domain defines the set of possible values that can be assigned to a parameter (in a solver configuration). It can be one of the following or the union of any number of them (except for the flag domain, which can only occur on its own).

1. real: values between a lower and an upper bound
2. integer: values between a lower and an upper bound
3. ordinal: list of values in a min to max order
4. categorial: set of possible values
5. optional: consists only of a special value not specified"
6. flag: consists of two special values önänd öff"(for parameters that are flags, i.e. present or not)

Definition 5.3:
The parameter space of a solver is defined by its parameters and their possible values. The parameter space can be further constrained by dependencies between parameters such as

1. Parameter X can be specified if parameter Y takes on certain values
2. Parameter X has to be specified if parameter Y takes on certain values
3. Parameter X has to take on certain values depending on the values of parameters Y, Z, ...

There are several tasks that come up in the context of EDACC: Determine if a given solver configuration is valid, i.e. in the solver's parameter space. Given the parameter space, construct a valid solver configuration. Given a valid solver configuration, find a "neighbouring" solver configuration that is also valid.

Definition 5.4:
A parameter graph is a directed, acyclic graph that represents the parameter space. It consists of AND-Nodes and OR-Nodes and edges between them. Edges are directed and allowed only between different types of nodes. OR-Nodes can have multiple incoming edges, while AND-Nodes can only have exactly one incoming edge. Additionally edges have a group number which is 0 if the edge doesn't belong to any group. Parameter graphs have a single unique AND-Node without any incoming edges. This node will be referred to as start node.

Definition 5.5:
OR-Nodes have a reference to a parameter.

Definition 5.6:
AND-Nodes have a domain and a parameter reference to the same parameter as the preceding OR-node. AND-Nodes partition the possible values of the parameter that they (and the preceding OR-node) reference. The domain of an AND-Node has to be a subset of the domain of the preceding OR-Node.

The general idea is that the parameter space is specified by following the structure of the graph from the start node and constraining the parameters using the domains encountered on the nodes. AND-Nodes imply that all outgoing edges have to be followed while OR-Nodes mean that exactly one edge has to be followed.

More formally:

Definition 5.7:
A solver configuration is valid if the start node (an AND-Node) of the parameter graph is satisfied. Satisfied means:

1. an AND-Node is satisfied if the corresponding parameter value lies in its domain and all OR-nodes adjacent via ungrouped edges are satisfied.
2. an OR-Node is satisfied if exactly one adjacent AND-Node is satisfied and for at least one set of incoming edges with common group number the preceding AND-Nodes are all satisfied.

5.2 Algorithms on parameter graphs

Constructing a valid, random solver configuration:

```
Input: parameter graph; Output: (random) solver configuration
done_and := {startnode} # set containing only the start node
done_or := {} # empty set

L := list of OR-nodes adjacent to startnode

while (L has nodes with >= 1 group of edges coming from \
       AND-nodes in done_and):
    or_node := choose and remove such a node randomly from L
    done_or.add(or_node)
    and_node := choose an AND-node adjacent to or_node
                randomly (or with user input)
    done_and.add(and_node)
    Assign a random (or user chosen) value from and_node.domain to
     the and_node.parameter of the solver configuration

    Add all OR-nodes that are adjacent to and_node, not yet in
        L and not in done_or to L

return the solver configuration
```

Validating a solver configuration (TODO: issues with optional parameters):

```
Input: Parameter Graph, Solver configuration; Output: Boolean
assigned_and_nodes := Set of AND-nodes with parameters that are
                        part of the solver config

Test if all required OR-node parameters are present
 i.e. of OR-nodes that have at least one incoming edge
 group from assigned AND nodes

or_nodes := Set of preceding OR-nodes of assigned_and_nodes
for each or_node in or_nodes:
    - Test if or_node has exactly one adjacent AND-node in
      assigned_and_nodes
    - Test if at least one group of edges comes from
      assigned AND-nodes

Test if all OR-nodes adjacent to start node are in or_nodes

return True if all tests were passed
```

Open problems:

- Discretise real valued parameters. Especially the discretisation of domains divided into several domains (several AND-Nodes) is unclear.
- find an algorithm to iterate over the neighbourhood of a given solver configuration (with discretised parameters)
- crossover and mutation operators for genetic algorithms

5.3 Example

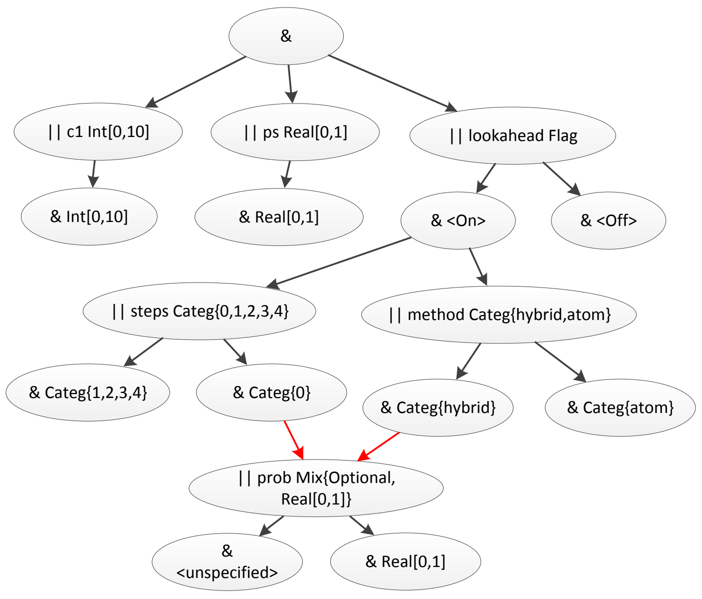
Example 4: Consider a solver that has the following parameters:

- *c1* which takes on integer values in $[1, 10]$.
- *ps* which takes on real values in $[0, 1]$.
- A flag called *lookahead* which can be present or not.
- A categorical parameter *steps* which takes on values in $\{0, 1, 2, 3, 4\}$.
- Another categorical parameter *method* whose value is either "hybrid" or "atom".
- A parameter *prob* which can be left out or take on real values in $[0, 1]$.

Furthermore there are some restrictions and requirements:

- Both *c1* and *ps* have to be always specified.
- If the *lookahead* flag is present, both *steps* and *method* have to be present.
- If *steps* takes on the value 0 and *method* takes on the value "hybrid", then the parameter *prob* can take on values in its real domain $[0, 1]$ or be left out.

This parameter space can be encoded in a parameter graph as defined earlier in the following way:



The two red edges imply the membership of the edges to the same edge group $\neq 0$. Black edges mean that the edge doesn't belong to any group. For simplicity, the parameter references of AND-Nodes (to the same parameter as the preceeding OR-Node) are not shown in the graph.

5.4 Implementation etc.

- define parameter graphs in XML / GUI
- API documentation for configurators

6 Client

6.1 Introduction

The computation is client is used to compute the jobs of experiments. Usually there have to be a lot of jobs computed to evaluate experiments and since they are independent from each other, this task can be parallelized across many CPU cores. The computation client can be started on arbitrarily many machines and will manage the available CPUs and fetch and jobs from the created experiments. It connects to the central database and downloads all required resources such as instances and solver binaries and writes back the results to the database.

6.2 System requirements

The client is written in C/C++ and should be able to run on most Linux distributions where a MySQL C connector library is available.

TCP/IP Connections: Because the central storage location for all required computational resources, experiment metadata and results is a MySQL database, the client has to be able to establish a connection to the machine that hosts the database. This means that the machines where the client runs on have to be able to establish a TCP/IP connection to the database machine.

Connection alternatives: The client was mainly tested on the bwGRID[1], a distributed computer cluster that consists of several hundred nodes at several physical locations at universities of Baden-Württemberg, Germany. Even though the machine hardware is homogenous, the network topology of bwGRID is not. In cases where direct network access from the computation nodes back to the database server is not possible it is usually possible to tunnel a connection over the cluster's login node back to the database via SSH.

Other than that, the client has to be able to write temporary files to some location on the filesystem. This can be configured (6.3.1) if it differs from the client binary location.

Shared filesystems: Because the client will download missing solver binaries and instances and upload results it also needs a reasonably fast network connection to the database. Shared filesystems can considerably reduce the required bandwidth since every file is only downloaded once. As alternative you can pre-package the required files and put them into the correct folders before starting the client. However, if you modify experiments while the client is running it will still download missing files.

[1] http://www.bw-grid.de/

6.3 Usage

6.3.1 Configuration

Configuration file: Configuration is done by some command line arguments and a simple configuration file, called "config". This file has to be in the **working directory** of the client at runtime. In the configuration file you have to specify the database connection details and which hardware the client runs on. This is done by configuring so called "grid queues" in the GUI application. They contain some basic information about the computation hardware such as number of CPUs per machine. The client will then use this information to run as many parallel jobs as the grid queue information allows it on each machine where it is launched. Here is a sample configuration file:

Example 5:

```
host = database.host.foo.com
port = 3306
username = dbusername
password = dbpassword
database = dbname
gridqueue = 3
verifier = ./verifiers/SAT
```

Note that the gridqueue value is simply the ID of the grid queue. Another (optional) configuration option is the verifier line. It tells the client if it should run a program on the output that a solver generated. For example, if your experiments consist of attempting to solve boolean propositional logic formulas you can use a SAT verifier that tests if the solution given by a solver is actually correct. The verifier configuration option is simply a path to an executable that the client calls with standardized parameters. See section 6.4 for more information on verifiers.

Command line arguments: Beside the configuration file there are several command line options the client accepts, please also see "./client –help":

```
-v <verbosity>:
  Integer value between 0 and 4 (from lowest to
  highest verbosity)
-l:
  If flag is set, the log output is written to a file
  instead of stdout.
-w <wait for jobs time (s)>:
  How long the client should wait for jobs after
  it didn't get any new jobs before exiting.
-i <handle workers interval ms>:
  How long the client should wait after handling
  workers and before looking for a new job.
-k:
  Whether to keep the solver and watcher output files after
  uploading to the DB. Default behaviour is to delete them.
-b <path>:
  Base path for creating temporary directories and files.
-h:
  Toggles whether the client should continue to run even
  though the CPU hardware of the grid queue is not homogenous.
```

Verbosity controls the amount of log output the client generates. A value of 4 is only useful for debugging purposes, a value of 0 will make the client log important messages and all errors.

If the ”l” flag is set, log output goes to a file whose name includes the hostname and IP address of the machine the client runs on to avoid name clashes in shared filesystems typically found in computer clusters. Otherwise log output goes to standard output.

With the ”-w” option you can tell the client how long to wait before exiting after it didn't start any jobs. This can be useful to keep the clients running and ready to process new jobs while you evaluate preliminary results and add new jobs or whole experiments. The wait option is also used to determine how long attempts should be made to reconnect to the database after connection losses. The default value is 10 seconds.

The ”-i” option controls how long the client should wait between its main processing loop iterations. If this value is low, it will more often look for new jobs when there are unused CPUs. For maximum job throughput this value should be lower than the average job processing time but lower values will also put more strain on the database and increase the client's CPU usage. The default value is 100ms which should work fine in most cases. The client will also adapt to situations where there are free CPUs but no more jobs and increase the interval internally and fall back to the configured value once it got another job.

The ”-k” flag tells the client if it should keep temporary job output files after a job is finished of if it should delete them. The default is to delete them.

The ”-b” base path option can be used to specify a directory the client can use to write temporary files to. The default value is ”.”, i.e. the working directory at runtime.

! → The first client to start with a particular grid queue will write the information about the machine it runs on to the grid queue entry in the database. All following clients will then compare their machines to the information in the grid queue and exit, unless the number of cores and the CPU model name match. With the ”-h” option you can override this behaviour.

6.3.2 Launching

After configuration you can simply run the client on your computation machines. On computer clusters there are often queuing systems that you have to use to gain access to the nodes. On bwGRID for example, we could use the following short PBS (portable batch script) and submit (*qsub scriptname*) our client to a node with 8 cores:

```
#!/bin/sh
#PBS -l walltime=10:00:00
#PBS -l nodes=1:ppn=8
cd /path/to/shared/fs/with/client/executable
./client -v0 -l -i200 -w120
```

! → You should always run the client from within its directory (i.e. cd to the directory) to avoid problems with relative paths such as the verifier path from the example configuration above.

As soon as clients start you should be able to see jobs changing their status from ”not started” to ”running” in the GUI's or Web frontend's job browsers.

6.3.3 Troubleshooting

If errors or failures occur the client will always attempt to shut down cleanly, that is stop all running jobs and set their status to "client crashed" and write the last lines of its log output as "launcher output" to each job. This can fail when network connections fail or the client receives a SIGKILL signal causing it to exit immediately. In case of network failures you should still be able to find useful information in the client's logfile on the local filesystem.

6.4 Verifiers

Verifiers are programs that the client runs after a job finishes. Verifiers are getting passed the instance of the job and the solver output as arguments and are supposed to exit with a status code that conveys some information about the result of the job. For example whether the output is correct given the problem instance. This code will be written to the database as "result code" and "verifier exit code". Any output the verifier writes to standard out will be written as "verifier output". The call specification for a verifier binary looks like this:

```
./verifier_binary <path_to_instance> <path_to_solver_output>
```

We provide a verifier for the SAT problem that works on CNF instances in DIMACS format and solvers that adhere to a certain output format (see the source code). **!** $\rightarrow$ Note that you have to make sure that your possible result codes are specified in the *ResultCodes* table in the database before running clients or there will be errors when the client tries to write results. By convention, the web frontend and GUI application consider status codes that begin with a decimal "1" as correct answers.

6.5 Experiment priorization

Sometimes it can be useful to compute several experiments in parallel but give some a higher priority than others. In order to accomplish that, experiments can be marked as inactive and individual jobs can be prioritized. Only jobs of active experiments with priority equal to or greater than 0 are considered for processing by the client. Futhermore, experiments can be assigned a priority. The clients will then try to match the relative number of CPUs working on an experiment with its relative priority to all other experiments that are assigned to the same grid queue. For example, if you have three experiments with priorities 100, 200 and 300 respectively the running clients will try to have 16% of CPUs working on the first, 33% of CPUs working on the second and 50% of CPUs working on the third experiment.

! → The client is running solely on Unix and is not distributed yet for Windows systems.

7 Web Frontend

8 Monitor

9 Troubleshooting

Index