Edalize it. Don't critizise it

Olof Kindgren

***Abstract*—Edalize (https://github.com/olofk/edalize) is an abstraction library for interfacing different EDA tools with a single description of the input files and tool configuration.**

***Index Terms*—Edalize, FuseSoC**

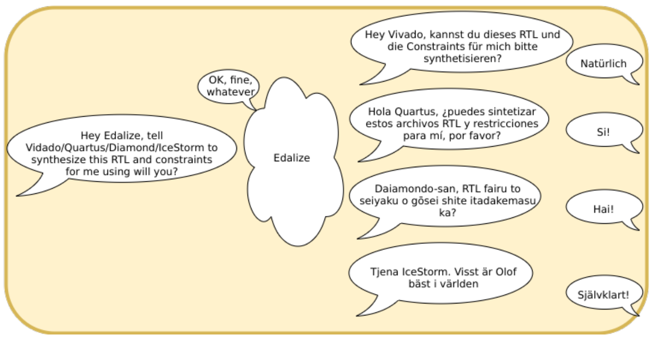
I. Introduction

A language like C has multiple compilers such as GCC and LLVM. Switching between them is easy peasy because the two tools mostly speak pretty much the same language.

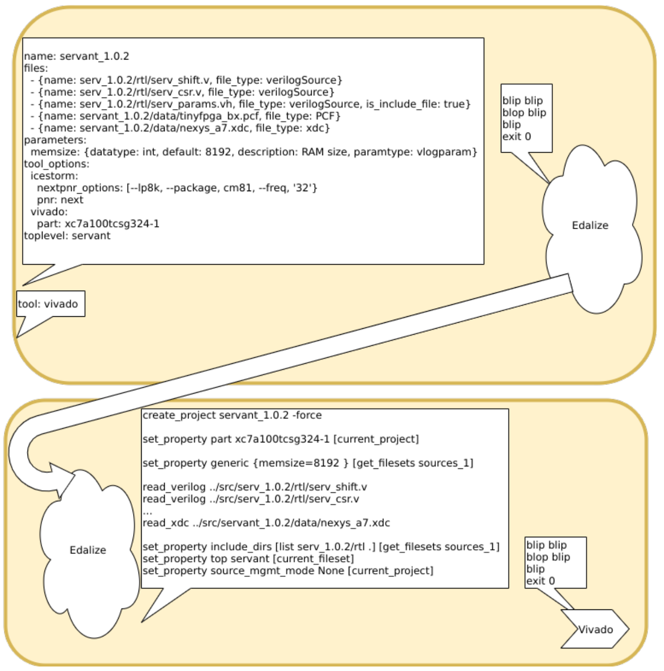
The same thing however can not be said about different EDA tools even though many of them really accomplish the same thing

Hey GCC, compile these files for me will you?
Will do!
Hey LLVM, compile these files for me will you?
Sure thing!
Hey Vivado, kannst du dieses RTL und die Constraints für mich bitte synthetisieren?
Natürlich
Hola Quartus, ¿puedes sintetizar estos archivos RTL y restricciones para mí, por favor?
Si!
Daiamondo-san, RTL fairu to seiyaku o gōsei shite itadakemasu ka?
Hai!
Tjena IceStorm. Visst är Olof bäst i världen
Självklart!

And this is where Edalize comes in, as a Rosetta stone for EDA tools.


Hey Edalize, tell Vidado/Quartus/Diamond/IceStorm to synthesize this RTL and constraints for me using will you?
OK, fine, whatever
Edalize
Hey Vivado, kannst du dieses RTL und die Constraints für mich bitte synthetisieren?
Natürlich
Hola Quartus, ¿puedes sintetizar estos archivos RTL y restricciones para mí, por favor?
Si!
Daiamondo-san, RTL fairu to seiyaku o gōsei shite itadakemasu ka?
Hai!
Tjena IceStorm. Visst är Olof bäst i världen
Självklart!

In practice though, the communication will look more like this



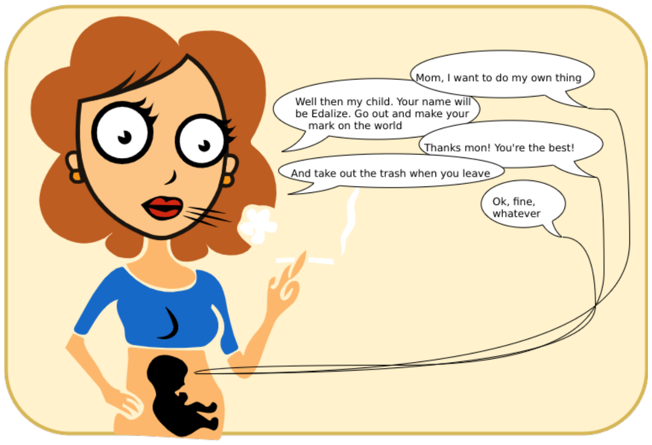
```
name: servant_1.0.2
files:
  - {name: serv_1.0.2/rtl/serv_shift.v, file_type: verilogSource}
  - {name: serv_1.0.2/rtl/serv_csr.v, file_type: verilogSource}
  - {name: serv_1.0.2/rtl/serv_params.vh, file_type: verilogSource, is_include_file: true}
  - {name: servant_1.0.2/data/tinyfpga_bx.pcf, file_type: PCF}
  - {name: servant_1.0.2/data/nexys_a7.xdc, file_type: xdc}
parameters:
  memsize: {datatype: int, default: 8192, description: RAM size, paramtype: vlogparam}
tool_options:
  icestorm:
    nextpnr_options: [--lp8k, --package, cm81, --freq, '32'}
    pnr: next
  vivado:
    part: xc7a100tcsg324-1
toplevel: servant
```

```
tool: vivado
```

```
create_project servant_1.0.2 -force

set_property part xc7a100tcsg324-1 [current_project]

set_property generic {memsize=8192 } [get_filesets sources_1]

read_verilog ../src/serv_1.0.2/rtl/serv_shift.v
read_verilog ../src/serv_1.0.2/rtl/serv_csr.v
...
read_xdc ../src/servant_1.0.2/data/nexys_a7.xdc

set_property include_dirs [list serv_1.0.2/rtl .] [get_filesets sources_1]
set_property top servant [current_fileset]
set_property source_mgmt_mode None [current_project]
```

II. History

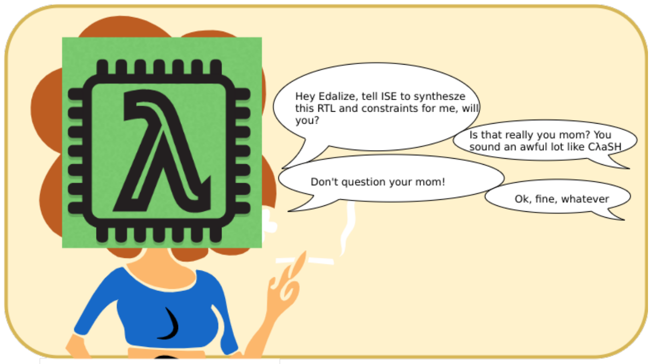
Edalize was born inside of FuseSoC, but over time it became clear that the part of FuseSoC that was interfacing EDA tools could be useful as a standalone project, and thus Edalize was born


Mom, I want to do my own thing
Well then my child. Your name will be Edalize. Go out and make your mark on the world
Thanks mon! You're the best!
And take out the trash when you leave
Ok, fine, whatever

Now that Edalize was its own entity, it needed a way to talk its companion FuseSoC

Mom, I know how to talk to a lot of EDA tools, but how should I talk to you?
We will invent a new format for describing all files and options that the EDA tools need. We shall call it EDA Metadata, or EDAM for short
Cool! Let's go with that
Also, because it's mostly a fancy wrapper, like Edam cheese
Moooooom!!!

With the EDAM format, there was a standardized way to communicate between FuseSoC and Edalize. But having a standard also means that either part can communicate with any other project that also implement this standard. And this is really why Edalize and EDAM was created, to allow other projects to reuse either part as they see fit.


Hey Edalize, tell ISE to synthesze this RTL and constraints for me, will you?
Is that really you mom? You sound an awful lot like CλaSH
Don't question your mom!
Ok, fine, whatever

And it seems to work. Edalize has been picked up by several projects that need to interface EDA tools, rather than implementing all this by themselves, and the list keeps growing.

III. Overview

As of writing, Edalize supports 21 different EDA tools with a handful other in the works. This includes simulators, FPGA synthesis tools, linters and soon ASIC synthesis and formal verificaion.

The EDAM API is documented at https://edalize.readthedocs.io/en/latest/edam/api.html so there's no point in going through that here. But a few words about the steps that Edalize takes after being passed an EDAM description.

Edalize can be asked to run through all these steps, or stop after having completed one of them. The following subsections will go through their purpose and the reason for stopping after each step.

A. *Configure*

The first step that Edalize takes is called setup or configure. This converts the EDAM description into the native format of the requested EDA tool and creates a script or Makefile that can launch the whole process with a single command. At this point, the EDA tool is not actually launched. This means that the configure step can be done without having access to the EDA tool. Doing the configure step also unshackles the project from Edalize and any subsequent step is designed to be usable without Edalize. This is useful to create project files that can be sent together with the source to other who don't use Edalize, or for archival purposes. It can also be used by those who prefer to create an initial setup with Edalize and continue using the GUI of their EDA tool of choice from then on.

B. *build*

The build step does the heavy lifting of running the EDA tool to produce some sort of binary image. For simulators, this creates the compiled simulation model. For FPGA toolchains this creates the FPGA image to be programmed to the FPGA. For linting tools, this performs the actual linting.

For FPGA toolchain the process can be stopped here if the FPGA image is the requested result. For simulators, it makes sense to stop here if the same compiled simulation model is to be executed with a set of different parameters in the following run stage.

C. *run*

Running means different things for different tools. For simulators it means running the compiled simulation model, and Edalize allows setting run-time parameters to run the simulation with different parameters without recompiling. Common uses of this is to run an integrated CPU with different software.

For FPGA toolchains, running means programming the FPGA image to the board. This has turned out to not be very intuitive, so it might change in the future.

IV. Future work

Today, Edalize is based on backends or tools, with some confusion regarding the terminology. Some backends can be set up to work in different modes (e.g. verilator used in C++, SystemC and lint modes). Other backends are built up from parts that can be switched out to equivalent functionality (e.g. icestorm which can use ArachnePnR or NextPNR, or several other FPGA-vendor-powered backends that can potentially use yosys for synthesis). This works mostly fine for the existing flows but there are several cases where this is not ideal.

1) Netlist simulations: This typically requires running most of an FPGA (or ASIC) flow, export the result to RTL and then run a simulator. This is not easy to achieve today without duplicating functionality.

2) VUnit simulations: The VUnit support sets the stage for VUnit and uses the VUnit tool interfacing code instead of the support in Edalize. The situation for Cocotb is simuilar.

So for these reasons, we are looking at a revamped Edalize called

Edalize, slight return

A. Implementation

Instead of tool-specific backends, ESR will be based around flows which are graphs built up from stages (nodes)

Each stage will pass files and an EDAM file between them. By having all stages input and output EDAM files we can easily combine them in different ways.

Edalize will still consist of three major stages, configure (setup?), build and run. configure creates all project files and creates the graph. Build executes all the build stages. The run stage is for final tasks that make sense to run several times even though though the source files are unchanged. E.g. running several simulations runs without rebuilding the simulation model, or reprogram an FPGA multiple times with the same FPGA image. As a counter-example, e.g. synthesis does not fit into this category as it should produce identical result given the same input.

At configure time, the flow graph is set up and the flow configuration and tool settings are applied to each stage to generate the necessary tool project/config files. A ninja (tbd) file is created to describe the dependencies between the stages.

The build phase should only execute ninja (or similar) to run the graph to completion

The run phase needs to be better defined. It has runtime parameters but might need something else too.