RoadRunner: A Modularized Design and Tool Manager for Hardware Projects

Mattis Hasler
Barkhausen Institut
Dresden, Germany
mattis.hasler@barkhauseninstitut.org

Sebastian Haas
Barkhausen Institut
Dresden, Germany
sebastian.haas@barkhauseninstitut.org

***Abstract*—In hardware design, the tool pipeline's abstraction level is generally not very high. A design is usually built for one setup, meaning one specific tool flow. Adding support for other tools is a manual task that must be repeated for each tool. That makes using external IP in a project cumbersome because, in most cases, importing an IP also means adjusting it to the project's tool set. RoadRunner is a versatile design and tool flow manager emphasizing the separation of concerns in multiple aspects of hardware design projects. The definition of hardware modules and their attributes—like clock definitions—is done hierarchically to reflect the hardware design and allow easier reuse of modules compared to centralizing attributes in project files. Additionally, the hierarchical module definition is tool-agnostic, allowing the separation of hardware design from the tools used, further easing the reuse of modules. In RoadRunner, the processing steps are separated into distinct working directories to prevent unpredictable side effects and increase portability with workload managers. Portability is further improved by separating the "what" to do with a tool from the "how" to use a tool. Where the project should define what to do, and the IT infrastructure should define how to set an environment to make a tool usable. The flexibility to compose and work with inhomogeneous projects is shown in an multi-processor system-on-chip design that includes an open source RISC-V core and an AI accelerator and targets an FPGA, an ASIC implementation, and a fully open-source simulation environment simultaneously.**

I. Introduction

In hardware design, it is worthwhile to wrap the execution of tools behind a simple interface. Such a project setup is usually particular to the environment and one specific tool flow. It enables engineers to work on big designs without knowing every detail about the modules and tools used or the integration of each (third-party) component. However, changing something in the tool flow requires manual work and a lot of tool expertise. Also, intellectual property (IP) developed on another tool flow needs adoption to be included because design and tool flow configuration are often intertwined. Generally, the plain HDL description of a design must be enriched with metadata to be smoothly fed through a processing pipeline. For example, asynchronous registers must be tagged in a design for synthesis to run smoothly, which is impossible in Verilog directly. Available solutions to enrich a hardware design with metadata are defined mainly by tool vendors and are tool-specific and lack the flexibility to describe truly hierarchical designs.

Hardware design projects are huge, confusing, and comprised of many tasks, making debugging the project hard. With the increasing complexity of designs, the number of 3rd-party IPs increases, which results in an increasing number of used languages and tools. Every tool needs its specific environment, produces a unique output structure, and may clutter its working directory with logs and auxiliary files. Cluttering in project directories makes it unclear to the untrained eye which task —or tasks—a file belongs to. In addition, infrastructure has become more decentralized, making it necessary that projects work on different machines.

This work introduces RoadRunner, an EDA project management tool that separates different aspects of a hardware design project to address the problems that traditional project structures face. It allows hierarchical definition of designs, not only at the HDL but also at the metadata level. The goals of RoadRunner are to provide a human-readable and debuggable working environment that separates design from tool flow description and makes both reusable and interoperable. Tool and source language support is modularized to allow design definitions to be reused with a different tool set. The general working environment for an EDA project is shown in Fig. 1. It comprises the following key components:

1) The **RoadRunner Config Space** holds the project definition. It is used for design and tool flow definition and capturing the results of processing steps, making them available for follow-up steps. It features hierarchical definitions, sub-project inclusion, cross-references, and parametrizable dynamic content. The config space is rendered from special YAML files in both the project's directory tree and result directories created by already executed processing steps.
2) **Isolated Working Directories** are created when running a processing step (aka. command) defined in the config space. It contains all the files needed to execute the selected command. These may be source files, results, generated auxiliary files, and scripts controlling the execution. All input and output stays within, making a working directory autonomous; it stays executable even when copied to a remote location.
3) **Results** are directories that capture the outputs of processing steps that are to be persistent and could be reused in later processing steps. A result directory contains a config space definition describing the result's structure, which is included in the config space.
4) A **Machine Config** is independent of projects. It holds information on how to access third-party tools (TPTs) on the current machine. While the project defines *what*

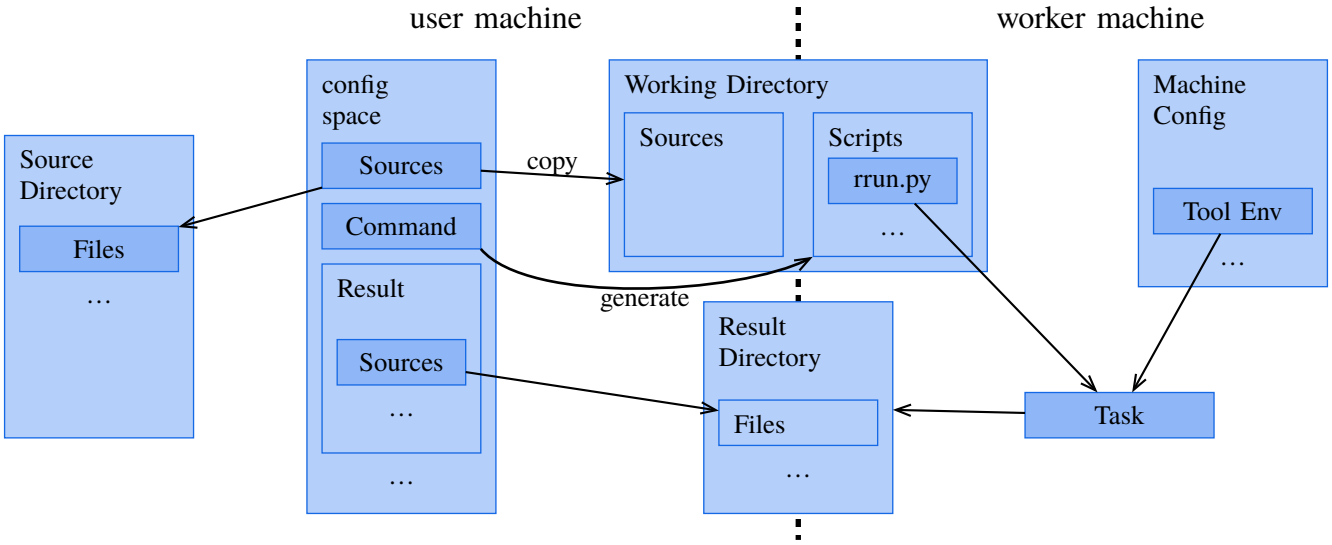

Fig. 1: Parts of a RoadRunner project. The config space acts as a central storage of structural information. A working directory contains all files for a processing step. Files can be sourced from a local directory tree and results of previous processing step. The machine config contains info to make a task runnable on a (worker) machine.

to do with a specific TPT, it does not have to deal with *how* to call it, which is completely left to the IT administration of each machine.

5) A **Task** is the execution of one processing step in an EDA tool flow. The Task is defined in the config space by a *command node*. From the command definition, the working directory will be created in the preparation phase. In the execution phase, which may happen on a different machine, the entry point of the working directory is called. The *machine config* is used to get access to the TPT to be used. Generated results that are placed into the result directory are automatically included in the config space after the Task is finished.

II. Related Work

As of now, there is a multitude of hardware project management tools. From our experience, at least two big companies that do digital design have in-house solutions, which can be described as sophisticated tool-flow-specific shell scripts. One problem these tools have is that they were developed on a specific infrastructure and thus have particular and deep-rooted requirements, like a specific version control system (like Subversion) or software management tooling (like environment modules). Similarly, open-source projects often use in-house developed make-based build systems like the *Rocket Chip* processor generator [1] that do not expect any software management. Instead, it assumes that all needed software is available through an installation process that reportedly takes half a day to complete.

There are open-source tools that manage different IP sources and TPTs. Most of these tools focus on a specific use case, a set of tools, or a project structure. *hdlmake* [2] describes itself as a Makefile generator. It features resolving module dependencies by fetching repositories and generates Makefiles to run simulations and synthesis with tools from several vendors, including Xilinx and Mentor. As a bonus, it allows the remote execution of tasks. However, it only works with FPGA-targeting projects.

Although *OpenPiton* [3] is foremost described as a general-purpose, many-core processor, it is also a framework that includes creating hardware, firmware, and software within the landscape of the Piton processor. It provides a polished interface to tinker with every part of the processor and run every step, like simulation, FPGA synthesis, and ASIC backend steps. As the project grows around the Piton Processor, the possibility of using it for other projects is minimal. A more general approach is pursued in the *Silicon Compiler* [4] project. It provides a Python-based API to describe hardware projects, targeting an automated source-to-hardware flow.

Another project that allows the generation of SoC structures based on a set of parameters is *PLSI* [5]. It structures the creation of a SoC in two steps: the generation of "cores" and the composition of these "cores" into a SoC. Apart from the available generators for Rocket Cores and BOOM Cores, it is possible to create more generators for other cores. It is possible to run simulations of the SoC in Verilator and Synopsys VCS. Similarly, synthesis is limited to Synopsys DC and formal verification to Synopsys formality.

Successor to PLSI, *HAMMER* [6], [7] defines a versatile system to mix and match tools and technology nodes while separating concerns by putting each type in a different set of plugins. Still, plugins "communicate" through a standard API to make them interoperable and project-independent. *HAMMER* also defines an IR that uses metaprogramming and hooks to allow the user to adjust a tool flow to their needs. The tool-flow customization possibilities allow arbitrarily complex flow definitions that, in our humble opinion, may become too complex to be adoptable by other projects. Apart from the flow definition, *HAMMER* lacks flexibility regarding the composition of designs from different sources and source languages. Similarly, the OpenRoad [8] project focuses on the automation of the

backend process. It supports various tools and a growing set of PDKs to support an open-source backend tool flow.

On the contrary, *FuseSoc* [9] focuses more on the composition of a design by allowing the definition of a dependency tree that can automatically be resolved. Design definitions, including dependencies, are defined in a fixed YAML scheme. *FuseSoc* has, unlike HAMMER, a fixed list of tools that can be utilized to do simulations or synthesis for FPGAs.

Another tool to specify and automatically resolve design dependencies is *Bender*. Although not published separately, *Bender* definitions are widely available in the pulp ecosystem [10] and thus well known in pulp-related projects. It uses a YAML scheme to define modules with their source files and dependencies, usually as git repositories. *Bender* can automatically resolve the dependencies of whole projects, fetch the needed files, and then call some hardcoded software or export the derived file lists in various formats for further usage by other tools.

RoadRunner combines the ideas of *HAMMER* to modularize the tool flow and the ideas of *FuseSoc* and *Bender* to compose hierarchically defined designs.

Central to the RoadRunner project is the config space, which uses YAML to describe a project's structure.

The IEEE standard format IP-XACT also offers the description of hardware IP together with metadata, and is used by many commercial vendors. However, IP-XACT focuses on describing hardware modules, including a very detailed connectivity definition [11]. In contrast, RoadRunner also allows the description of modules but assesses the integration of 3rd-party tools into the project as equally important.

RoadRunner draws inspiration for its execution environment isolation from Bazel[12]. Following the isolation idea consequently leads to a working directory becoming self-contained. At some point, even the used tools are part of the working directory. RoadRunner breaks the isolation at this point because many EDA tools are too big to be copied around like that. Additionally, in contrast to Bazel, RoadRunner puts effort into making the isolated environment as clear and human-readable as possible and maintaining the possibility of entering an environment and rerunning steps manually.

III. Config Space

The config space provides core functionality in RoadRunner to provide flexible and hierarchical definitions of modules and their sources, attributes, metadata, and transformation steps in the form of TPT invocations to produce result data that, in turn, gets reinserted into the config space. In Fig. 2, a config space is shown. Conceptually, there are three types of nodes: All nodes that contain information about the project source files or metadata are "resource description nodes". "Transformation steps" take a node (or a whole sub-tree) as input and insert a new node into the config space. Finally, "composition" nodes describe the relation of nodes, e.g. lists, attribute sets, or cross-links.

The config space of a project is loaded from a YAML file named `RR`, situated in the project's root directory. The standard attribute-set-and-list structure provided by YAML is

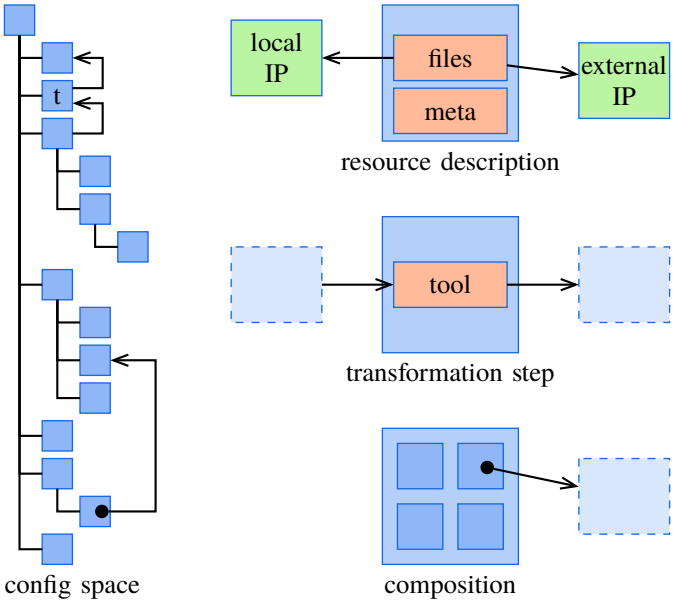

Fig. 2: Structure of the config space. It is a tree structure to allow hierachrical definitions resembling the usual hardware design structure. Each node in the config space can define resource by linking source files or define additional meta data, describe a processing step aka. transformation step, or describe a composition of a set of sub-nodes.

extended with additional functionality. Among others, there is the possibility to cross-reference within the config space, load sub-trees from additional YAML files, define variables, walk the config space recursively to gather attributes hierarchically, and render content dynamically using an inline Lua interpreter and dynamic attribute sets. When containing an `RR` file, subdirectories can be included in the config space as a subtree, allowing easy building of a config space topology that resembles the directory structure. With dynamic attribute sets, variants of the same config space sub-tree can be generated based on context variables. For example, an attribute that defines the optimization level might depend on the current use case.

```
opt: # optimization level
  /SYNTHESIS: 5   # full ~ in synthesis
  /DEBUG:     0   # no ~ in debug
  /default:   2   # some ~ otherwise
```

In this case, the given attribute set would render to `opt: 2` given there are no flags set. With flags set, the effective value of the attribute can change.

IV. Modules

With the config space, it is possible to describe all sorts of metadata in a structured, hierarchical, and modular way. However, RoadRunner does not define a fixed scheme to allow or disallow specific attribute names. Modules will traverse the config space and read out the needed attributes. The set of attributes a module reads out depends on the domain to which it is dedicated. A domain might be an HDL language or a TPT. For example, the "Verilog" language module provides functions to gather the files needed to compile a hardware module. While doing so, it will read the "sv" and "v" attributes to gather

source files and walk the config space recursively by following "include" attributes.

As an example of a tool module, the "Icarus" module provides functionality to compile and run simulations using the open-source Icarus Verilog Simulator [13]. It will read out attributes to define the parameters needed to be passed to the simulator binaries, such as the "toplevel" attribute, to specify the simulated hardware module. It also utilizes the "Verilog" module to gather source files needed for the simulation. The module exports a command handle that allows the simulator to be used from a transformation node in the config space.

V. Commands

To use a tool in a RoadRunner project, a transformation (i.e. command) node must be defined in the config space. A simple command node for an Icarus-Verilog-based simulation could be defined as follows:

```
simulation:
  tool: Icarus
  sv: testbench.sv
  include: =:module1          #cross-reference
  toplevel: testbench
```

A command node refers to a command handle defined by a tool using the "tool" attribute. For example, `tool: Icarus` calls the standard command handler in the "Icarus" module to process this transformation node. The command handler will then read all other attributes.

A command handler is implemented as a Python function. It receives a config space context—a pointer to a config space node—pointing to the command node as an argument, and will prepare the execution of a tool. RoadRunner prepares a dedicated working directory for each command invocation. The command handler copies all needed source files to the working directory and generates additional scripts to perform the requested TPT execution. To make this process as easy as possible, RoadRunner provides convenience functions to help with these tasks.

A. Isolated Execution

A command handler is supplied with a working directory by RoadRunner. The handler's task is to provide all the resources needed for the command and copy them to this directory. That usually includes source files and additionally generated scripts. Additionally, it uses the "Pipeline" and "Call" classes to define tool calls and turn them into a set of scripts in the working directory that can be started by an entry script (Fig. 1). This way, the working directory becomes self-contained. It no longer depends on the project directory and can be moved to various environments for execution, such as a different machine. Another effect is that the working directory is isolated from any other processing step defined in the project, as well as the potential side effects it might have. Only an explicitly defined set of files will be copied to the result directory and brought back into the project's config space for other processing steps to be used.

B. Execution Time Environment

The execution phase of a command is controlled by a subsystem of RoadRunner called RoadExec, which is copied to each working directory. It starts by calling the entry point script `rrun.py` at the root of the working directory. As shown in Fig. 1, RoadExec will load the tool environment from the machine configuration. The tool environment contains information on how to set up an environment to run a specific tool. It can vastly differ from machine to machine, and it should be the concern of the administrator of each machine to provide this information. The project, in contrast, should not be concerned with this to stay agnostic to the IT infrastructure in which it is installed. Apart from setting up the environment for the used tools, RoadExec controls the order of configured invocations. It starts the tools according to the structure defined by the command handler when creating the working directory. There can be a mixture of sequential and parallel command groups.

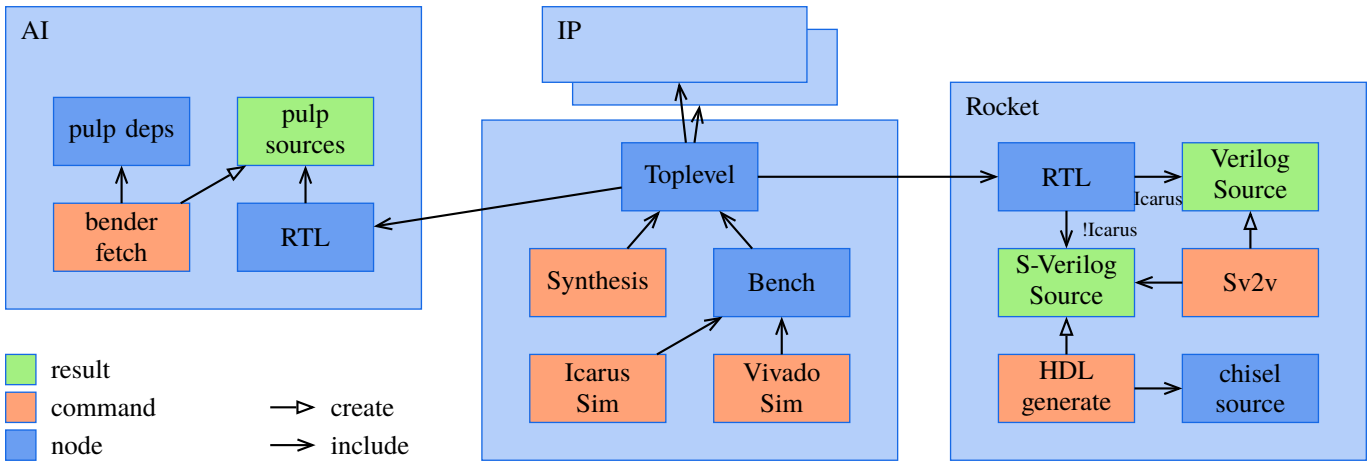

Fig. 3: Project hierarchy of the MPSoC as defined in the config files (RR). Command nodes (orange) can be executed to execute a 3rd party program and may create a result node (green). Resources for command execution are gathered through an include hierarchy, that may span across the config and can include any kind of node including result nodes.

VI. Example

In an extended example, the flexibility of RoadRunner is to be presented. The considered example design is an MPSoC. It features multiple RISC-V cores connected with a network-on-chip and additional custom top-level modules. While most top-level modules are custom modules written in Verilog, some need special handling. The RISC-Vs are Rocket Cores [1] written in Chisel [14] depending on Chipyard's make-based RTL-export system. The MPSoC features an AI accelerator that heavily uses modules from the pulp ecosystem, which Bender manages. The design is built to be synthesized to an Xilinx FPGA evaluation board and a Cadence toolchain working on a 22nm FDSOI PDK from GlobalFoundries. Simulating the design is possible on commercial simulators of Cadence, Synopsys, and Xilinx, and for early development on the open-source Icarus Verilog Simulator [13]. Tapeout is done on the premises of an external foundry access provider, which has its own custom project and tooling management. For the tapeout version, a DDR Interface IP from Synopsys is included, as well as a PLL IP and multiple pad cell IPs provided by the foundry access company. The tapeout-specific IP is left out of the design for other use cases by exploiting RoadRunner's dynamic content mechanics.

An overview of how the project is structured in RoadRunner is given in Fig. 3. Each IP is held in its subdirectory and is defined in a separate RR file, depicted in the figure as big boxes. The top-level definition uses cross-references to include RTL from the IPs. It also defines the simulation commands close to the corresponding RTL definition, i.e. the top-level simulations in the top-level node and IP level simulations in the IP nodes. The AI accelerator and the Rocket Core IPs define special commands to deal with special origins and provide smooth integration into the RoadRunner project.

A. External Resources

The rocket core IP and the AI accelerator are not developed in-house and do not use RoadRunner. Still, a RoadRunner config file has been created to include these external resources in the RoadRunner project. For the AI accelerator IP, two static resource nodes are defined. As shown in Fig. 3, these are the "RTL"-node, which only lists the (System)Verilog sources, and the "pulp_deps"-node to define the dependencies into the pulp ecosystem in a Bender config file. In the usual Bender workflow, these dependencies are downloaded from git repositories as given in the Bender config file. The retrieved dependencies are placed somewhere in the project directory to be used in the following steps, such as simulation or synthesis. In the example project, RoadRunner's Bender module is used to import dependencies specified in a Bender file:

```
aiAccel:                          #(Bender Fetch)
  tool: Bender
  benderDir: aiAccel    #dir with bender-file
```

This command loads the Bender-defined dependencies from the external IP included in the project as a git submodule.

The Rocket Core IP is also included as a git submodule. The project defines two commands on this asset (in Fig. 3 named "chisel_source"). The first command transforms the source code given in Chisel to SystemVerilog. The second command is only needed for the Icarus-based simulation flow, as it compiles the SystemVerilog ("S-Verilog Source") to Verilog ("Verilog Source"). The correct version is selected automatically using RoadRunner flags. When the "RTL" node is included and the "Icarus" flag is set, the Verilog variant is used, and the SystemVerilog version is used otherwise.

```
clock_generator:
  /XILINX:                #FPGA implementation
    sv: xilinx/ClockGen.sv
    xilinxLib: mmcc0          #needed library
  /TAPEOUT:
    sv: ip/ClockGen.sv
    include: =:ip.adpll.rtl     #using PLL IP
  /default:                   #only simulation
    sv: sim/ClockGen.sv
```

Listing 1: Dynamic clock generator selection depending on used Software and target. Three variants all define the `ClockGen` module, internally using different implementations based on different IP.

B. Simulation

The top-level "RTL" node includes different IPs throughout the project, including the AI accelerator, the Rocket Core, and multiple others (Fig. 3). It also defines command nodes for simulating the design using the Icarus Simulator and Xilinx Vivado. Both commands include the exact top-level RTL definition, even though it will render slightly different designs. As described before, depending on the simulator type, the rocket core definition will include either the Verilog or the SystemVerilog sources of the processor. Similarly, the clock generation will differ depending on the simulator, using Xilinx's clock IP, a non-synthesizable open-source version, or a purchased third-party IP for ASIC synthesis.

Pushing the metadata and implementation derivation selection to the units makes the actual simulation command almost trivial, containing no special information apart from the to-be-simulated top-level module.

```
simpleSim:                          #(Icarus Sim)
  tool: Icarus                      #using Icarus
  include: =;Bench      #cross-ref bench node
  toplevel: SimpleBench
```

The selection of the correct variants is controlled entirely by the flags the Icarus command handler defines, e.g. `SIMULATION` and `ICARUS`.

C. Implementation Results

RoadRunner has been used to tape out three chips, named Masur23 [15], Masur24 and Masur25. The latter two are currently being measured in the lab. All three chips share a project structure similar to the example project from this work. RoadRunner itself is open source and can be obtained from our GitHub repository[1] or via Pypi.org[2]

D. Synthesis

The example design can be synthesized for FPGAs using Vivado. As shown in Fig. 3, the synthesis command (Synthesis)

[1]https://github.com/Barkhausen-Institut/roadrunner
[2]https://pypi.org/project/roadrunnerEDA

uses the same top-level definition as the simulation test bench. Specific differences between the implementation for simulation and synthesis are resolved by the tool wrapper, applying different flags to the configuration. For example, as shown in the code example in Listing 1, the clock implementation differs depending on the target selected by flags.

To find the maximum synthesis frequency, RoadRunner's flow control subsystem is used. It allows a command to be controlled by a given Lua script, run multiple commands with varying flags, and react to their outcomes (e.g. exit codes).

To make the primary clock frequency controllable by RoadRunner, the main clock frequency generated in `ClockGen.sv` depends on the SystemVerilog define `FREQMULT`. The clock generator definition is altered to be:

```
clock_generator:
  /XILINX:
    sv: xilinx/ClockGen.sv
    env:
      FREQMULT: <%-freqMult%>
    ...
```

The SystemVerilog define `FREQMULT` follows the RoadRunner flag `freqMult` for this module. Running the synthesis command defined at `:synth` can now be parametrized with different values for `freqMult`, for example: `:synth+freqMult~42`. Using RoadRunner's `Flow` module, a Lua script is used to search for the maximum frequency that completes the synthesis successfully by running variations of the `:synth` command. The algorithm, shown in Listing 2, performs a search of halving intervals, narrowing down the optimal value for `freqMult` in $O(n \log(n))$ tries. The valid range is between 32 and 64. In Table I, the steps are summarized to reach the final result. The search algorithm ran for 3 hours and performed five syntheses unsupervised to obtain the result of `freqMult = 42` or `freq = 131.25MHz`.

VII. Conclusion

RoadRunner is an EDA project management system that focuses on flexibility, extendability, and separation of concerns. It features the config space, an attribute-set-list-tree data structure to capture metadata about EDA projects. Several dynamic extensions enable reusability and allow hierarchical and modularized project definitions. The definition of processing steps, source files, metadata, and the results are kept in the config space. RoadRunner separates the processing of data into two phases. (I) All necessary data, including sources and scripts to call the tools, are collected in an isolated working directory. (II) The work defined in a working directory is executed. An example is shown using RoadRunner to implement an MPSoC using different build systems for different design parts. It exploits the flexibility to provide specific tailoring for FPGA, ASIC, and simulation variants. It also showcases the flexibility of RoadRunner by running a set of synthesis runs to find the frequency maximum for an FPGA design efficiently.

```
--interval halving
lo, hi = 32, 64 --min/max for freqMult
while hi - lo > 1 do
  mid = (hi + lo) // 2
  t = task(":synth+freqMult~" .. mid)
  wait(t); s = status(t)
  if s.exitCode == 0 then
    lo = mid --sucess, set new min
  else
    hi = mid --failt, set new max
  end
end
print("optimal freqMult:" .. lo)
```

Listing 2: Algorithm to find the maximal frequency for the FPGA design. The synthesis task `:synth` is called with varying values for flag `freqMult`.

TABLE I: Automatic syntheses to find optimal frequency.

low / hi	freqMult	frequency [MHz]	synthesis successful
32 / 64	48	150	no
32 / 48	40	125	yes
40 / 48	44	137.5	no
40 / 44	42	131.25	yes
42 / 44	43	134.375	no

Acknowledgement

The project on which this report is based was funded by the German Federal Ministry of Education and Research under grant number 16ME0527. The author is responsible for the content of this publication.

References

[1] K. Asanovic *et al.*, "The rocket chip generator," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, vol. 4, pp. 6–2, 2016.

[2] E. van der Bij, "Hdlmake." Accessed: Jan. 23, 2023. [Online]. Available: https://ohwr.org/project/hdl-make

[3] J. Balkind *et al.*, "OpenPiton at 5: A Nexus for Open and Agile Hardware Design," *IEEE Micro*, vol. 40, no. 4, pp. 22–31, 2020, doi: 10.1109/MM.2020.2997706.

[4] A. Olofsson, W. Ransohoff, and N. Moroze, "A Distributed Approach to Silicon Compilation: Invited," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, San Francisco, California, 2022, pp. 1343–1346.

[5] P. Dabbelt, "PLSI: A Portable VLSI Flow," 2017.

[6] E. Wang, A. Izraelevitz, C. Schmidt, B. Nikolic, E. Alon, and J. Bachrach, "Hammer: Enabling reusable physical design," in *Workshop on Open-Source EDA Technology (WOSET)*, 2018.

[7] H. Liew *et al.*, "Hammer: a modular and reusable physical design flow tool," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 1335–1338.

[8] T. Ajayi *et al.*, "Toward an open-source digital flow: First learnings from the openroad project," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–4.

[9] O. Kindgren, "A scalable approach to IP management with FuseSoC," in *Proc. Workshop Open Source Design Autom.*, 2019.

[10] "PULP Training Slides." Accessed: Apr. 17, 2023. [Online]. Available: https://www.pulp-platform.org/docs/pulp_training/PULP_robert_manuel_deep_dive.pdf

[11] W. Kruijtzer *et al.*, "Industrial IP integration flows based on IP-XACT™ standards," in *Proceedings of the conference on Design, automation and test in Europe*, 2008, pp. 32–37.

[12] K. Wang, G. Tener, V. Gullapalli, X. Huang, A. Gad, and D. Rall, "Scalable build service system with smart scheduling service," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 452–462.

[13] S. Williams, "Icarus verilog." 2006.

[14] T. Developers, "Chisel/FIRRTL: Home," *Accessed: Mar*, vol. 18, 2021.

[15] S. Haas, C. Dunkel, F. Pauls, M. Hasler, and Y. Verma, "Trustworthy Silicon: An MPSoC for a Secure Operating System," in *2024 IEEE Nordic Circuits and Systems Conference (NorCAS)*, 2024, pp. 1–7.