Gherkin Implementation in SystemVerilog Brings Agile Behavior-Driven Development to UVM

William L. Moore
Paradigm Works
bill.moore@paradigm-works.com

***Abstract*—When silicon project documentation is incorrect or misunderstood, costly hardware bugs can escape into production. In the software realm, Agile methodologies such as Behavior-Driven Development (BDD) and the Gherkin language emerged to ensure code behavior matches the documented intent. Bathtub is a new library written completely in SystemVerilog which enables silicon design and verification teams to realize the benefits of BDD and Gherkin to facilitate collaboration and generate true living documentation—executable specifications that are always accurate, accessible, and up-to-date. This paper describes BDD, Gherkin, and Bathtub, and shares findings from their use in a silicon project.**

I. Introduction

Successful creation of silicon devices requires the collaboration of teams performing functions such as architecture, digital and mixed-signal design, design verification, software development, testing, marketing, and management. To be effective, all parties must share a common understanding of what the device does, i.e., its behavior. A persistent challenge is that often behavior is specified in documentation that is static and separate so that despite best efforts, changes to the specifications may not be reflected in the design and vice versa. References [1] and [2] report that changing, incorrect, or incomplete specifications cause over 40% of application-specific integrated circuit (ASIC) functional flaws and over 50% of field-programmable gate array (FPGA) functional flaws that escape into production. What's needed is living documentation bound to the design and validated by automatic processes such as Continuous Integration acceptance tests.

The software development community has pioneered Agile methodologies that increase coder productivity and code quality. Agile Behavior-Driven Development (BDD) specifically realizes the dream of living documentation. Teams practicing BDD create natural language documents with a machine-readable structure which can be parsed, interpreted, and executed as unit tests against their code. When the tests pass, the team can be confident that the code matches the documentation, and that the documentation is an accurate and accessible single source of truth for the code's behavior. Failing tests helpfully indicate a disconnect between specification and implementation.

Cucumber, supported by SmartBear, is a widely used BDD tool which introduces Gherkin, a syntax for documentation which Cucumber compiles into executable tests. Cucumber is available for several popular programming languages, but not SystemVerilog.

This paper introduces Bathtub, a library implemented entirely in SystemVerilog which parses Gherkin files, maps their behaviors to Universal Verification Methodology (UVM) virtual sequences, and executes those sequences against the design under test (DUT). I provide a brief overview of BDD and Gherkin and describe Bathtub's implementation and operation with a simple example. I share our experience and findings using Bathtub in a limited experimental capacity on an actual system-on-chip (SoC) project, and outline plans for future development. My goal is to demonstrate Bathtub's potential to make BDD available to silicon teams.

II. Behavior-Driven Development

Agile software development techniques help teams efficiently introduce incremental changes to a software project. Before an incremental change, or "increment," is begun, the team should collaboratively establish what feature they are building and why, how that feature will behave, and when the feature can be declared to be done. Behavior-Driven Development is a set of Agile practices and tools that address those questions by facilitating communication among members of three diverse collegial groups: business, development, and testing.

Business people represent the customer or end user of the software increment and are responsible for defining the high-level behavior of the feature. They are more interested in what the increment does than in the low-level details of how it works.

Development people are responsible for coding and delivering the working increment. They rely on their business colleagues to identify features that are valuable to end users. Developers are necessarily responsible for low-level technical details.

Testing people ensure the increment behaves as intended by creating and running tests of the software. Testers must have an accurate understanding of the intended behavior and operation of the feature. They are in many ways the first and most knowledgeable users of the software. Including testers in the initial stages of development can help identify problem areas early and prevent defects from being introduced in the first place.

Figure 1 illustrates how the team members interact in BDD use cases.

The first step in BDD is for a small team of business people, developers, and testers to meet in a series of "discovery workshops"—short, focused meetings with an agenda of defining completely the behavior of the upcoming increment. In these workshops, the participants clarify the concepts and language used to describe the feature. They summarize the behavior in high-level terms. They compile scenarios that illustrate how the increment works. Finally, they capture any questions that cannot be answered within the group. All of this is recorded in an informal loosely structured document which could be as simple as handwritten notes on index cards, or a shared text file.

Following the discovery workshops, a subset of participants formulates the descriptions and scenarios from the discovery workshops as an executable specification. This specification is arranged now in a formal structured format that is machine-readable so that tools can run them as automated tests against the increment. The prevailing format for these specifications focuses on concrete examples expressed in some version of the *Given-When-Then* pattern, described below. All discovery workshop participants are encouraged to review the specification to ensure its correctness and integrity.

Once the team has created the executable specification, they prepare it for test automation by providing test fixture code and writing "glue" code that interprets the natural language specification in the target programming language. Finally, they use BDD tools to parse and execute the specification, reporting out successes and failures.

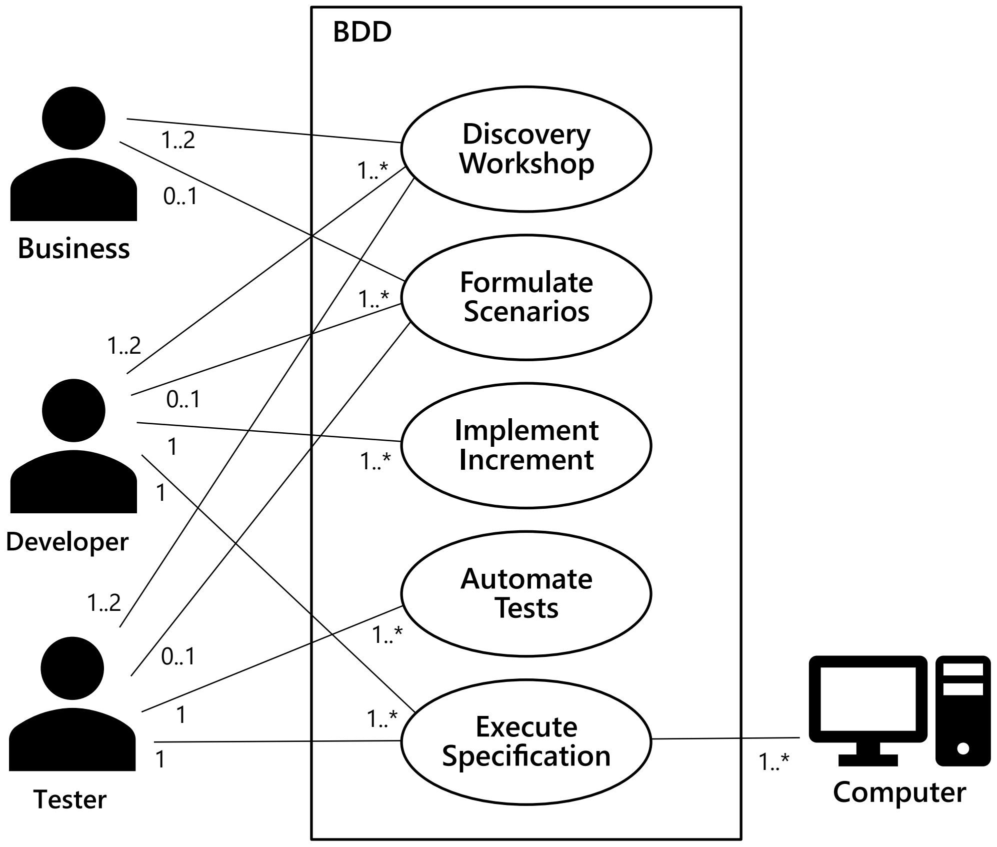

Figure 1. BDD use cases.

In the spirit of Test-Driven Development (TDD), the automated testing should precede the implementation of the increment so that the automated tests fail without the increment and pass only when the increment is complete and functional. Indeed, the automated tests may serve as acceptance tests that signify that the increment is done.

The executable specification is stored in the same source code management system as the code it describes. Since it is documentation, the team should make it accessible to all interested eligible audiences. Some source code management systems provide direct web access to raw managed files. Alternatively, teams may automate the process of publishing executable specifications to an accessible corporate intranet, perhaps with enhancements such as syntax highlighting and content searching.

III. Gherkin

The executable specification formulated from the outcome of the discovery workshops must have sufficient structure that it can be parsed and executed. What is needed is a file format that provides such structure while supporting human readability.

Gherkin is a free, open-source, domain-specific language for describing discrete behaviors of systems with concrete examples. A Gherkin file is a plain text file that developers and testers create with their preferred editors and maintain in the same management system as source code and traditional tests. A Gherkin file is known as a "feature file" because it describes the feature to be developed in an increment. The feature is described in free-form natural language and illustrated by concrete examples, or "scenarios." Each scenario is decomposed into steps beginning with keywords *Given*, *When*, or *Then*. By convention, *Given* steps are setup or preconditions, *When* steps are the key actions that define the behavior, and *Then* steps assert expected outcomes.

A. Discovery Example

Consider a fictional product team developing an Arithmetic Logic Unit (ALU), tasked with implementing an integer division function in the upcoming increment. In a discovery workshop meeting they discuss the division operation's behavior and write down some illustrative examples in an informal semi-structured format.

- Feature: ALU Integer Division
    - In integer division, the remainder is discarded
        - Example: 15 ÷ 4 = 3
    - Attempting to divide by zero results in an error
        - Example: 10 ÷ 0 → DIV_BY_ZERO flag is raised
    - Etc.

Everyone participating in the discovery workshop agrees that the recorded behaviors and examples completely and accurately reflect the business needs of the project and provide enough detail for the developers and testers to implement and verify the feature. Such collaboration and consensus are the intended benefits of the discovery phase of BDD.

B. Formulation in Gherkin

At this point, one or two technical participants take on the task of manually formulating a Gherkin feature file which covers the examples from their semi-structured discovery workshop notes. Fig. 2 is an excerpt from their Gherkin file which illustrates basic Gherkin syntax. Gherkin keywords are shown in **bold** to highlight them but Gherkin files are stored as plain unformatted text files.

```
# This Gherkin feature file's name is alu_division.feature

Feature: Arithmetic Logic Unit division operations

    The arithmetic logic unit performs integer division.

    Scenario: In integer division, the remainder is discarded
        Given operand A is 15 and operand B is 4
        When the ALU performs the division operation
        Then the result should be 3
        And the DIV_BY_ZERO flag should be clear

    Scenario: Attempting to divide by zero results in an error
        Given operand A is 10 and operand B is 0
        When the ALU performs the division operation
        Then the DIV_BY_ZERO flag should be raised
```

Figure 2. A sample Gherkin feature file.

Line 1: Comments begin with #. This comment helpfully demonstrates the convention that feature filenames end with the suffix ".feature."

Line 3: The *Feature:* line provides a title for the high-level feature. A feature file can have only one *Feature:* keyword.

Line 5: This is free-form text for documentation purposes. It can span multiple lines.

Line 7: A *Scenario:* is a concrete example of a discrete behavior, comprised of steps. The text is a description for documentation purposes.

Line 8: The *Given* step is declarative and sets up this division's operands.

Line 9: The *When* step indicates the procedural operation being tested.

Line 10: The *Then* step asserts the expected outcome. It is traditional to use the auxiliary verb "should" in Gherkin *Then* steps ("should be clear") but it carries the same meaning as stronger verbs like "must" or "shall" in that it is considered an error if the assertion fails.

Line 11: *And* is an alias for the preceding keyword that makes the scenario more readable than repeating the keyword *Then*. Gherkin has additional syntactic sugar step keywords *But* and *; the asterisk allows the user to create bullet lists of steps. Scenarios may have any number of *Given*, *When*, *Then*, *And*, *But*, and * steps in any combination.

Lines 13–16 describe the second scenario, the error case. Note that some lines are common between the two scenarios, differing only in literal values, illustrating that steps can be parameterized and reused across behaviors.

Leading and trailing whitespace is insignificant, but the prevailing convention is to indent keyword lines consistently.

Gherkin syntax includes additional keywords:

Rule: An optional layer of hierarchy between features and scenarios.

Background: A list of common setup steps automatically run before every scenario.

Scenario Outline: A parameterized scenario which is run iteratively with a list of literal arguments.

Examples: The list of literal arguments to be run iteratively through a scenario outline.

""" Multi-line doc string delimiter for steps that need more text than simple integer, real, and one-line string literals.

| Data table row and cell delimiter for steps that require one- or two-dimensional arrays of literal values.

@ Tags for labelling related *Feature:*, *Rule:*, or *Scenario:* blocks, e.g., for selecting a subset of scenarios to run.

Gherkin syntax consists solely of the keywords at the beginning of each line. The text following each keyword is arbitrary, not subject to any syntactic or grammatical requirements.

IV. BDD for Silicon Design and Verification

The BDD discussion to this point has been generic and programming language–independent. Still, BDD was created by software developers for software developers to improve their Agile software projects. This paper contends that BDD is equally applicable to the field of silicon design and verification. The end result of silicon development is physical hardware devices and systems, but in the front-end phase of modern hardware description language (HDL) development, a silicon project has much in common with a traditional software project. Register-transfer level (RTL) hardware descriptions and verification environments are code, and simulator runs are executions of that code. If a sufficiently self-checking test passes in simulation, then the code "works." As in software projects, silicon teams can use BDD to facilitate communication among project owners ("business"), RTL designers ("developers"), and design verification engineers ("testers"), leading to crucial common understanding about the behavior of impending design features.

Agile, continuously deployed software projects have faster development lifecycles than complete ASIC projects which can last many months or even years. However, every silicon project is unique and may have increments with durations and scopes well suited to BDD. For example:

- Periodic RTL releases to verification with staged feature enhancements and bug fixes
- Incremental design changes between silicon spins or chip revisions
- Intellectual property deliverables not tied to silicon fabrication timelines
- FPGA designs which can be programmed to devices at will
- Development and validation of UVM components themselves as opposed to RTL; here the design verification engineer is the developer in the discovery workshops.

V. Bathtub for Automation

Bathtub integrates BDD seamlessly into the hardware design and verification process. The Bathtub library is written entirely in native SystemVerilog so there is no need for a separate program to pre-process the feature file. The simulator runs Bathtub which reads the Gherkin feature file, parses it, then executes it as a self-checking test alongside and against the DUT. Furthermore, Bathtub utilizes and extends the industry standard UVM library and methodology for silicon verification.

Figure 3 depicts how information flows among the use cases from Figure 1. The team members generate documents and code in each activity for use in the next.

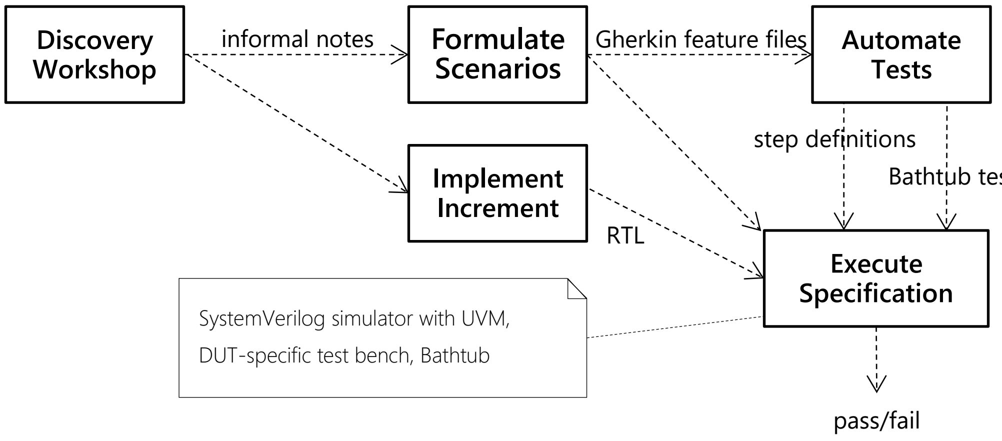

Figure 3. Bathtub information flows.

Figure 4 shows how Bathtub is integrated within a UVM test bench. The shaded regions are specific to Bathtub. The unshaded regions are not dependent on Bathtub and may be reused unchanged from pre-existing UVM tests or in future tests. With respect to UVM, Bathtub is an engine which runs automated virtual sequences on the test environment's virtual sequencer.

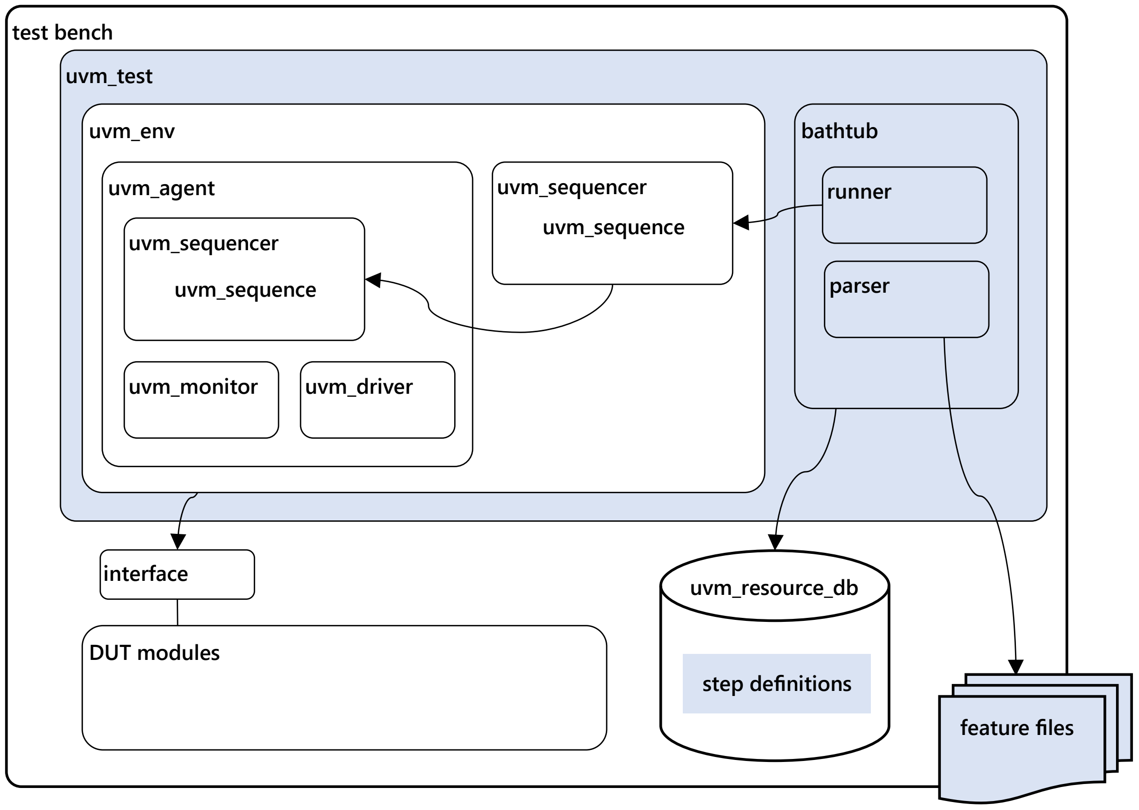

Figure 4. UVM test bench with Bathtub. Shaded regions are specific to Bathtub.

VI. Step Definitions

A Gherkin feature file at the lowest level consists of a series of steps—text strings that describe discrete operations in a natural language. For Bathtub to be able to execute a step, the user must write SystemVerilog glue code that maps each text string to a SystemVerilog task which Bathtub can call. Tasks are not first-class objects in SystemVerilog; they cannot be referenced by object handles or called indirectly. Therefore the user implements Bathtub steps as UVM virtual sequence classes, descended from base class `uvm_sequence`. The user overrides the sequence's `body()` task to perform the step's operation. Sequences, of course, are first-class objects. In BDD terminology, Bathtub virtual sequences are called step definitions. After a BDD team formulates the feature file, team members—typically the tester or developer—must code functional step definitions covering every *Given*, *When*, and *Then* step in the feature file.

Recall that feature file steps may be reused with different values. In Fig. 2, lines 8 and 14 are identical except for the integer literals which are the division operands:

Line 8 **Given** `operand A is 15 and operand B is 4`

Line 14 **Given** `operand A is 10 and operand B is 0`

The step definition writer could create two separate step definitions for these two steps, but they would be redundant and would cause maintenance difficulty if more values were added to the feature file. At their discretion the step definition writer may write parameterized step definitions in which the literals are replaced by placeholders to be filled in with actual values later. For placeholders, Bathtub uses SystemVerilog format specifiers. The step definition writer can represent the multiple step strings from Fig. 2 with one parameterized string using the "`%d`" format specifier for decimal integers:

```
Given operand A is %d and operand B is %d
```

Bathtub recognizes the following SystemVerilog format specifiers:

Format Specifier	**Meaning**
`%b`	Binary integer
`%o`	Octal integer
`%d`	Decimal integer
`%h, %x`	Hexadecimal integer
`%f, %e, %g`	Real number
`%s`	String
`%c`	A single ASCII character

The parameterized step string does not appear in the feature file; it is used only in the step definition, as described below.

Fig. 5 lists all the steps from Fig. 2 factored down to four unique steps with SystemVerilog format specifiers "`%d`" and "`%s`" taking the place of the integer and string literals, respectively. The step definition writer must write these four step definitions.

```
1. Given operand A is %d and operand B is %d
2. When the ALU performs the division operation
3. Then the result should be %d
4. Then the DIV_BY_ZERO flag should be %s
```

Figure 5. The feature file scenario steps parameterized with format specifiers.

Fig. 6 shows the first step definition.

```
1    // alu_step_definition.sv
2    class set_operand_A_and_B_vseq extends alu_base_vsequence implements
     bathtub_pkg::step_definition_interface;
3        `Given("operand A is %d and operand B is %d")
4        int operand_A, operand_B;
5         `uvm_object_utils(set_operand_A_and_B_vseq)
6        function new (string name="set_operand_A_and_B_vseq");
7            super.new(name);
8        endfunction : new
9        virtual task body();
10           // Extract the parameters
11           `step_parameter_get_args_begin()
12           operand_A = `step_parameter_get_next_arg_as(int);
13           operand_B = `step_parameter_get_next_arg_as(int);
14           `step_parameter_get_args_end
15           // Do the actual work using the API in the base sequence
16           super.set_operand_A(operand_A);
17           super.set_operand_B(operand_B);
18       endtask : body
19   endclass : set_operand_A_and_B_vseq
```

Figure 6. A sample *Given* Bathtub step definition.

Line 2 defines a step definition sequence that extends a presumed base `uvm_sequence` virtual sequence class that provides implementations of the required DUT operations. Every step definition must implement the Bathtub interface class `step_definition_interface`.

It is recommended that the verification environment creator follow standard UVM practices. In brief, typically the DUT is instantiated in a test bench and its ports are connected to SystemVerilog interfaces. The interfaces are passed as virtual interfaces to the UVM environment, which contains UVM drivers that drive stimulus to the DUT's ports under the direction of UVM sequencers. Those sequencers are coordinated by a virtual sequencer. The verification environment creator typically creates a virtual sequence base class compatible with the virtual sequencer. The virtual sequence base class sets up and provides access to all the driver sequencers so that child virtual sequence classes can communicate with the DUT. Bathtub step definitions should all extend that virtual sequence base class. The base class need not have any dependencies on Bathtub; the intent is that the user can reuse the same base class in Bathtub and in a traditional UVM test.

Back to the example, it is presumed that the environment has a virtual sequence base class `alu_base_vsequence`, not shown, that provides convenience methods `set_operand_A()` and `set_operand_B()`. Those convenience methods are the application programming interface (API) for the child classes to access the DUT. Class `alu_base_vsequence` can implement the convenience methods by calling `` `uvm_do() `` macros or equivalents on the driver sequencers.

In line 3, Bathtub macro `` `Given() `` registers the parameterized step string to be mapped to this sequence and automatically implements all interface class methods required by `step_definition_interface`. Bathtub also provides respective `` `When() `` and `` `Then() `` macros.

Line 4 declares two local integer variables for our operands.

The `body()` task of this UVM sequence does two things. First it extracts the actual arguments from the feature file step text and assigns them to our local class variables with Bathtub `` `step_parameter `` macros. Macros `` `step_parameter_get_args_begin() `` and `` `step_parameter_get_args_end `` expand into code that uses the SystemVerilog `$sscanf()` function to scan the actual step string from the feature file using the parameterized step string from the `` `Given() `` macro as a format string. Actual values returned by `$sscanf()` are stored in a queue in the order they appeared in the format string. The `body()` task pops the values from the queue with the macros `` `step_parameter_get_next_arg_as() ``. Fig. 6 lines 12 and 13 call the macros twice to get the two operand values as integers. The `` `step_parameter_get_next_arg_as() `` macro argument is a type which can be `int`, `real`, or `string` and should match the type of the variable receiving the argument.

After extracting arguments, the `body()` task delegates its work to the base class methods `set_operand_A()` and `set_operand_B()`.

Fig. 7 is a step definition for the last *Then* step from Fig. 5. Its sole parameter is a string, so the `body()` task converts it to a bit value that can be compared with the logic value from the DUT. The converting case statement accepts a variety of string values to give the feature file writers a flexible selection of synonyms to enhance readability. The `body()` task contains a SystemVerilog simple immediate assertion that flags an error if the assertion fails, causing the test to fail. Strong consequential assertions like this ensure that the Bathtub test is self-checking.

```
// alu_step_definition.sv cont.
class check_DIV_BY_ZERO_flag_vseq extends alu_base_vsequence implements bathtub_pkg::step_definition_interface;
    `Then("the DIV_BY_ZERO flag should be %s")
    string flag_arg;
    bit expected_flag;
    bit actual_flag;
    `uvm_object_utils(check_DIV_BY_ZERO_flag_vseq)
    function new (string name="check_DIV_BY_ZERO_flag_vseq");
        super.new(name);
    endfunction : new
    virtual task body();
        // Extract the parameter
        `step_parameter_get_args_begin()
        flag_arg = `step_parameter_get_next_arg_as(string);
        `step_parameter_get_args_end
        case (flag_arg) // convert the string to a bit value
            "raised", "asserted" : expected_flag = 1;
            "clear", "deasserted" : expected_flag = 0;
            default: `uvm_error("UNEXPECTED ARG", flag_arg)
        endcase
        super.get_div_by_zero_flag(actual_flag);
        check_DIV_BY_ZERO_flag : assert (expected_flag === actual_flag) else
            `uvm_error("MISMATCH", $sformatf("expected %b; actual %b", expected_flag, actual_flag))
    endtask : body
endclass : check_DIV_BY_ZERO_flag_vseq
```

Figure 7. A sample *Then* Bathtub step definition.

VII. Bathtub Operation

The user must create a UVM environment class that extends `uvm_env` and instantiates and configures the virtual sequencer and all other required components. This environment class need not have any dependencies on Bathtub and can be the same environment class used by other UVM tests.

Finally, the user must create a Bathtub-specific `uvm_test` test class. Figure 8 shows a minimal Bathtub test implementation for our ALU example. The `build_phase()` function instantiates a `baththub_pkg::bathtub` object and the environment object. The `run_phase()` task configures the Bathtub object by providing it a reference to the environment's virtual sequencer (line 17), and the file system pathnames of the Gherkin feature files (line 18). The test calls `bathtub::run_test()` to begin execution.

```
class bathtub_test extends uvm_test;
    `uvm_component_utils(bathtub_test)
    alu_env my_alu_env; // uvm_env containing the virtual sequencer
    bathtub_pkg::bathtub bathtub;

    function new(string name = "bathtub_test", uvm_component parent = null);
        super.new(name, parent);
    endfunction : new

    virtual function void build_phase(uvm_phase phase);
        bathtub = bathtub_pkg::bathtub::type_id::create("bathtub");
        super.build_phase(phase);
        my_alu_env = alu_env::type_id::create("my_alu_env", this);
    endfunction : build_phase

    task run_phase(uvm_phase phase);
        bathtub.configure(my_alu_env.alu_vseqr); // Virtual sequencer
        bathtub.feature_files.push_back("alu_division.feature"); //Feature file
        phase.raise_objection(this);
        bathtub.run_test(phase); // Run Bathtub!
        phase.drop_objection(this);
    endtask : run_phase
endclass : bathtub_test
```

Figure 8. A sample Bathtub test.

The user compiles a simulation with their DUT, test bench, step definitions, and Bathtub package `bathtub_pkg` and runs their Bathtub test, e.g., with command-line argument "`+UVM_TESTNAME=bathtub_test`." At time 0, the simulator automatically registers the step definitions and stores them as UVM object wrappers in the UVM resource database, indexed by regular expression. The format specifiers from the format strings are automatically replaced by equivalent POSIX regular expression terms. For example, integer format specifier "`%d`" is replaced with regular expression "`([0-9]+)`." (The actual regular expression for an integer is more complicated to account for optional sign, underscore, unknown, and three-state characters, but it is simplified here for clarity.) The following function call is an illustration of how Bathtub stores the step definition in the resource database.

```
uvm_resource_db#(uvm_object_wrapper)::set(
  "operand A is ([0-9]+) and operand B is ([0-9]+)",
  "bathtub_pkg::step_definition_interface",
  set_operand_A_and_B_vseq::get_type());
```

Function `uvm_resource_db::set()`'s first parameter, `string scope`, is a regular expression intended for wildcarded hierarchical component paths. We use `scope` here in a novel way for our step definition regular expressions instead.

Task `bathtub::run_test()` reads and parses the feature files with SystemVerilog's native file I/O functions and string operations, and builds a data structure of scenario steps. If Bathtub detects a Gherkin syntax error in a feature file, it flags an error and halts. Otherwise, Bathtub processes the steps in order, looking them up in the UVM resource database:

```
uvm_resource_db#(uvm_object_wrapper)::get_by_name(
  "operand A is 15 and operand B is 4",
  "bathtub_pkg::step_definition_interface",
  1);
```

If the step string with its literal arguments matches a regular expression in the resource database's `scope` field, then the associated step definition sequence object is retrieved from the database, instantiated, provisioned with the actual step attributes, and executed by the user's sequencer.

VIII. Results

As a design verification engineer at a startup that developed ultra low-power wireless SoCs, I became interested in innovating ways to improve technical communication among RTL designers, DV engineers, embedded software engineers, and management. Our documentation was high quality and effective, but we struggled with widespread structural challenges.

- Much of the documentation was static and subject to drifting out of sync with the design.
- Documentation and design developed on independent timelines so there was often lead or lag between the two.
- Documentation was stored in corporate collaboration systems that were separate from our code development environments and sharing among the systems was limited.
- Documentation was written with rich formatting which made automation difficult.
- It was challenging for colleagues from different domains to understand each other's technical work.

While learning about Living Documentation, BDD, and Agile, I conceived and implemented Bathtub and deployed it experimentally on the verification of two RTL blocks.

The first block was an Energy Aware Sub-system. It was primarily a vehicle for bringing up Bathtub, so the feature files were created by copying and pasting text from the existing design specification. As the Bathtub work was mostly solo, this wasn't true collaborative BDD but rather a necessary bootstrapping activity. The Bathtub bring-up was successful and the automated tests were included in our regression suite.

The second block was Encryption intellectual property. For this block, the team deliberately adhered more closely to classic BDD and conducted a small Discovery Workshop with the designer, a technical manager, and myself on verification. We employed a collaborative whiteboard app to capture scenarios and I formulated them into a feature file. Again, the effort was successful and the block ultimately passed all the acceptance tests.

One unexpected challenge was how difficult it is to balance the opposing goals of keeping the feature file readable and the step definitions writable. It's an art. I frequently iterated between rewriting scenarios that were too low-level or too general, and refactoring step definitions to keep them flexible and reusable, to manage persistent state between steps, and to avoid regular expression collisions. Gherkin is by design a sparse language, lacking basic structures like flow control and iteration. In contrast, SystemVerilog is full-featured and can express advanced concepts like concurrency and recursion. With deliberate phrasing, a seemingly simple scenario step can and should hide its complexity inside the step definition.

Writing step definitions is time consuming. They are small and relatively simple, but they are numerous. Plus the more definitions I wrote, the harder it became to keep track of them. I endeavor in this paper to highlight the benefits, but the return-on-investment of time is an open question.

Overall, this experience offered several insights.

- It is beneficial for representatives from diverse domains to collaborate effectively.
- BDD is not verification. Automated tests can demonstrate basic functionality but traditional approaches like constrained random testing are still required to close coverage gaps and find bugs.
- Bathtub does lay the foundation for verification since Bathtub requires UVM infrastructure to be brought up early. It's a smooth process to extend a directed acceptance test into wider, more aggressive stimulus.
- Executable specifications are a form of documentation, but they couldn't replace all our documentation. It would have been excessive to rewrite all our legacy specifications in Gherkin, and we still had need to publish traditionally formatted documentation for wider audiences. Executable specifications are best suited as working documents for teams producing new increments, and thereafter as internal reference material.
- With suitable UVM verbosity selected, all the scenarios and steps are printed to the log file which is a tremendous debugging aid. The test is self-documenting.
- Since feature files are read at runtime, modifying them does not trigger SystemVerilog simulators to recompile the code base. Even better, when running a simulator interactively, I could modify a feature file and simply reset the simulator to pick up the changes without having to exit. These abilities save time during debug.
- Enumerating design and verification tasks in a feature file naturally aids planning and progress reporting. If a feature file has ten scenarios, those are ten well-defined tasks which are easily tracked.
- Gherkin files are decoupled from implementation and are easy to reuse. I was able to modify step definitions to verify both RTL and its associated SystemVerilog reference model with the same feature file.
- There is power in the *Given-When-Then* pattern. Once embraced, it can be brought to bear on a variety of technical and nontechnical problem-solving activities. Just in the field of design verification, *Given-When-Then* thinking can bring clarity to verification plans, stimulus, assertions, functional coverage, and validation and verification of functional safety requirements.
- SystemVerilog is a capable language that can run general applications beyond the simulation of circuits.

IX. Prior Work

Reference [9] describes the innovative application of a BDD tool to the RTL design and verification flow. The authors of that paper use Cucumber and the Ruby programming language to generate Verilog RTL, test benches, and even Property Specification Language (PSL) assertions. Cucumber is run as a separate process prior to simulation. There is no mention of UVM for procedural verification.

SVUnit [12] is a well-established library which brings Agile TDD to SystemVerilog design and verification. Just as software BDD conceptually builds on software TDD, Bathtub builds on SVUnit. The key concept is writing unit tests or feature file acceptance tests before design work starts. I used SVUnit to unit-test some Bathtub functions.

X. Future Development

Bathtub is functional, but as of this publication there are Gherkin features that are not fully implemented. They include tags, locations, language localization, doc strings, and rules.

There are countless improvements to be made in Bathtub documentation, usability, and maintainability.

Bathtub is open-source software, available under the MIT License at https://github.com/williaml33moore/bathtub.

Acknowledgment

I would like to acknowledge my colleagues at Everactive, Inc., who encouraged the development of Bathtub with a culture of innovation, inclusiveness, and openness to new ideas. I would like to thank the following individuals for their contributions: Richard Sawyer, Alain Gravel, Chris Lukas, Farah Yahya, Carlos Olmos, Andy Shih, Jacob Breiholz, Yuyang Li, Thomas Hurtt, Brian Daniels, and Carmen Cook.

References

[1] H. Foster, "Part 2: The 2022 Wilson Research Group Functional Verification Study," Siemens Digital Industries Software, Dec. 28, 2023. [Online]. Available: https://blogs.sw.siemens.com/verificationhorizons/2022/10/24/part-2-the-2022-wilson-research-group-functional-verification-study/

[2] H. Foster, "Part 12: The 2022 Wilson Research Group Functional Verification Study," Siemens Digital Industries Software, Dec. 28, 2023. [Online]. Available: https://blogs.sw.siemens.com/verificationhorizons/2023/01/09/part-12-the-2020-wilson-research-group-functional-verification-study-2/

[3] C. Martraire, "Rethinking Documentation," in *Living Documentation: Continuous Knowledge Sharing by Design*, Boston, MA, USA: Addison-Wesley, 2019, ch. 1. [Online]. Available: https://learning.oreilly.com/library/view/living-documentation-continuous/9780134689418/

[4] J.F. Smart, "BDD—the Whirlwind Tour," in *BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle*, Shelter Island, NY, USA: Manning Publications, 2014, ch. 2. [Online]. Available: https://learning.oreilly.com/library/view/bdd-in-action/9781617291654/

[5] "What is Scrum?," Scrum. https://www.scrum.org/learning-series/what-is-scrum/the-scrum-artifacts/what-is-an-increment (accessed Dec. 29, 2023).

[6] "Cucumber Open - Get Started with BDD Today," Cucumber. https://cucumber.io/tools/cucumber-open/ (accessed Sep. 15, 2023).

[7] "Gherkin Reference - Cucumber Documentation," Cucumber. https://cucumber.io/docs/gherkin/reference/ (accessed Sep. 15, 2023).

[8] "cucumber/gherkin: A parser and compiler for the Gherkin language," Github. https://github.com/cucumber/gherkin (accessed Sep. 15, 2023).

[9] M. Diepenbeck, M. Soeken, D. Große and R. Drechsler, "Behavior Driven Development for circuit design and verification," 2012 IEEE International High Level Design Validation and Test Workshop (HLDVT), Huntington Beach, CA, USA, 2012, pp. 9-16, doi: 10.1109/HLDVT.2012.6418237.

[10] *IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language*, IEEE Std 1800™-2012, 2013.

[11] *Universal Verification Methodology (UVM) 1.2 Class Reference*, UVM 1.2, Accellera, Napa, CA, USA, Jun. 2014.

[12] J. Rensch, N. Johnson, "How Do You Verify Your Verification Components?," 2016 DVCon, USA, 2016.

[13] J. K. Brown et al., "27.1 A 65nm Energy-Harvesting ULP SoC with 256kB Cortex-M0 Enabling an 89.1µW Continuous Machine Health Monitoring Wireless Self-Powered System," 2020 IEEE International Solid-State Circuits Conference - (ISSCC), San Francisco, CA, USA, 2020, pp. 420-422, doi: 10.1109/ISSCC19947.2020.9063067.

[14] C. J. Lukas et al., "15.2 A 2.19µW Self-Powered SoC with Integrated Multimodal Energy Harvesting, Dual-Channel up to −92dBm WRX and Energy-Aware Subsystem," 2023 IEEE International Solid-State Circuits Conference (ISSCC), San Francisco, CA, USA, 2023, pp. 238-240, doi: 10.1109/ISSCC42615.2023.10067337.

[15] G. Nagy, S. Rose, *Discovery: Explore Behaviour Using Examples*, 2018. [Online]. Available: http://bddbooks.com/#discovery (accessed Nov. 13, 2023).

[16] S. Rose, G. Nagy, *Formulation: Document Examples with Given/When/Then*, 2021. [Online]. Available: http://bddbooks.com/#formulation (accessed Nov. 13, 2023).

[17] *Universal Verification Methodology (UVM) 1.2 User's Guide*, Accellera, Elk Grove, CA, USA, Oct. 2015.

[18] C. Cummings, H. Chambers. SystemVerilog OVM/UVM Verification Training Guide. (Mar. 6-8, 2013). Provo, UT, USA: Sunburst Design.

[19] "williaml33moore/bathtub: BDD Gherkin implementation in native SystemVerilog, based on UVM," Github. https://github.com/williaml33moore/bathtub (accessed Dec. 29, 2023).