# Graph-IC Verification

Dennis RAMAEKERS
ST-Ericsson
Grenoble, France
dennis.ramaekers@stericsson.com

Grégory FAUX
STMicroelectronics
Grenoble, France
gregory.faux@st.com

***Abstract***
In this paper, we describe the application of graph-based verification techniques to a complex and highly configurable display controller IP block. The main challenges, the technology and the major benefits are developed, including the reuse at system level for basic connectivity verification and more complex system-level tests involving multiple IPs.

***Keywords***
Graph-based verification, test generation, vertical reuse, scenario model

## I. INTRODUCTION

Highly integrated features and sophisticated processing capabilities are the factors driving today's portable electronics evolution. The smartphone and tablet market explosion made the corresponding SoCs integration and complexity grow in an exponential manner. Who imagined just a few years ago that we would have advanced 3D applications on a mobile platform? The hallmark of these new devices is innovative architectures and processors combined with the increasing integration level and never-ending addition of sophisticated logic. These all contribute to making the devices harder to verify.

Constrained-random verification, coupled with an appropriate standard methodology such as the Universal Verification Methodology (UVM), is the norm for IP verification today. Its major strengths are the automation for both stimulus generation and expected outcome computation, the ability to find unexpected corner cases and the reuse of material among multiple projects.

However, the ability to describe and generate complex scenarios is still a challenge with this technology. Vertical reuse, meaning the ability to reuse materials between verification levels, is also an important issue. Consider these two reasons:

- Logic complexity can be hard to manage at the IP level, but it is unrealistic for people working at the system level to acquire deep knowledge of multiple IPs. Knowledge and material sharing is crucial for productivity and quality.
- There are increasing demands for system tests that go beyond connectivity checks. Those tests include the execution of partial or complete data flows that involve multiple IPs and reflect real use cases of the system.

We begin by describing the challenges faced by ST-Ericsson during the verification of a complex video controller, at both the IP and system levels. We then proceed to provide details of the graph-based verification technology. The application of this technology to our controller is developed. Finally, we expose the benefits we received and the limitations we faced.

## II. VERIFICATION CHALLENGES

We will describe the complexity of the display engine and explain the verification challenges brought by the different video stream features. We will show the limits of a classic constrained-random approach as the verification solution and the difficulties associated with supporting derivatives.

The role of the display engine is to compose pictures and send video streams to display screens using various output formatters. We can identify several stages within this flow where each stage has its own configuration capabilities and application-specific constraints:

- A pixel fetcher stage that manages memory access and pixel fetching. It handles multi-layer picture data fetching and can handle up to four channels.
- Video channels that handle picture composition and configurable pixel processing.
- Output formatters that take pixels from a FIFO filled by the channel. They format the pixel stream for a dedicated display screen using different video protocols.

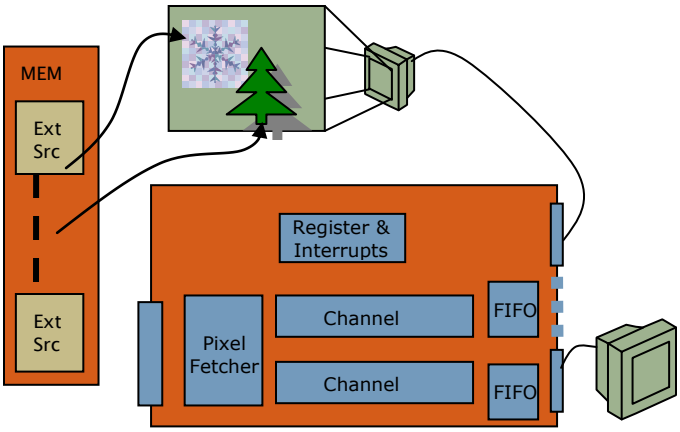


**Figure 1. An overview of the display processor**

For managing picture composition possibilities and supporting a wide range of possible video formats, additional features are necessary for pixel format conversion, resource management (such as input buffer or overlay configuration), flow synchronization (configured through multiple clock and synchronization modes) and communication with the external world (such as using interrupts).

Crossing configuration parameters for each stage generates a large pool of possible configuration candidates. Crossing configuration possibilities within each stage multiplies the pool of possibilities and quickly becomes unmanageable with numerous corner cases that are hard to identify. Adding the temporal possibilities with highly configurable synchronization modes makes the verification space explode and it becomes a nightmare to handle the pattern generation space in a generic non-directed way.

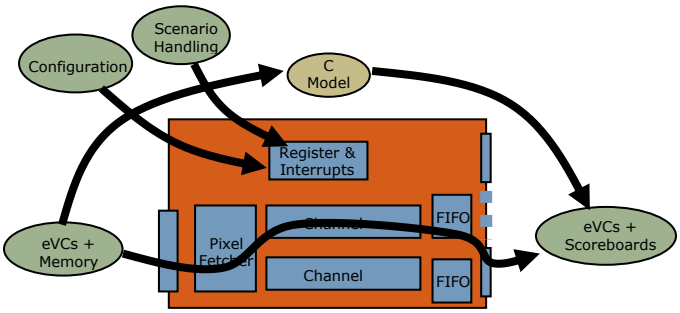


**Figure 2. An overview of the testbench**

Previously we verified the controller using a testbench based on ***e*** code and the associated *e*RM methodology for stimuli generation, results checking and coverage collection. These elements were coupled with a C reference model called during simulation. This model provided the expected outputs based on the design-under-test (DUT) configuration and incoming data.

The classic constrained-random approach can help with this list of possible configurations but it quickly reaches limits, especially with coverage collection and relevant configuration generation:

- The coverage collection database explodes when trying to cross all configuration possibilities.
- Configuration sequences require hundreds of registers writes that need to be described in a simple way. These are dependent on multiple generated fields.
- Random pattern generation needs to be highly constrained to ensure the relevance of the test cases generated. There are very strict steps in sequence generation that have to be enforced. This results in a huge number of interdependent constraints between each aspect of configuration and decreases the flexibility of random generation. This also impacts the ability to reach many corner cases and makes environment update and maintenance difficult.
- Resource management, especially resource allocation such as overlays or input buffers, becomes tedious due to the high configurability of the IP. This is especially true when resources are allocated or freed dynamically.

It is the multiplication of configuration possibilities at each stage that becomes hard to manage in a random generation context. A huge number of items that are strongly linked to each other must be generated. In such a complex generation space, the verification engineer loses controllability and visibility of the testbench. Making things worse is the fact that multiple engineers were involved in the creation of each part of the verification environment and each part needs to be compatible with each other for system-level verification. In this case, seven people worked on the IP over time through different projects. Nobody had a complete view of the environment, which caused knowledge transfer issues and made it difficult for new engineers to ramp up on the project.

Other consequences of the lack of visibility and controllability were test implementation difficulties and tricky debug sessions. The user could not easily describe new tests because there were a lot of parameters to define and generation contradictions were often raised because of the interdependencies. As the configuration parameters were distributed and tightly linked to each other, debug was difficult on both sides. On the RTL side it was because of the number of cross-configuration possibilities, and on the verification environment side it was due to the constraint network applied to pattern generation.

The complex interdependencies meant that the verification environment created is tightly linked to the specification of the IP. This made horizontal reuse for several IP versions difficult. IP evolves with SoC evolution and each project requires new features, variants of existing features or removal of others. As configuration and sequence generation is dependent on several features of the IP, due to the interdependencies, changing, removing or adding a feature can result in a significant update to the environment. Reusing IP on a new project often required code duplication, significant adaptation of existing parts, and parallel maintenance for multiple versions.

As expected these difficulties were compounded at the top level. The complexity of configuration parameters and sequences was high and we were unable to benefit from IP experience due to the differences in verification approaches. Having constrained-random sequences is not directly transposable to a C-code execution flow such as the one used for system top-level verification. For vertical reuse we had put in place a mechanism that probed configuration data at the IP level and transformed it into something useable for the system-level verification team, but it still needed a lot of support and was not flexible.

We decided that to improve the verification environment and find a better solution to tackle the problems outlined, we had to consider a new approach. A graph-based verification flow appeared to be a promising technology. This would enable a platform-agnostic application with a more natural way to describe configuration and scenario goals. The tool was expected to bring about clarity in constraint definition and also to enable vertical reuse between the IP and the system-level verification teams.

## III. GRAPH-BASED VERIFICATION

### A. Technology outline

The solution to be described is the one proposed by Trek, a Breker Verification Systems tool. It makes use of a directed graph, which can also be cyclic. The graphs are composed from three kinds of node (also called *goal*). Using the three node types, the user expresses the way the graph walk, or evaluation, happens. A *leaf* node has no children. It corresponds to any vertex with no outgoing edge. The two other node kinds, called *select* and *sequence* nodes, have an arbitrary number of children. For a select node, one of the possible edges will be selected and the corresponding subgraph evaluated. For a sequence node, all subgraphs are evaluated in order.

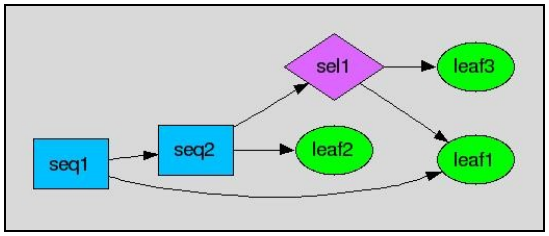


**Figure 3. A simple graph with two possible walks**

Figure 3 shows a simple graph. *seq1* is a sequence node, and its two children, namely *seq2* and *leaf1*, are evaluated in that order. *seq2* is also a sequence node that will evaluate *sel1* followed by leaf node *leaf2*. *sel1* is a select node, meaning that only one of its children, *leaf3* or *leaf1*, will be evaluated. Therefore there are two possible paths for this graph, corresponding to the two following sequences: (*seq1, seq2, sel1, leaf3, leaf2, leaf1*) and (*seq1, seq2, sel1, leaf1, leaf2, leaf1*).

Graph traversal relies on a random evaluation of each select node's subgraph. The random mechanism uses a seed that guarantees the reproducibility of the generated scenarios and configurations. Thus, the users will run multiple tests with different seeds to cover their scenario and configuration space, but one given seed will lead to the same branch choices and variable generation.

Additional capabilities are provided that influence node sequencing. A weight may be assigned to any select node's subgraph that would modify the probability of a child being selected. Similarly, a repeat value can be assigned to any child of a sequence node. This corresponds to the number of times this child is evaluated before the next node is considered. The default value for repeat and weight is one. A value of zero is allowed for both. A child with a zero weight is never chosen. A child with a repeat value of zero is never evaluated.

We can also apply constraints to the graph walk. Constraints can be applied to the children of select nodes. Possible constraints are *forcing* and *masking*. When a child of a goal is masked it will never be evaluated. This corresponds to a weight of zero. On the other hand, we can force a path in order to direct the graph walk. When a child of a select node is forced, it will be systematically chosen for evaluation. This corresponds to setting the weight of the other children to zero. The mechanism of mask and force can be applied statically or applied and removed dynamically. Users may constrain all instances of a given goal, only instance(s) from given subgraph(s) or mask instance(s) upon a selected node evaluation.

Concurrency is mandatory to cope with the parallel nature of hardware design and verification languages (HDL/HVLs). This is possible as any subgraph can be declared as a thread, with possibility to stop/resume at any time. Thanks to that, multiple subgraphs can be evaluated simultaneously.

### B. *A closer look at the syntax*

The graph is captured using a Backus-Naur Form (BNF).The following code describes the three nodes of Figure1.

```
goal leaf1;
goal seq1 := seq2 leaf1;
goal sel1 := leaf3 | leaf1;
```

In addition, the user can specify a C++ function for each node. This corresponds to the actual node evaluation and is executed before its subgraphs are evaluated. This method has multiple applications:

- Enabling arbitrary logic computations for the evaluation of goals. This also allows the argument list to be modified
- Changing the default order of evaluation (body then subgraph) by explicitly calling the subgraph evaluation
- Calling to a reference model of the design
- Interfacing with an existing testbench (***e***, SystemVerilog, SystemC, Verilog) for VIP configuration, event waiting, method calls, data injection or grabbing
- Writing C code, input data or expected results files in the scope of test generation for system level

Those last two applications are important to understand because they are enablers for reuse across multiple levels of verification. Conceptually there is a split between a first set of goals that is platform agnostic (common between all verification stages) and a second set that is tightly connected to the platform. Examples of such goals are register read and write, wait for event (such as interrupt) and testbench component access.

Consider the example of a register write. At the IP level, we made usage of FIFOs to communicate between the graph and existing ***e***-based environment. This is the case here: the graph sends its requests on one side and the corresponding VIP gets its operations from this FIFO:

```
goal write_reg (address, data) {
   var txn;
   txn["address"] = address;
   txn["data"] = data;
   // Direct call to goal portSend
   :portSend ("REG_WR", txn);
}
```

Our SoC verification environment relies on C tests running on embedded CPU(s). Therefore, the outcome of the graph evaluation should be C code. Our register write goal simply dumps the required code in our test file:

```
goal write_reg (address, data) {
   // Retrieve register name from address
   var regName = :get_reg_name(address);
   // Dump to C test file
   ::log("&MCDE+",regName,"=",data,";");
}
```

A generator node (declared using the generator keyword instead of goal) is the root of a subgraph whose evaluation can be blocked anytime thanks to a *wait()* action. When reaching a wait point, the subgraph evaluation is held and returns to the parent of the generator node in order to continue further. Thus, graph execution continues even though the subgraph held by the wait action is not completed yet. The wait state will be released when the generator node is called again or when a specific call to the held thread is done using a dedicated primitive, called *next()*. As already stated, generators permit implementing multiple parallel threads and managing interaction with the testbench.

Any node body method and subgraph can be redefined, the last definition load being the one used. This allows extension of existing graphs in a non-intrusive manner. Applications are multiple: graph reuse across projects and across platforms, specific test definitions, etc.

### C. *Modeling with a graph*

When formulating a graph, the user starts with a node representing the desired outcome of the graph evaluation, that is, the verification intent(s). The challenge is to decompose this functional intent into a set of subgraphs that will generate the required stimuli to achieve this functional intent together with the expected outcomes of the design.

What was a high-level node in this graph is likely to be used as a subgraph within a higher-level verification scenario where multiple IPs are combined, or for a directed scenario that might be created for performance or stress testing. This compositional process is further developed in the following section, based on the controller example.

## IV. APPLICATION ON OUR CONTROLLER

The original ***e*** environment was presented in Section I. We decided to introduce the graph-based technology for the scenario and stimuli generation. This choice was obvious: it leveraged large parts of our existing environment while improving its flow. This introduction also enabled vertical reuse for connectivity checking at the top level. A diagram for the resulting testbench is provided in Figure 4.

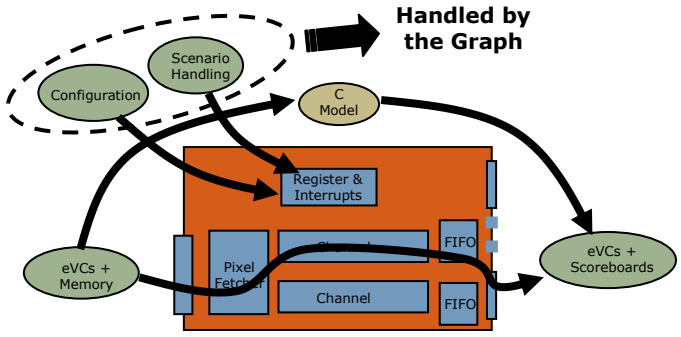


**Figure 4. The part handled by the graph**

Shown below is the BNF for a simplified graph for one video channel of the controller. The corresponding visualization is shown in Figure 5.

```
// Repeat between 1 and 3 video streams
goal testChA  := testCh:1,3
{…}

// Stream handling modeled as a sequence
goal testCh := selFifoItf
               genChnl
               enableChnl
               processFrames
               disableChnl
{
   // Force the subgraph walk
   …
   // Unlock resources
   …
}

// Select/configure out FIFO and video itf
goal selFifoItf := selectFA | selectFB;

// Out FIFO A only connects to interface 0 & 1
goal selectFA := itf0 | itf1;

// Out FIFO B only connects to interface 0 & 2
goal selectFB := itf0 | itf2;


goal itf0{
   // Once an interface is selected, mask it
   // so that the other channel won't use it
  …
}

// Select/configure a channel
goal genChnl := genOvl
                genPixProc;

// Select/configure overlay(s)
goal genOvl;

// Generate pixel processing pipeline config
goal genPixProc;

// Enable channel
goal enableChnl;

// Process video stream
goal processFrames := waitINT
                      waitFrameEnd;

// wait frame interrupt
goal waitINT := wait_event
{…}

// Wait Frame end from VIP
goal waitFrameEnd := wait_event
{…}
```

Several interesting features are addressed here, such as locking resources (in *itf0* leaf node), channel reconfiguration (repeat applied on *testCh*), and a connection to the ***e*** testbench (via FIFO in the *wait_event* node). Overall, this graph models the use of a channel and the related generation aspects.

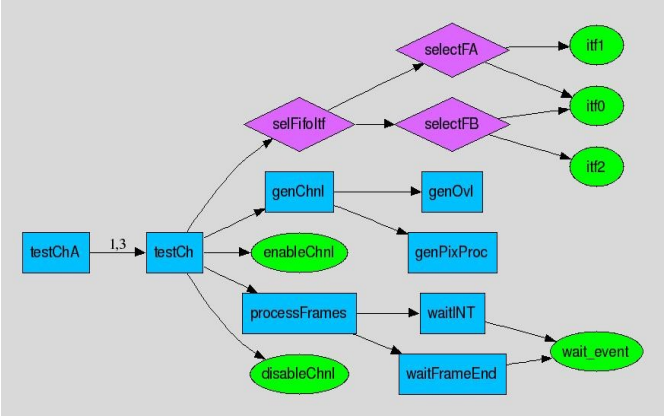


**Figure 5. Visualization of the channel graph**

An important requirement associated with the verification of the controller is the ability to generate C code for system-level tests. The graph is evaluated to generate a C test file, memory initialization file(s) and expected output file(s) for post simulation checking. This relies on the redefinition of nodes that were tightly connected to the platform as already explained in Section II.B. The process is illustrated in Figure 6.

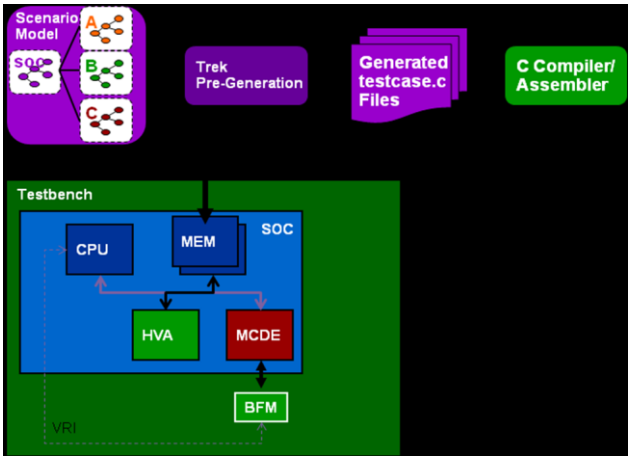


**Figure 6.The C generation flow**

The first application of this code generation capability was the creation of the connectivity checking tests. The natural extension of this first test generation is the possibility to compose multiple IP graphs to generate more interesting test cases when combining multiple IPs at the system level. This is a way to exercise complex and realistic data paths through the system. While the automation of integration test creation is valuable, generating these system-level tests would be a productivity breakthrough.

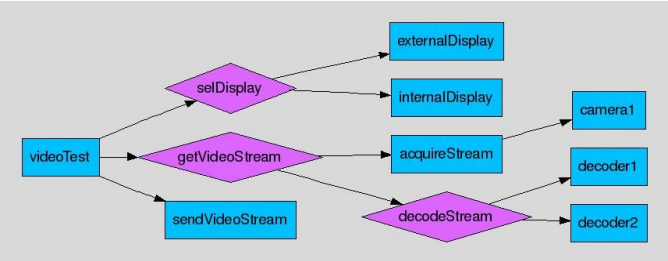


**Figure 7. Graph combining multiple IPs**

The test resulting from the Figure 7 graph traversal will generate an interesting system-level use case of a video data flow. Multiple entities will be involved during this test. Each one may apply several constraints. For example, the internal display and the external one (typically a TV) may not support the same video format or resolution, or will not require the same kind of video processing. Similarly, both decoders will not support the same kind of encoding format. Those differences will constrain the selection of the encoded video stream.

Since they involve multiple IPs at the same time, our generated C tests must also handle the CPU resources sharing between them. In other words, our C test, running directly on our CPU without any operating system, should execute each IP code in a multi-threaded manner. Let's take the example of an IP scenario waiting for an interrupt triggering. The corresponding C code cannot afford to wait actively for it. Instead, it should stop execution, allowing the CPU to run another IP code. The C test guarantees that our IP code will be resumed upon IT triggering. To ease such a test generation, the solution is delivered with a top-level graph library and the associated methodology. All IP graphs should be hooked to this top-level graph and use several predefined leafs.

## V. RESULTS

Applying a new methodology on top of an existing environment is not the easiest way to upgrade. But it enabled us to measure the effectiveness of this methodology and to enjoy the benefits of the problems that were solved.

The first obvious improvement brought by the graph-based approach is clarity in scenario and configuration generation. Even if the description is still done using a programming language, the approach for structuring the configuration and scenario elements is still close to drawing the scenario on a whiteboard. It forces you to start with a high-level idea (the verification intent) and to refine it into additional levels of detail. The graph methodology helps you to separate problems and to use a step-by-step approach for the creation of configurations and scenarios. Building this tree ensures readability and clarity. This is helped by displaying the tree in a graphic manner: each node clearly corresponds to some features or scenario elements, and choices between branches are easy to follow up. The model execution starts with what you want to test, and you constrain the graph walk to reach the final scenario you want. This way you will easily get all of the elements required for your test.

The generation flow is both more readable and more controllable. Having the complete configuration and scenario elements split into sub-parts and having an intuitive structure that enables you to reach each node, allows the user to easily apply constraints for graph crossing and for the generation of values for the configuration fields. Each feature will have its own branch (or sub-tree). This structure also allows having fewer global constraints and by the way reduces the interdependencies between the parts.

Having the scenario and configuration generation graph viewable, in a user-friendly way, enables newcomers to ramp up quickly. It is easier to explain and to understand the feature configuration when it is described step-by-step with a natural sequencing as it is in the graph. It is also easier for understanding how to build a complete scenario. Indeed, you will start describing the scenario with the end goal in mind and go through the graph with only the branch choices you need or want.

This last aspect enables easy scenario description. We were able to quickly generate different sets of test cases. We started with very simple cases for a basic and sanity test suite, and we were able to rapidly add other test cases which exercised additional features. At the end we ran tests with an "everything is possible" aspect, leaving the configuration as random as possible, but having a step-by-step test suite with gradually increasing complexity permitted us to separate problems and to have well-identified debug areas.

One other aspect brought by this more user-friendly description of the scenario is easy access to the information. It helps during debug session as it provides a clear view of the scenario. As there are fewer interdependencies, the debug context can be reduced and the user can focus on relevant aspects of the scenario or the configuration. Moreover, during debug sessions the user can add some temporary constraints for failing test simplification which aim at reducing the debug area further. For example you can disable one specific feature and check the effect. If the test continues to fail in the same way you can continue debugging without losing time on the now-disabled feature. Of course this is also possible with a classic constrained-random approach, but given that all you have to change is one branch selection it is really easy and well identified. Adding similar constraints in a full constrained-random environment may influence several parts of it, especially when there are a lot of interdependent constraints.

These statements might seem a bit subjective, but they are shared by the people who worked on this project. Let's consider some more concrete advantages to get a better idea about the gains this methodology has brought us.

The controllability on scenario generation brought by the tool permitted us to describe some test cases that we did not run on previous versions and to re-write existing test cases leading to environment dead-ends. Some additional complex scenarios could be run and thus filled in the holes in our test suite. This is especially true for multi-channel tests with channel or resource on-the-fly reconfiguration that were very tricky to handle in our previous environment. We can distinguish five test families that were improved or created thanks to the graph approach:

- Overlay on-the-fly reprogramming
- Error tests
- Complex reconfiguration scenarios
- Channel reset feature
- System memory boundaries crossing for huge or non-aligned overlays

As explained in Section III, the result of a graph evaluation can be exploited in different ways. At the IP level we use the output of the graph for driving some ***e*** code, and the same output data can be exploited for generating some C code. This permits us to constrain the graph for a specific walkthrough and to replay the same scenario and configuration at either the IP or top level. This step was really important and was the enabler for a real test case exchange between IP and top-level verification teams. Instead of reworking log files generated after a tedious specific test case writing, we could define precisely some graph constraints and share with both teams.

One additional discovered benefit was the capability of easily reusing an existing graph for different projects. With a graph you can over-constrain some parts or over-write some goals to obtain a variant of the graph that could fit other IP versions. Typically between IP versions you will have different numbers of channels, some new pixel processing methods or removal of others, or different number of resources (buffers, overlays, etc.) By describing one graph that supports all of the possibilities, we could capitalize on a common description that can be reduced for each version. We were able to split the environment into a "common" part and a "project" part. The common part is shared by all projects (i.e,. all different IP versions verified in parallel). It supports all the IP features with all configuration possibilities. The user will then configure the environment in the project part to match the current version of the IP.

- Original testbench: ~28000 *e* code lines
- New testbench:

| | *e* code | Graph |
|---|---|---|
| Common | 10000 | 8500 |
| IP specific | 3500 | 2000 |

**Figure 8. Code lines count**

To give an idea, Figure 8 represents 650 goals for the common part and 80 goals in the IP specific part.

This really accelerated the environmental setup for a new IP version. It also assists with maintenance: the more you put in common, the less you have to duplicate or rewrite between projects! Currently, an environment setup for a new IP version usually takes less than two weeks.

This is a big step forward in execution efficiency compared to the time previously needed (more than one month for having the ***e*** code and the C model working together!). It permits us to quickly benefit from existing parts for the new hardware configuration of the IP.

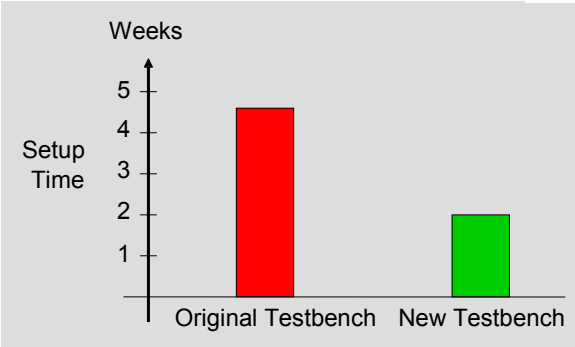


**Figure 9. Setup time gain with the new environment**

Thus, the verification engineer can rapidly focus on new features or updates, parts that contain most of the bugs in the new version.

We have been surprised by the RTL bugs found thanks to this new environment. Indeed, re-writing some feature configuration permitted to have cleaner configurations. Applying previously existing scenarios but using the new environment made us find some unexpected bugs. For example, the programming of a highly configurable output formatter was under-exercised due to too many interdependent constraints. Only a subset of possibilities was previously exercised on this formatter and thus hid some bugs. To conclude, application of this graph-based approach permitted us to also improve quality regarding the IP functionalities.

## VI. CONCLUSION

We have shown a representative application of graph-based verification for scenario and stimuli generation for a complex video controller IP. We also developed the capacity to reuse the scenario model for generation of IP integration C tests at the system level. The possibility of generating tests to exercise dataflows of multiple IPs has also been addressed. The resulting benefits were a more readable testbench, which is easier to maintain, shorter debug sessions and increased quality of the IP.

## VII. ACKNOWLEDGEMENTS

Thanks to Adnan Hamid, CEO of Breker Verification Systems, for his help in this project.