Abstract—Mutation-testing has been considered as an important coverage metric to measure the quality of simulation-based verification and validation processes [1, 2, 3]. On the other hand, IP-XACT has evolved to the IEEE standard for IP reuse and IP-based System-on-Chip (SoC) integration, which covers both RTL and TLM. In this paper, we present our effort to enable the mutation-based simulation coverage metric for system level IP integration with IP-XACT. Two major ingredients are required for this extension. First, as IP-XACT system designs are XML files, which are not originally for execution, we need an execution/simulation engine for IP-XACT designs. For this, we created a code generator that generates SystemC models from IP-XACT XML designs, such that we can simulate and test an IP-XACT design. Second, we define the mutation operators on IP-XACT schema, which is the model of errors that we can inject into IP-XACT designs during mutation testing. With IP-XACT, the mutation maintains a focus on the integration and configuration of components. We implemented the code generator and mutation operators in an Eclipse-based IP-XACT editor with the help of Eclipse Modeling Framework. Then several experiments were conducted on a TLM library for CoreConnect SoC modeling. From the results, we can see that the defined IP-XACT mutation serves an effective qualification for simulation tests, in terms of its ability to reveal the weakness of the tests.

Keywords—mutation testing; IP-XACT; IP reuse; SoC verification

I. INTRODUCTION

IP reuse is the mostly employed method to promote our electronic design productivity, which is urged to keep up with the increasing silicon capacity. Meanwhile, the number of IP cores that we integrate into a System-on-Chip (SoC) also increases dramatically. The verification problem of these IP-based integrated systems cannot be conquered without systematic approaches. Early in [5], unique challenges of SoC verification have been discussed. It is pointed out that the verification of SoCs has its uniqueness coming from the fact that a SoC is a complex electronic system and at the same time highly integrated from multiple independent components. Compared to traditional board-level systems, it is not possible that we use glue logics to patch an after-manufacturing SoC problem. Therefore, it is required for SoC designs that more comprehensive and stringent verification efforts are devoted.

This leads to the question how we can measure effectively the progress and quality of a SoC verification process. For simulation-based verification of electronic designs, various coverage metrics are the usual answer. Examples include the most basic code coverage and the functional coverage provided by emerging verification languages like SystemVerilog [10]. Distinct from others, mutation testing [14, 15] provides a systematic coverage measurement by qualifying a test with its fundamental ability to reveal design errors. Though originated from software testing domain, it has been widely studied and accepted for hardware design verification [1, 2, 3]. An industrial tool called CERTITUDE™ for generic HDLs including VHDL and Verilog is introduced in [9, 4]. Figure 1 shows the principle of mutation testing.

A mutation of a design, as an error injection, is a single, syntactically correct modification to the design, such as replacing an and operator by or:

\[
C \leq A \text{ and } B; \quad \Delta \text{ mutation: } C \leq A \text{ or } B;
\]

The resulted design copy is called a mutant. Then a test is qualified by simulate under this test both the mutant and the original design. If the test is good enough, it should be able to distinguish the mutation by producing a different output between the two simulations and the mutant is said to be killed by this test. We can generate thousands of mutants by different types of mutation at all relevant locations of the design. The percentage of killing after applying all the tests becomes the coverage metric to measure the quality of the overall testing or simulation process.

On the other hand, it can be noted that SoC design and verification consists of two separate phases. At the component level, possibly by different vendors, IPs are designed conforming to strict interface protocols and they are meant to be verified as thorough as possible. Then at system level, we integrate these IP components around the chosen on-chip bus architecture or network. Now the verification assumes correct components and should concentrate on the integration of the
components. When applying the mutation based coverage to the verification of individual IP component, we try to kill efficiently as many mutants as possible in the manageable time. High mutation-based coverage helps to eliminate potential verification holes of individual components, which can be hard to identify later in their integration phase.

To apply a system level mutation testing, instead of relying on generic languages like VHDL or Verilog, we find the IP-XACT standard [13] specifically aimed at IP-based system integration, which makes its exact suitability. By mutation testing with IP-XACT, we keep the verification focus on the integration, configuration and interaction of IP components. Moreover, IP-XACT is a well-accepted IEEE standard for IP reuse, which implies more potential adoptions of an IP-XACT based verification metric. In this work, we put practical efforts to define and implement an IP-XACT based mutation testing.

We present in this paper our work as follows. We at first give a necessary introduction to the IP-XACT standard. Then we explain in Section III our definition of IP-XACT mutation testing and how that is realized. Several experiments are reported in Section IV. We review some related work in Section V and then conclude by Section VI.

II. IP-XACT STANDARD

The purpose of IP-XACT [13] is to establish a unified format for IP exchange and reuse among different IP providers, system integrators and tool vendors. At the core, it defines an XML schema as the data model for describing the meta-data of IP components and their integrations as systems. Around this standard data model, a vendor-neutral IP-reuse environment can be established as shown in Figure 2.

![Figure 2. IP-XACT design environment [13]](image)

Before describing a component, schema definitions busDefinition and abstractionDefinition are used to specify the port constraints and other properties for a bus interface. In abstractionDefinition, ports can be described as either transactional for TLM or wire for RTL. This makes IP-XACT cover both TLM and RTL components.

With schema definition component we package an IP core. It contains the reference to the real design files of the IP and declaration of the physical ports as in the IP. It describes the bus interfaces implemented by the IP, along which the address space and register files on this interface are also defined. Here we have further the description how the bus specification ports are mapped to the IP physical ports. Moreover, an IP can also be configurable by declaring parameters, which may have default values and later resolved during system integration.

An IP-XACT design describes a SoC system, or subsystem integration. Mainly, we have the instantiation of components and their interconnections. In the instances, we configure appropriate values to component parameters. Interconnections between components can be established through their bus interfaces or in an ad-hoc manner, i.e. direct port-to-port connection.

Besides, the schema provides the opportunity to encapsulate IP configuration and other third-party tools as Generators, which can further be weaved to form Generator Chains. A Tight Generator Interface (TGI) defines both the API for invoking the tools from a main IP-XACT design environment and a set of Web-service based interfaces that this environment should provide to the tools for accessing and manipulating its IP-XACT files. This enables independent development and integration of IP-XACT compliant tools, even on a distributed basis.

As it became an IEEE standard, IP-XACT offers the best potential to drive further our IP-reuse productivity. A discussion on industrial application of the IP-XACT based SoC design flows can be found in [8].

III. MUTATION TESTING WITH IP-XACT

IP-XACT provides us a good opportunity to define the systematic mutation-testing at SoC system level and maintain the focus on components integration. With Figure 3 we show the two major requirements for enabling this IP-XACT based mutation testing.

Since IP-XACT describes designs as XML files, which are not naturally executable, this becomes our first challenge. A simulation engine for IP-XACT designs is the prerequisite that we can apply mutation testing on the simulation processes. For this, we built a code generator, which takes an IP-XACT XML design file as input and generates a SystemC simulation model. As SystemC has an immediate simulation engine, we are then able to simulate and test our IP-XACT design.

Second, the core requirement of mutation testing on any language is the definition of a set of mutation operators. These operators are the mechanism how we mimic the typical design errors and inject artificial errors into a design. They define where and what syntactic changes we can make based on the language constructs. Here, the language is IP-XACT schema, upon which the mutation operators should be defined. In the next subsections, we explain in details how the code generator works on IP-XACT and how the mutation operators are defined.
A mutant is killed, or detected, if a checker of the testbench states a fail during the testing with the original design and a SystemC code generator, assuming an IP database exists in the design environment. As introduced, an IP-XACT design contains simply component instances and the connections between them. The code generation engine takes such an IP-XACT design XML file as input and, based on an IP component repository, executes the following subtasks:

- It checks the validity of the involved IP-XACT descriptions, which includes the design input and the referenced component and busDefinition/abstraction Definition descriptions from the repository. Any missed reference will terminate the process.
- Besides the schema definition, IP-XACT standard also defines a set of semantic consistency rules that an IP-XACT document should obey for example, whether two transactional ports are compatible to be connected together. These are also checked for the design.
- After the checking and gathering of necessary information, a SystemC design file is generated. In this file, we create instances of the design components with their parameter configurations. Component interconnections are established by port-interface/port-port bindings, for TLM, or port-wire connections for RTL.
- Further, the code generator composes also a Makefile. The purpose of this Makefile is to enable a seamless launch of simulation for the IP-XACT design. Information like source files and library dependencies is collected from the IP-XACT descriptions for components. As IP-XACT schema specifies common types of design files that can be used, such as vhdlSource, verilogSource, systemCsource, swObject, etc, this eases the task of Makefile generation.

At the end, this Makefile is ready for a direct make simulation, which will compile, if necessary, all the referred IPs and the SystemC top design and start a simulation immediately. The benefit with SystemC simulation is that, by a simulation tool such as QuestaSim from Mentor Graphics, it is convenient to get a co-simulation of SystemC and other HDLs, so as to cover both TLM and RTL IP components.

A big challenge for the code generator lies on the creation of appropriate SystemC connections between components. At RTL, as ports have a narrow range of type choices, their binding is quite effortless. On the contrary, TLM ports are
bound via the SystemC interfaces that they implement or expect for communication, which are abundant through user extensions. We need to devise a mechanism for determining whether and how two IP-XACT TLM ports can be connected. Figure 5 illustrates our first proposal.

First, we notice that SystemC interface classes for TLM communication can be considered as a non-private inheritance tree starting from `sc_interface`, as the example figure shows on the left hand side. Then, to enable an automated determination of TLM port binding solely based on IP-XACT descriptions, we specify the following two guidelines when using IP-XACT to describe TLM components:

- For a TLM port that implements a communication interface, such as `TLM_port_2` in Figure 5, its IP-XACT description should declare all the interface classes on the inheritance paths down to this interface. Therefore, the IP-XACT for `TLM_port_2` includes `sc_IF_1`, `sc_IF_2`, `sc_IF_3` `sc_IF_4`, and `sc_IF_6`, as it is indeed capable of providing all these communication services.

- For a TLM port that awaits a communication interface, for example `TLM_port_1`, we should declare in IP-XACT just the TLM interface that it expects for binding, here the `sc_IF_2`.

Based on this rule, the compatibility between a pair of IP-XACT TLM ports should be decided by seeing whether the interface description in the requires-port is included in those from the provides-port. If the provided interfaces indeed include the required interface, a corresponding SystemC TLM binding can be safely generated between the two ports.

With such a fully automated flow, we obtain an engine for simulating and testing IP-XACT designs and their foreseeable mutants, which prepares us for defining the IP-XACT based mutation.

### B. Mutation Operators Definition

For a specific design language, mutation operators intend to introduce syntactically correct perturbations into an object, which may trigger its erroneous execution behavior. Now our target language is IP-XACT schema and the design object is a bus-centric SoC integration with multiple components. By examining the elements in IP-XACT schema that are related to components integration and suited for mutation, we have our first definition of IP-XACT mutation operators as listed in Table I.

<table>
<thead>
<tr>
<th>Mutation operator name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ParRep</code></td>
<td>Replace a parameter configuration with another valid value</td>
</tr>
<tr>
<td><code>ParIns</code></td>
<td>Insert a parameter configuration with a valid value</td>
</tr>
<tr>
<td><code>ParDel</code></td>
<td>Delete a configuration</td>
</tr>
<tr>
<td><code>BusDel</code></td>
<td>Delete a bus interconnection</td>
</tr>
<tr>
<td><code>AdhDel</code></td>
<td>Delete an ad-hoc connection</td>
</tr>
<tr>
<td><code>BAddrIncr</code></td>
<td>Increase the base address of a slave component</td>
</tr>
<tr>
<td><code>HAddrDecr</code></td>
<td>Lower the high address of a slave component</td>
</tr>
<tr>
<td><code>AddrExch</code></td>
<td>Exchange the address spaces of two memory-mapped slave components</td>
</tr>
<tr>
<td><code>PortMapDel</code></td>
<td>Delete in a bus interface a port mapping from component physical-port to bus logical-port</td>
</tr>
</tbody>
</table>

The first operator `ParRep` uses another valid value, for example a pre-defined `choice` in IP-XACT, to replace an existing parameter configuration. Operator `ParIns` inserts into the design a configuration for some parameter. The replacement or insertion value can be chosen randomly. A third operator called `ParDel` deletes a configuration, so that the default value of this parameter takes into effect.

2) **Connection deletion operators:** Designers can omit some connections between components. The mutation operators in this class model such errors and delete completely a connection description. In IP-XACT design, we have two kinds of component interconnections. One is the connections through pre-defined bus interfaces and another one is ad-hoc connections, i.e. not through any bus protocol. Operator `BusDel` operates on the former and `AdhDel` operates on the latter.

3) **Memory-maps modification operators:** This class of operators introduces deviations on the address spaces of slave components from their original configurations, which makes the testing software have a wrong view of the hardware system. With erroneous interaction between software and hardware, it may further lead to a wrong behavior of the system and a negative test verdict, if the testing software is comprehensive enough.
Operator $B\text{Addr}Incr$ increases the base address of a slave component by a small value, with the caution that it should not exceed the upper address boundary of the component. Respectively, operator $B\text{Addr}Decr$ decreases a slave high address to a level not less than the base address. Another $\text{AddrExch}$ operator chooses two slave components and makes an exchange of their address spaces.

4) **Port-map deletion operator**: $\text{PortMapDel}$ is our last operator definition for introducing a missing port mapping into a bus connection. It deletes a $\text{portMap}$, as a pair of bus $\text{logicalPort}$ and component $\text{physicalPort}$, in the $\text{portMaps}$ of a bus interface. This $\text{portMap}$ can be, for example, the interrupt assertion or $RNW$ (read-not-write) selection for a peripheral component. By such a missing port mapping, wrong bus transactions are intended to be brought out in a system simulation.

C. Tool Implementation

To build a prototype tool that can be used for case studies, we leveraged the Eclipse Modeling Framework (EMF) [11] to implement an IP-XACT editor, which then incorporates both the code generation engine and mutants generator. EMF facilitates the building of tools and code generators with an automated process of transforming a structured meta-model to Java classes. XML schema is among the several meta-model formats that EMF supports. After generating Java classes based on the types and elements in an XML schema, EMF generates also an interface to the common Eclipse editing environment. In this way, we obtain an elementary Eclipse editor that is capable of viewing and editing the corresponding XML files and ready for functionality extensions.

Our investigation on IP-XACT uses its IEEE standard version. Therefore, we first fed this standard schema into the EMF process and obtained a basic editor for IP-XACT XMLs. Using the Java classes generated accurately from IP-XACT, the SystemC code generator was constructed, which is able to set up an IP repository when the editor starts and transform an IP design file as described previously.

As the Java classes are mapped from IP-XACT schema, the mutation operators can be realized by implementing correspondingly the manipulation schemes on the Java classes. At runtime, a mutation, i.e. error injection is conducted by modifying the Java objects that represent an IP-XACT XML file and saving the modified objects as another XML file to be the mutant.

IV. EXPERIMENTAL RESULTS

We have applied the above defined IP-XACT mutation testing in several experiments with a TLM IP library for CoreConnect SoC modeling. This library from IBM provides a wide range of TLM components for not only functional but also power modeling, including the Processor Local Bus (PLB) and On-chip Peripheral Bus (OPB) for CoreConnect architecture, a PowerPC ISS that connects to the PLB, DMA and interrupt controllers, DDR memory model, peripherals like a UART, etc. In the library, we find also several example system designs with accompanying tests, which became the objects of our experiments. The main purpose of these experiments is to show that our IP-XACT-based mutation can serve an effective coverage metric to reveal the quality shortage of SoC system tests.

To have IP-XACT based design flows, we at first packaged the TLM components by IP-XACT component descriptions. Then for each example design in the library, we created an IP-XACT design description that represents exactly the same components integration as the original design.

For each design, mutants are generated by applying the mutation operators through the Eclipse editor. Also in the editor, all the original designs and their mutants are fed into the code generator to generate SystemC simulation models, which are then simulated with the original tests. Table II reports the number of generated mutants and the killing coverage.

<table>
<thead>
<tr>
<th>Example design</th>
<th>Mutants generated</th>
<th>Mutants killed</th>
<th>Killing ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>43</td>
<td>37</td>
<td>86.0%</td>
</tr>
<tr>
<td>2</td>
<td>69</td>
<td>44</td>
<td>63.8%</td>
</tr>
<tr>
<td>3</td>
<td>52</td>
<td>38</td>
<td>73.1%</td>
</tr>
<tr>
<td>4</td>
<td>114</td>
<td>71</td>
<td>62.3%</td>
</tr>
</tbody>
</table>

The first example is a simple design with a PLB bus component and its arbiter, a memory component, and a CPU model with several read/write threads as tests. On its IP-XACT description we had 43 mutants generated, among which 37 were killed. The second design is a PowerPC ISS exercise by connecting the ISS model, a DDR memory controller and an I/O device model to PLB. 44 mutants were killed from the total 69 mutants generated on its IP-XACT representation. In the third design, PLB and DCR (Device Controller Register) bus are simultaneously used to connect the CPU and memory model, in order to test the coordinated transactions across PLB and DCR. It resulted in 52 mutants and a killing coverage of 73.1%. The last design is a larger integration where the ISS, interrupt controller, DDR memory model and UART are assembled with PLB, DCR, OPB buses. Also, a console model is integrated to connect the software on ISS to outside. A total of 114 mutants were generated by IP-XACT mutation and only 71 of them turned out be killed.

The un-killed mutants reveal exactly the weakness and holes of the tests, which we should not ignore. To promote the mild coverage scores, extra effort is required to improve the tests. In this respect, we view the IP-XACT mutation definition as an effective coverage metric on the verification quality of system level SoC designs.

V. RELATED WORK

Towards better quality measurement of simulation-based functional verification, researchers have been trying to invent complementary and more advanced metrics to overcome the shallowness of basic code coverage metrics such as statement coverage. In [21, 22], an observability-based coverage is
defined which is comparable to mutation-testing based coverage. Both techniques share a common feature: the examination of test data ability to trigger, propagate and observe some error effects. In this observability-based coverage, so-called tags are attached to assignment statements to model possible design errors. During simulation, specific calculus is used to determine the propagation of tag effects to observation points. However, tags are only a homogeneous approximation of potential error effects. In contrast, mutation-modeled errors by mutation operator definition are able to cover a wide range of typical design errors on any design language. Later, tests generation procedures aiming at this observability-based coverage are also discussed [23].

[6] is early work that applies mutation testing on hardware design verification. Instead of implementing their own mutation operators on an HDL, they conduct the mutation on a VHDL design by translating it to software program and feeding it to a software mutation testing tool Mothra [15]. Tests are also generated in the software mutation tool and further applied for both design verification and manufacturing testing. It is restricted by the nature of the software mutation tool as unit-level verification. In [18, 19, 20], automated test data generation methods are developed targeting RTL or behavioral VHDL mutants. They work mainly on individual component designs. A genetic algorithm is employed in [18] and a constrained random verification methodology is adopted in [19]. [20] relies on a search-based test generation algorithm to kill stubborn mutants.

General issues concerning system level mutation testing have been similarly discussed in software domains [17]. There, system components are interacting Java objects. The mutation operators are defined in three groups, configuration, interaction, and graphical user interface. Operators related to configuration are alike actions such as replacement or exchange of parameter values. Differently, interaction operators deal with method calls and the third group is not typically relevant for hardware designs. It is also identified as an issue how to determine the killing of a system mutant, for which a variant called flexible weak mutation is developed.

As TLM becomes the standard way for ESL modeling, [7] proposes the TLM mutation testing. First, the standard TLM communication primitives are formalized using the extended finite state machine (EFSM) model. Then, they are able to leverage the existing transition fault model for FSMs on the TLM EFSMs to define mutation operators, which reflects the typical design errors with TLM.

Aiming at IP-based design flows, we take SoC system level to be the component integration phase, which can have TLM or RTL components. We built the system level mutation testing on the standard IP-reuse language IP-XACT, which has not been exploited before this work.

VI. CONCLUSION

In this paper, we presented our effort to define and implement with IP-XACT the system level mutation testing for SoC verification. We explained what the major requirements of IP-XACT mutation are and how we met them. By using a SystemC code generator as the simulation engine for IP-XACT, we are able to cover the simulation of both RTL and TLM component integrations, or mixes of them. The provision of a Makefile makes the mutation testing process more convenient. With IP-XACT, our definition of mutation operators maintains a focus on the integration, configuration and interaction of IP components. Also with this focus, the resulted mutants retain a manageable amount. For the tool implementation, the use of EMF automatically guarantees a total conformance to the IP-XACT standard schema.

Results from several experiments show that this IP-XACT mutation can effectively qualify simulation tests. Indeed, it inherits the typically high computation cost of mutation testing, as each system mutant needs to be simulated separately. Here we have even the extra cost of generating the SystemC simulation models of mutants. However, we view the expense worth it. With increasingly larger SoC integrations, it is important that we now adopt a systematic metric to measure the system verification process.

As future work to reduce the mutation cost, we may consider implementing a mutation schema by transforming all IP-XACT mutants of a system into a single SystemC model in our code generator. By experiments, we can further identify those mutation operators that are more effective in qualifying tests and eliminating the redundant ones. Possible equivalent mutants, which are not discussed in this work, can also be investigated.

Other plans include the extension of system-level mutation testing to cover both the hardware and embedded software of a system at the same time and the development of a facilitating procedure for automated test data generation.

REFERENCES


[5] A methodology for the verification of a system on chip, 1999


