1 Background

In the context of IP-based SoC development, IP packaging is understood as the assembling of pre-verified IP components into an integrated design for verification or synthesis. This integration includes mainly the instantiation, interconnection, and the corresponding configurations of components. In this context, IP-XACT is the IEEE Std. 1685 language/meta-model [1] for describing IP reuse data and SoC system assembly instances. It defines the meta-model as a set of XML schema.

During the last years, mutation analysis has been considered as an adequate quality metrics to measure the simulation-based verification and validation processes [2]. As such mutation analysis is language based. It models typical, potential design errors that are called mutation operators based on the syntax of the language under consideration. This derives the basic effectiveness of mutation analysis as those errors should be detected by any capable testbench.

We have defined an IP-XACT mutation analysis framework to systematically manage and enhance the verification quality of SoC designs.

1.1 IP-XACT standard

The purpose of IP-XACT [1] 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 built as shown in Figure 1.

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 (Transaction level Modeling) or wire for RTL (Register Transfer Level). This makes IP-XACT to 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 sub-system 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.

Additionally, 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.

2 IP-XACT-to-SystemC Model Generator for IP-XACT Based Mutation Analysis

We face two challenges for this goal. Since IP-XACT designs are in the form of XML files and XML is not naturally executable, to simulate an IP-XACT SoC design we need at first an execution engine for IP-XACT. It enables us to verify an IP-XACT design and debug its errors, which further gives the possibility of adding mutation analysis in a simulation-based IP-XACT verification flow. Our way of creating this IP-XACT simulation engine is to define a set of synthesis rules that map any IP-XACT design to another system model that is simulatable. The target language that we choose for this transformation is SystemC. The mapping rules imply a SystemC model generator from IP-XACT, which we also implemented as an Eclipse tool to obtain an experiment basis for IP-XACT mutation analysis.
Second, a key to the creation of any new mutation analysis metric is the definition of a mutation operators set on the target language. Here, these are the errors that we can implant into IP-XACT XML designs. Typical syntactic changes should be collected and classified based on IP-XACT XML schema to mimic representative and important errors that one can make with IP-XACT design.

Figure 2 shows an overview of our efforts to meet these two challenges towards an IP-XACT based SoC verification quality. Once both is available, the SystemC simulation and a set of mutation operators for IP-XACT, we can seek the bugs of a SoC design with functional tests and measure the adequacy of these tests by their ability to kill system mutants. An example of mutation operators is replacement of an existing parameter configuration. Then, each measurement of mutants is to see whether it simulates differently from the original SoC design. For a short introduction and the definition of more mutation operators we refer to [3].

Figure 4 shows the working flow of the IP-XACT-to-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.

- In addition to 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.

Figure 2. IP-XACT mutation analysis framework.
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 the 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, the 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 by a simulation tool like QuestaSim from Mentor Graphics, for instance, is that it is convenient to get a co-simulation of SystemC and other HDLs and 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 straightforward. In contrast, 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 proposal for automatic checking of IP-XACT TLM compatibility.

![Figure 3. IP-XACT-to-SystemC model generation.](image-url)
First, we note that SystemC interface classes for TLM communication can be considered as a non-private inheritance tree starting from `sc_interface` as Fig. 4 shows it 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 4, 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.

We implemented a prototype tool that can be used for case studies. We applied the Eclipse Modeling Framework (EMF) [4] to implement an IP-XACT editor, which then incorporates both, the code generation engine and the mutant 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 multiple meta-model formats that EMF supports. We first fed the IEEE standard version of IP-XACT 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.

3 Conclusion

This article presents an IP-XACT-to-SystemC model generator and, in particular, outlines how it serves the simulation engine of an IP-XACT based mutation analysis framework. 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 combinations of them. The generated Makefile makes the mutation testing process even more efficient. With IP-XACT, our definition of mutation operators maintains a focus on the integration, configuration and interaction of IP components. For the tool implementation, the use of EMF guarantees a total conformance to the IP-XACT standard schema.

References


