Using a Formal Model to Improve Verification of a Cache-Coherent System-on-Chip

Abderahman Kriouile[1,2,3] and Wendelin Serwe[2,3]

[1] STMicroelectronics, 12, rue Jules Horowitz, BP 217, 38019 Grenoble, France
[2] Inria
[3] Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France

Abstract. In this paper we report about a case study on the functional verification of a System on Chip (SoC) with a formal system-level model. Our approach improves industrial simulation-based verification techniques in two aspects. First, we suggest to use the formal model to assess the sanity of an interface verification unit. Second, we present a two-step approach to generate clever semi-directed test cases from temporal logic properties: model-based testing tools of the CADP toolbox generate system-level abstract test cases, which are then refined with a commercial Coverage-Directed Test Generation tool into interface-level concrete test cases that can be executed at RTL level. Applied to an AMBA 4 ACE-based cache-coherent SoC, we found that our approach helps in the transition from interface-level to system-level verification, facilitates the validation of system-level properties, and enables early detection of bugs in both the SoC and the commercial test-bench.

1 Introduction

Due to increasing design complexity, functional verification continues to be one of the most expensive and time-consuming steps in a typical System-on-Chip (SoC) design flow. In practice, most widely used techniques are based on extensive simulation due to the related flexibility. However, the success of simulation-based verification, both in terms of total effort spent and final verification coverage achieved, depends heavily on the quality of the tests executed during simulation. Generating tests to achieve high coverage for complex designs has always been a challenging problem. In general, more constrained and less random tests reduce the overall validation effort, because the same verification coverage can be achieved with fewer and shorter tests [19]. We distinguish in this paper three types of test generation techniques with decreasing degree of randomness: fully random, constrained-random [26], and fully specified tests, hereafter called directed tests, i.e., without randomization.

Fully random tests are the easiest to automate, but require long simulation runs to obtain a reasonable coverage. In many situations, directed tests are the only tests that can thoroughly verify corner cases and important features of a design [2, 15]. However, because directed tests are mostly written manually, it is impractical to generate a comprehensive set of directed tests to achieve a

coverage goal [19]. Automatic test generation using model checking is one of the most promising approaches for directed test generation. However, for large designs model checking rapidly faces state explosion, when considering hardware protocols in all their details.

Constrained-random testing uses constraint solvers to select tests satisfying a specified set of constraints; non-specified details are then filled in by randomization. The automation of the feedback from coverage analysis to constrained-random test generation led to *coverage-directed test generation* (CDTG) [22], which dynamically analyzes coverage results and automatically adapts the randomized test generation process to improve the coverage. CDTG is guided by different coverage metrics, such as state coverage and transition coverage [2] and shows various degrees of success [11]. For instance, it succeeds to achieve coverage goals for interface hardware protocols, but reaches its limits for complex system-level protocols, such as system-level cache coherency. Achieving good coverage for these recent protocols is a new challenge in the development of industrial test benches and calls for more directed and less random tests.

This paper is about the application of formal methods to improve the functional verification of a heterogeneous cache-coherent SoC for a commercial set-top-box supporting multiple Ultra HD flows on a single chip currently under development at STMicroelectronics. We use an extension of a previously developed system-level formal model of a cache-coherent SoC [16] and take advantage of equivalence checking, model checking, and test generation facilities offered by the CADP toolbox[4] [8].

The two principal contributions of this paper are the following.

1. A way to assess the sanity of an industrial *interface verification unit* (ivunit), consisting of a set of behaviors to cover. In our study, we focus on the complex behaviors expressed by so-called *checks* of a commercial ivunit.
2. A two-step approach to use model-based testing to generate clever semi-directed system level test cases from temporal logic properties. We use CADP to generate directed "abstract" system-level test cases, which are then refined with commercial CDTG tool into interface-level "concrete" test cases that can be executed at RTL level. Those tests concern system-level properties in the sense that several interfaces are activated. We propose the notion of a *system verification unit* (svunit) to measure the coverage and verdicts of system-level properties.

The rest of the paper is organized as follows: Section 2 presents related work. Section 3 recalls the main aspects of the considered SoC and its formal model [16]. Section 4 presents contribution 1 by describing the validation of an industrial ivunit using equivalence checking. Section 5 details contribution 2 by proposing our test generation methodology based on counterexamples generated by model checking. Section 6 presents experimental results and the industrial impact of our work. Finally, Section 7 concludes the paper.

[4] http://cadp.inria.fr/

2 Related Work

For instance, the specification-based test generation technique [21] uses a formal model to generate directed tests. Solutions based on model checking techniques are promising for functional verification and test generation for reasonably complex systems [10]. However, it is unrealistic to assume that a complete detailed model of a large SoC is tractable by a model checker. We address this issue by relying on a system-level model, abstracting from all irrelevant details. In this way, we succeed to model a complex industrial SoC and to extract relevant scenarios by model checking. The above approaches transform counterexamples produced by the model checker into test cases. In our approach, we use the counterexamples to produce smaller interesting configurations of the model that still do not satisfy a given property. We generate test cases from these smaller models, thus avoiding combinatorial explosion in many cases.

In the literature, it has already been proposed to mix model-based techniques and coverage-directed techniques. Coverage-directed techniques were used in property learning [4] ("reuse learned knowledge from one core to another"), which we do not use, and that relies on SAT-based BMC (whereas CADP implements totally different verification techniques). Some of those techniques [21] focus on homogeneous multicore architectures (exploiting symmetry between processors to reduce verification complexity), and only suggest how this could be extended to heterogeneous architectures (by grouping IPs into homogeneous groups to be studied separately). On the contrary, our approach was designed for heterogeneous SoCs and makes no symmetry assumption. Also, most of those techniques [4, 5, 21] remain at system level (SystemC-TLM level), whereas our approach starts from system level and goes down to RTL level.

Over the last two decades, the CADP toolbox has been used for verifying numerous complex hardware designs, including Bull supercomputers, STMicroelectronics multiprocessor architectures, Networks-on-Chip (CEA/Leti and University of Utah), and various asynchronous circuits. In this paper, we present an application of the latest languages and tools of CADP. Using the new generation formal language LNT [3] to describe the system and also the test purposes greatly facilitates the testing of complex behaviors. Similarly, the MCL language [17] provides a convenient way to express complex data-based temporal properties. We use a new prototype tool to generate tests on the fly. Finally, instead of a homogeneous system as in [9, 14], we study the less symmetric and thus more complex case of an heterogeneous SoC.

3 Formal Model of an AMBA 4 ACE based SoC

The recent AMBA 4 ACE (AXI Coherency Extension) protocol [1, 23], proposed by ARM, extends the AMBA 3 AXI protocol in order to support system-level cache coherency in SoCs. AXI defines communication at interface-level between a pair of master/slave ports, which are connected by several read and write channels (`AR`, `R`, `AW`, `W`, `B`). AXI defines two transactions *Read* and *Write*, each of

which consists of several transfers; each transfer is executed on an AXI channel. Thanks to the encapsulation mechanisms of AXI, a transfer on a channel can be considered to be atomic at higher levels.

ACE introduces system-level requirements on transaction ordering, adds coherency channels (`AC`, `CR`, `CD`), enriches existing channels with new coherency parameters (i.e., PassDirty, IsShared), and defines cache line states and several transactions (i.e., coherent transactions, cache maintenance transactions, memory update transactions, etc.). ACE introduces heterogeneity by defining two types of coherent masters: those with a cache are called *ACE masters*, and those without caches are called *ACE-Lite masters*. The latter can access data in the caches of the former, avoiding access to memory, which improves performance.

Example 1. The *ReadOnce* transaction is a coherent transaction (used in particular by an ACE-Lite master), which obtains the current contents of a memory line without keeping a copy into the cache. If an ACE-Lite master sends a *ReadOnce* transfer on the `AR` channel. The CCI then sends snoop requests to all ACE masters on the `AC` channels. Each ACE master answers on the `CR` channel with a Boolean indicating whether the data is in its cache, and a Boolean indicating if the master passes the responsibility of writing back the data in the memory (`PassDirty`). If the data is available, the ACE master sends also a data transfer on the `CD` channel. If none of the master has the data available, it is taken from the main memory. The CCI forwards the data to the ACE-Lite master using the `R` channel, to complete the transaction. If one of the ACE masters passed the responsibility to write back the data, the CCI must initiate a memory update, because an ACE-Lite master cannot take this responsibility.

We use an extension of a previously developed formal model [16] (about 3400 lines of LNT code) of an ACE-based SoC[5], consisting of a *cache-coherent interconnect* (CCI) connected to a non-cache-coherent Network-on-Chip (NoC). Figure 1 shows the overall architecture of the model in the configuration used in the present paper. Following the ARM® big.LITTLE™ solution [20], the two ACE masters are one big (powerful) and one little (lower-power) processor, enabling to dynamically adapt to changing computation load. The ACE-Lite master is a *Graphical Processing Unit* (GPU) that can access the caches of both processors. All three masters access the main memory through a non-cache-coherent NoC. The ACE protocol supports the coherency of data among the processors. Our formal model focuses on the cache-coherent part of the SoC.

The ACE specification contains some *global requirements*. Indeed, the ACE protocol does not guarantee system level cache coherency, but just provides support for it. Coherency has to be ensured by proprietary additional mechanisms on each implementation of a CCI. We model these global requirements in a constraint-oriented style by adding observer processes that prohibit incorrect executions. By omitting those observers, we obtain an *unconstrained model*, for which the global requirements are not necessarily satisfied.

[5] A large Petri net derived from our LNT model is available as Model Checking Contest 2014 benchmark (http://mcc.lip6.fr/pdf/ARMCacheCoherence-form.pdf).

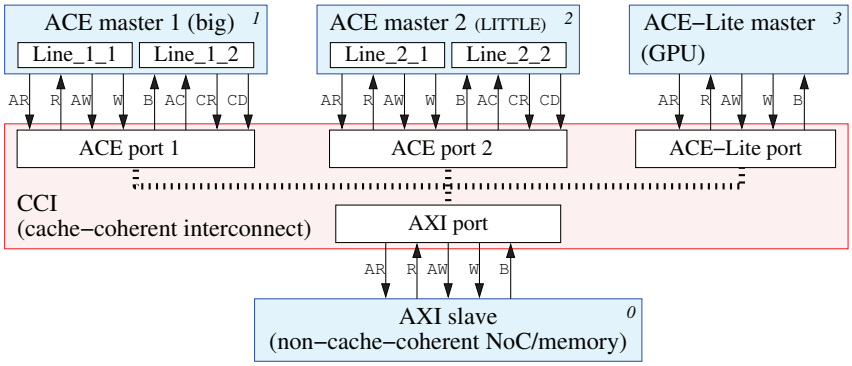

Fig. 1. Model architecture

A crucial feature of our formal model is that it is parametrized, in particular by the set of forbidden ACE transactions, the number of ACE masters, ACE-Lite masters, and cache lines per ACE master. Among the three masters, at most two initiate transactions at the same time. We will vary essentially one parameter, which is the set of forbidden ACE transactions; we refer to an instance of the model as `Model(`$\mathcal{F}$`)`, where $\mathcal{F}$ is the set of forbidden ACE transactions; thus, `Model(`$\emptyset$`)` corresponds to the complete, unconstrained model.

4 Sanity of a Formal Check List

Industrial CDTG test benches are based on a so-called *verification plan*, i.e., a list of all behaviors to be covered by tests on the *Design Under Verification* (DUV). The coverage of the verification plan is collected to measure test progression.[6] In our work, we focus on the formal *checks*, which are grouped in so-called *interface verification units* (ivunit). Each check is an event sequence, e.g., expressed in *Property Specification Language* (PSL) [12]. Covering a check consists in activating the check and finishing correctly the specified sequence. Activating a check means to detect the first event of the sequence. It is a failure if a check is activated and not correctly finished.

In this section, we report about the use of our formal model to validate a commercial ivunit. To this end, we encode each check of the ivunit as a *Labeled Transition System* (LTS) (by means of an LNT model) and use equivalence checking techniques (hiding, minimization, and comparison operations on LTSs).

In fact, the ivunit considers only a single interface (i.e., a single master/slave pair), whereas the formal model describes the complete SoC. To obtain the LTS of the interface between ACE master 1 (big) and the CCI (upper left part of Fig. 1), we hide in the LTS of the whole system all labels except those of the

[6] There are two types of behaviors in a verification plan: simple behaviors, called *cover points* and complex behaviors, called (formal) *checks*.

selected interface and then minimize the resulting LTS according to *divergence-sensitive branching bisimulation* (divbranching) [25], which preserves the branching structure and livelocks (cycles of internal τ-transitions). Applying those steps reduces the LTS as generated from the model (498,197 states, 1,343,799 transitions) by two orders of magnitude (3,653 states, 8,924 transitions). We store the reduced LTS in a file named `interface.bcg`, where the extension `.bcg` stands for Binary Coded Graph, the compact binary format used to store LTSs in CADP.

We continue our study by identifying a subset of nine industrial checks (called C1 ... C9), which have a level of abstraction corresponding to our formal model. Then we verify that each check is an overapproximation of the model behavior. Last, we study if the list of checks covers all behaviors of the model.

4.1 Local Sanity of Each Check

We aim at verifying that each check is well specified. Because each check uses only a subset of interface channels, we generate a corresponding sub-interface by hiding all channels except those occurring in the check, and apply again divbranching reduction.

Example 2. Check C1 requires that the current read request address should not overlap with any of the outstanding write requests. C1 uses only three channels: address read (`AR`), address write (`AW`), and write response (`B`). Thus we obtain the corresponding sub-interface LTS (105 states, 474 transitions).

We verify that each sub-interface LTS is included in the corresponding check LTS modulo the preorder of the divbranching bisimulation. We conclude that the check is a correct overapproximation of the behavior of the subset of ACE channels.

4.2 Global Sanity of the List of Checks

To verify that the list of checks covers all the behaviors of the interface model, we compare the parallel composition of all the nine checks with the interface LTS. We use smart reduction [6] to automatically optimize the order of composing and minimizing the checks in the parallel composition: the complete composition process takes approximately five minutes. We express the parallel composition in SVL with an LNT-style parallel composition operation: each check is required to synchronize on all the gates (channels) it uses; synchronization is n-ary, i.e., *all* checks that have a given channel (e.g., `AR`) in their synchronization set (on the left of `->`) synchronize on the channel (e.g., C1, C2, C3, C5 all together synchronize on `AR`).

```
"all_checks.bcg" = smart divbranching reduction of
        par   AR, AW, B    -> "C1.lnt"
           || AR, AW, R, B -> "C2.lnt"
           || AR, R        -> "C3.lnt"
           || AW, B        -> "C4.lnt"
```

```
   || AR, AW, R, B -> "C5.lnt"
   || AW, B, CD    -> "C6.lnt"
   || AC, CD       -> "C7.lnt"
   || AC, CR       -> "C8.lnt"
   || R            -> "C9.lnt"
end par;
```

We compare the interface LTS and the checks LTS `all_checks.bcg` (11,773 states, 8,171,497 transitions) to verify if the interface LTS is included in the LTS `all_checks.bcg` modulo the preorder corresponding to the divbranching bisimulation. This verification fails, i.e., we detect a *missing check*; the counterexample (provided by CADP) shows a `W` label following an `AW` label.

According to the ACE specification, there must be the same number of `W`'s and `AW`'s. We express this constraint by a new check (`C10`), avoiding the use of counters, using asynchronous parallel composition (`AW || W`). Adding C10 to the parallel composition of the checks, yields a new LTS `all_checks_bis.bcg` (38,793 states, 27,200,587 transitions).

We compare `all_checks_bis.bcg` and `interface.bcg` and observe now that `interface.bcg` is included in `all_checks_bis.bcg` for divbranching bisimulation. Hence, the check list is now complete with respect to our formal model. Although the missing check could also be found manually by inspecting the list of channels in the checks (all channels but `W` are present), our approach has the additional benefits of illustrating the missing behavior and enabling to formally, and semi-automatically, establish the completeness of the check list.

5 From Temporal Logic Properties to Clever Test Cases

An interesting idea for the generation of directed tests is to focus on and derive tests from potential faults of the DUV [18]. For system-level protocols, in order to obtain a description of potential faults corresponding to the global requirements of the SoC, we suggest to use system-level properties together with a model containing faults. In our case, we use the unconstrained model (see Sec. 3). Applying the theory of conformance testing [24], we generate abstract test cases, which then have to be translated to the input language of a commercial CDTG solver to randomly complete interface-level details and finally to run the tests on the RTL test bench.

Because we found the generation of abstract test cases directly from the complete model to be impractical, we suggest to use information contained in counterexamples to select *interesting configurations* of the formal model, which still contain violations of the global requirements, and to extract abstract test cases from the selected configurations. Figure 2 gives an overview of our test generation flow.

5.1 System-Level Properties

We express properties in the *Model Checking Language* (MCL) [17], an extension of the modal μ-calculus with high-level operators to improve expressiveness and

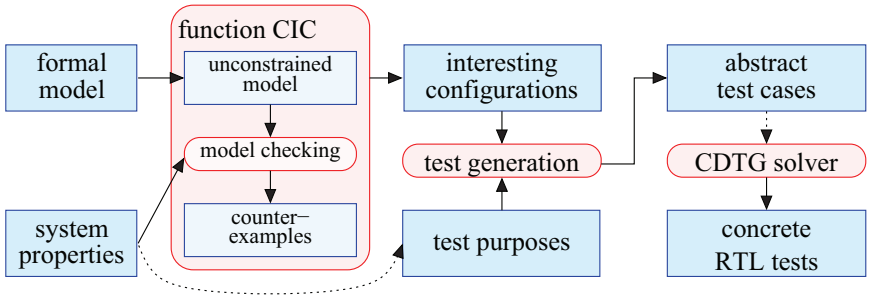

Fig. 2. Model-Based Test Generation flow

conciseness of formulæ. The main ingredients of MCL used in this paper are action patterns extracting data values from LTS transition labels, and modalities on transition sequences described by extended regular expressions. MCL formulæ are verified on the fly using the EVALUATOR 4.0 model checker of CADP.

Among the properties we considered[7], only the following two do not hold for the unconstrained model.

Data Integrity. The following property [16, φ_5], enforces correct order of write operations to the shared memory:

```
[ true * .
   {W !"WRITEBACK" ?c:Nat ?l:Nat ?d:Nat}.          (* memory update    *)
   (not {W !"WRITEBACK" !0 !l !d !c})*.
   {W !"WRITEBACK" !0 !l !d !c}.                   (* update written   *)
   (  (not {AC ... !c ?any of Nat !l}) and
      (not {W ?any of String !0 !l ?any of Nat ...}))*.
   {W ?any of String !0 !l ?h:Nat ... where h<>d}  (* different update *)
] false
```

The second line of this property matches an action corresponding to a rendezvous on gate `W` with four offers: the transaction ("`WRITEBACK`", i.e. a memory update), the initiating master (the id of which is stored in variable `c`), the memory line (the address of which is stored in variable `l`), and the data to be written (which is stored in variable `d`). When this update is effectively written to memory (2nd action on gate `W`), with as second offer the port number `0`, i.e., the memory, the property forbids (last action on gate `W`) a data `h` different from `d` to be written to the same memory line `l` without previously receiving a snoop request (gate `AC`) concerning line `l`.

[7] We considered several properties, such as: absence of deadlocks, absence of livelocks, complete execution of read and write transactions, data integrity, and coherency of ACE-states and parameters of ACE transactions. All these properties are satisfied by our constrained model.

```
function CIC (φ: Property, F: Set of Transaction): Set of (Set of Transaction) is
    if Model(F) ⊨ φ then
        return ∅
    else
        let Δ be a minimal-depth counterexample ;
        result := ∅ ;
        for each transaction T occuring in Δ do
            result := result ∪ CIC (φ, F ∪ {T} )
        end for ;
        if result = ∅ then result := { F } end if ;
        return result
    end if
end function
```

Fig. 3. Function CIC to compute a set of interesting configurations containing faults

Unique Dirty Coherency. To verify the coherency of the ACE states of all the caches of the system, we translated the state-based properties to action-based properties, using information about the ACE state added to transactions issued by cache lines. The following property [16, φ_3] requires that if a cache line is in the state `ACE_UD` (the cache line is unique and modified), then as long as the line does not change its status, all other cache lines containing the same memory line must be in the state `ACE_I` (the cache line is invalid)[8]:

```
[ true * .
   {?Ch:String ?op:String ?m1:Nat ?indM:Nat !"ACE_UD"} .
   (not({?Ch:String !m1 !indM ?s:String where s<>"ACE_UD"}))* .
   {?Ch:String ?op:String ?m2:Nat !indM ?s:String
    where (m2<>m1) and ace_state(s) and (s<>"ACE_I")}
] false
```

5.2 Computation of Interesting Configurations Containing Faults

Counterexamples of a desired property provide interesting scenarios to test corner cases. To improve test coverage, it is interesting to have as many different counterexamples as possible. However, on-the-fly model checking provides at most *one* counterexample for each property φ and configuration of the model, because the model checker stops as soon as it detects a violation of the property. Therefore, we take advantage of the parametrization of our formal model, by varying the set $\mathcal{F}$ of forbidden ACE transactions, to compute with the recursive function `CIC` (*compute interesting configurations*) shown in Fig. 3 a comprehensive set of *interesting configurations* of the `Model(`$\mathcal{F}$`)` containing faults.

Initially, all fifteen ACE transactions are allowed, i.e., we call `CIC(`φ`,`$\emptyset$`)`. Function `CIC` proceeds as follows. First, we configure the model to exclude the transactions in $\mathcal{F}$, and model check property φ. If φ is not satisfied, the model

[8] `ace_state(s)` is a macro predicate that holds iff the string `s` is an ACE state.

checker produces a counterexample Δ. We use the breadth-first search algorithm of EVALUATOR 4.0 to produce a counterexample of minimal depth, and avoid spurious actions in the counterexample. For each transaction T occurring in Δ, we call `CIC` recursively, deactivating T in addition to $\mathcal{F}$. Function `CIC` terminates, because the parameter $\mathcal{F}$ has an upper bound (the set of all transactions) and it strictly increases for each recursive call.

The set of interesting configurations $\{\texttt{Model}(\mathcal{F}_1),\ldots,\texttt{Model}(\mathcal{F}_n)\}$ corresponding to the set $\{\mathcal{F}_1, ..., \mathcal{F}_n\}$ computed by `CIC` has the following property: a configuration $\texttt{Model}(\mathcal{F}')$ does not satisfy the property φ if and only if $\mathcal{F}'$ is smaller than or equal to at least one combination $\mathcal{F}_i$.

We applied `CIC` to the two properties that were invalid on the unconstrained model (see Sec. 5.1). Altogether, Data Integrity yields 21 interesting configurations (14 from an architecture with two ACE masters initiating transactions, and 7 from an architecture with one ACE-Lite master and one ACE master initiating transactions) and Unique Dirty Coherency yields 18 interesting configurations from an architecture with two ACE masters initiating transactions (with only one ACE master, i.e., a single cache, Unique Dirty Coherency holds trivially).

5.3 Abstract Test Generation

We aim at generating as many tests as possible leading to invalidation of the property for each interesting configuration. We call those tests *negative tests*, because if a test succeeds, we detect a failure of the system; but if the system is correct, all tests will fail.

Our test generation approach is based on the theory of conformance testing [24], i.e., we compute from a specification of a system and a so-called *test purpose* [13] a set of abstract test cases. Intuitively, a test purpose is a means to characterize those states (called ACCEPT states) of the specification that should be reached during test execution. To prune the search space for test cases, the test purpose can also contain so-called REFUSE states: if such a state is reached while testing the DUV, the test is stopped and declared inconclusive. Technically, a test purpose is provided as an LTS, e.g., an LNT model. Thus we express the negation of each property as a test purpose in LNT.

Example 3. The LNT code for the test purpose corresponding to the Unique Dirty Coherency is shown in Fig. 4. After an outgoing action (gates `AR`, `AW`, and `W`) from an ACE master `cpu1` with a cache state `ACE_UD` (Unique Dirty), it monitors all outgoing actions of all ACE masters. If a different ACE master `cpu2` has an `ACE_UD` state we `ACCEPT` the test (a coherency error has been detected). If `cpu1` performs another action with a state other than `ACE_UD`, we `REFUSE` the test (the test is inconclusive).

We use two newly developed prototype tools for test generation. A first tool takes as input a model and a test purpose (both in LNT), and produces a *Complete Test Graph* (CTG), i.e., an intermediate LTS containing all information to extract (all) abstract test cases. We use a second tool to extract a set of abstract

```
process main [AR, AW, CR, ACCEPT, REFUSE: any] is
var cpu1, cpu2: INDEX_CPU, state: ACE_state_t in
  select
     AR (?any, ?cpu1, 1, ACE_UD)
  [] AW (?any, ?cpu1, 1, ACE_UD)
  [] CR (?any, ?any, ?cpu1, 1, ?any, ?any, ACE_UD)
  end select; -- cache line of cpu1 has unique dirty status
  select
     select
        AR (?any, ?cpu2, 1, ACE_UD) where (cpu2<>cpu1)
     [] AW (?any, ?cpu2, 1, ACE_UD) where (cpu2<>cpu1)
     [] CR (?any, ?any, ?cpu2, 1, ?any, ?any, ACE_UD) where (cpu2<>cpu1)
     end select;
     ACCEPT -- cache lines of both cpus have unique dirty status
  []
     select
        AR (?any, cpu1, 1, ?state) where (state<>ACE_UD)
     [] AW (?any, cpu1, 1, ?state) where (state<>ACE_UD)
     [] CR (?any, ?any, cpu1, 1, ?any, ?any, ?state) where (state<>ACE_UD)
     end select;
     REFUSE -- cache line of cpu1 no longer has unique dirty status
  end select
end var end process
```

Fig. 4. Unique Dirty Coherency test purpose described in LNT

test cases from the CTG. These test cases are abstract in the sense that they are system-level automata generated from the model. Thus, those abstract test cases have to be translated to the input language of the commercial coverage-based solver to randomly complete the interface-level details and to run the tests on the RTL test bench.

By extracting all test cases from each CTG and running each test case on the industrial test bench, we obtain a locally intensive test around corner cases specified by the global system-level properties.

Table 1 summarizes the results of our generation of abstract test cases for a test purpose encoding the negation of a property φ. The first two columns describe the property and the architecture. Columns 3 to 5 report the size of the global CTG (produced from the unconstrained model) and the time to extract test cases. The remaining columns give information about our approach based on individual CTGs (produced from the interesting configurations): column 6 presents the number of CTGs, each of which is extracted from an interesting configuration, columns 7 to 10 report the size of the largest and the smallest CTG, and the last column gives the time to extract test cases from all the individual CTGs. We see that the approach based on individual CTGs is much more efficient than the extraction of test cases directly from the global CTG, for which the extraction of test cases does not finish in half a year. Also our

Table 1. Experimental test case extraction results

	masters	global CTG states	trans.	extr. time	nb. of largest CTG CTGs	states	trans.	smallest CTG states	trans.	extr. time
φ_3	2ACE	6,402	14,323	>½ y	18	903	1,957	274	543	≃7h
φ_5	2ACE	23,032	48,543	>½ y	14	462	888	59	107	<1h
	1ACE/1Lite	2,815	7,071	>½ y	7	193	394	59	107	<1h

approach reduces the size of the largest CTG by a factor of 7 for φ_3 (Unique Dirty Coherency), a factor of 14 for φ_5 (Data Integrity) in the case of the architecture with one ACE master and one ACE-Lite master initiating transactions, and a factor of 49 for for φ_5 in the case of the architecture with two ACE masters.

6 Industrial Results and Impact

Our formal model is used inside STMicroelectronics as a reference in discussions with verification engineers and interconnect architects. It helps to understand the new aspects introduced by ACE and to define the verification strategy. In this context, the OCIS interactive step-by-step simulator with backtracking of CADP is found useful for exhibiting execution scenarios of interest.

We also used OCIS to extract the list of possible transaction initiations for each correct initial state of the system. A correct initial state of the system is a correct combination of initial ACE states of the caches. For example, if a memory line exists initially in two different caches, the state of these caches cannot be `ACE_UD` for both caches. So doing, we produce in less than one day 296 simple protocol tests, each of which consists of one single ACE transaction, from request to response, including all triggered snoop requests, if any.

Using some of the counterexamples generated during the computation of the interesting configurations (cf. Sec. 5.2), we produced also ten complex protocol tests containing concurrency between different ACE transactions.

6.1 Making the Test Bench Ready for System-Level Verification

The original test libraries developed by the verification engineers are interface tests. With a not so good coverage of system, new tests describing system scenarios are necessary. Because, system requirements cannot be verified on a single ivunit separately, we complete the verification infrastructure and introduce the notion of a *system verification unit* (svunit) connected to all ivunits, enabling to combine behaviors of different interfaces in order to validate system-level requirements. For the considered SoC, we defined an svunit consisting of 56 PSL sequences, 56 PSL basic cover points, and 36 PSL checks. This enables to verify on the RTL test bench that each coherent transaction produces the corresponding snoop transactions, and that each snoop transaction eventually receives a response from the snooped master.

Further modifications of the test bench are required to enable the execution of the concrete test cases derived from our abstract test cases. In particular, it is necessary to control the order of events. First, we added more synchronizations between different *Verification Intellectual Property* (VIP) events to enforce the desired order of the events. Second, we added speed-up randomization: by default the speed of a master for each of its channels is completely random. To express that a master is faster than another one or to enforce an order between two concurrent actions of a same master, we specify speed-up ranges (e.g., fast, slow, or very slow). So doing, the speed-up remains random, but in a limited range, ensuring the desired order.

6.2 Industrial Results

During the implementation of our abstract test cases on top of commercial VIPs, we detected ten bugs in those VIPs. This enabled the CAD supplier to correct the bugs before the use of these VIPs became critical in the development path of STMicroelectronics.

Because the VIPs and the coverage lists are provided by the same CAD supplier, some verification gaps may not be detected. In fact, the same misinterpretation of the ACE specification may find its way into both the VIPs and the coverage lists. Working with a different approach led us to validate the industrial checks (provided in the ivunits), and thanks to our directed tests we detected unverified behaviors.

In October 2014, STMicroelectronics architects detected a limitation in the IP implementation of the CCI. This limitation manifests in a subset of the counterexamples for the data integrity property we verified 20 months before. Precisely, when the CCI initiates a memory update (e.g., see Example 1), some parameters of this update are set to fixed values possibly loosing some important information, and disturbing the ACE-Lite flow in the non-coherent part of the SoC. This limitation corresponds to a gap that we have detected on the commercial VIPs one year before, when we started experimenting with the translation of abstract to concrete test cases. Our method for computing interesting faulty configurations (see Sec. 5.2) enabled us to provide all the scenarios triggering this limitation. In addition, we wrote new PSL checks to detect those corner cases. We should notice that our 306 extracted tests trigger those checks 16 times, whereas the other tests of the STMicroelectronics test library never trigger these checks.

Our generated tests have direct impact on the development flow of an industrial SoC of STMicroelectronics. We observe that the coverage of the verification plan increased significantly[9] and that the coverage of the svunit part of the verification plan is complete (100%), i.e., all the aspects corresponding to system-level behaviors are tested.

[9] The coverage of the verification plan increased from 30% to 68%. Notice that 100% coverage is not achievable for the considered SoC, because the verification plan, as defined by the VIPs, includes some features of ACE (e.g., distributed virtual memory), which are handled by the VIPs, but are not used by the considered SoC.

7 Conclusion

We used a system-level formal model of a SoC to improve functional verification in several aspects. First, we studied the sanity of a list of industrial formal checks. We also verified principal system properties with an explicit-state model checker. Using models containing faults, we computed a comprehensive set of interesting configurations, which are then used to generate negative abstract tests. Those tests were translated into RTL level through a coverage-directed test generation platform, thus intensively test the system around corner cases.

Our approach capitalizes on existing environments while solving their limitations for system-level protocols. This had an impact on an industrial SoC in production: It helped to improve the test bench and to increase test coverage. In addition, our approach contributed to the maturation of commercial VIPs.

Currently, we work on automating manual parts of our approach, in particular the translation of abstract test cases to the inputs of a CDTG solver. Given the success of our approach, it seems interesting to apply this approach to the system-level protocols in the next generation of SoCs.

Concerning reusability, our approach to assess the sanity of a check list against a formal model is akin to *crosschecking*, a technique widely used in the hardware community to improve confidence on the verification components. To apply our test generation approach, the formal model must be configurable, so as to violate a property. These preconditions seem acceptable, as we found modifying parts of the model (e.g., some data types) feasible using simple scripts, and the literature presents several techniques to automate the production of faulty models.

Acknowledgements. We are grateful to H. Garavel and R. Mateescu (Inria), G. Barthes and M. Zendri (STMicroelectronics) for their contributions and valuable remarks. We would also like to thank C. Chevallaz and G. Faux (STMicroelectronics) for helpful discussions, and the anonymous reviewers for their suggestions that contributed to improve the paper.

References

1. ARM. *AMBA AXI and ACE Protocol Specification*, Feb. 2013. version ARM IHI 0022E, http://infocenter.arm.com/help/topic/com.arm.doc.ihi0022e.
2. M. Benjamin, D. Geist, A. Hartman, G. Mas, and R. Smeets. A Study in Coverage-Driven Test Generation. In *Design Automation Conference*, pp. 970–975. IEEE, 1999.
3. D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, C. McKinty, V. Powazny, F. Lang, W. Serwe, and G. Smeding. Reference manual of the LNT to LOTOS translator (version 6.1). INRIA/VASY – INRIA/CONVECS, Dec. 2014.
4. M. Chen and P. Mishra. Property learning techniques for efficient generation of directed tests. *IEEE Transactions on Computers*, 60(6):852–864, 2011.
5. M. Chen, X. Qin, H.-M. Koo, and P. Mishra. *System-Level Validation: High-Level Modeling and Directed Test Generation Techniques*. Springer, 2013.

6. P. Crouzen and F. Lang. Smart reduction. In *FASE*, *LNCS* 6603, pp. 111–126. Springer. 2011.
7. H. Garavel and F. Lang. SVL: a Scripting Language for Compositional Verification. In *FORTE*, pp. 377–392. IFIP, 2001.
8. H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. *STTT*, 15(2):89–107, 2013.
9. H. Garavel, C. Viho, and M. Zendri. System design of a CC-NUMA multiprocessor architecture using formal specification, model-checking, co-simulation, and test generation. *STTT*, 3(3):314–331, 2001.
10. A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. *Software Engineering Notes*, 24:146–162, ACM, 1999.
11. O. Guzey and L.-C. Wang. Coverage-directed test generation through automatic constraint extraction. In *High Level Design Validation and Test Workshop*, pp. 151–158. IEEE, 2007.
12. IEEE standard for property Specification language (PSL). *IEEE Std 1850-2010*, pp. i –188, 2010. http://standards.ieee.org/findstds/standard/1850-2010.html.
13. C. Jard and T. Jéron. TGV: theory, principles and algorithms. *STTT*, 7(4):297–315, 2005.
14. H. Kahlouche, C. Viho, and M. Zendri. An industrial experiment in automatic generation of executable test suites for a cache coherency protocol. In *Testing of Communicating Systems*, pp. 211–226. Springer, 1998.
15. H.-M. Koo, P. Mishra, J. Bhadra, and M. Abadir. Directed micro-architectural test generation for an industrial processor: A case study. In *Microprocessor Test and Verification*, pp. 33–36. IEEE, 2006.
16. A. Kriouile and W. Serwe. Formal Analysis of the ACE Specification for Cache Coherent Systems-on-Chip. In *FMICS*, *LNCS* 8187, pp. 108–122. Springer, 2013.
17. R. Mateescu and D. Thivolle. A model checking language for concurrent value-passing systems. In *FM*, *LNCS* 5014, pp. 148–164. Springer, 2008.
18. D. A. Mathaikutty, S. K. Shukla, S. V. Kodakara, D. Lilja, and A. Dingankar. Design fault directed test generation for microprocessor validation. In *DATE*, pp. 1–6. IEEE, 2007.
19. P. Mishra and M. Chen. Efficient techniques for directed test generation using incremental satisfiability. In *VLSI Design*, pp. 65–70. IEEE, 2009.
20. A. Peter Greenhalgh. Big. LITTLE Processing with ARM Cortex™-A15 & Cortex-A7, 2011.
21. X. Qin and P. Mishra. Efficient directed test generation for validation of multicore architectures. In *Quality Electronic Design*, pp. 276–283. IEEE, 2011.
22. H. Shen, W. Wei, Y. Chen, B. Chen, and Q. Guo. Coverage directed test generation: Godson experience. In *Asian Test Symposium*, pp. 321–326. IEEE, 2008.
23. A. Stevens. *Introduction to AMBA 4 ACE*. ARM whitepaper, June 2011.
24. G. J. Tretmans. *A formal approach to conformance testing*. Twente University Press, 1992.
25. R. J. Van Glabbeek and W. P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, 1996.
26. J. Yuan, C. Pixley, A. Aziz, and K. Albin. A framework for constrained functional verification. In *Computer Aided Design*, pp. 142–145. IEEE, 2003.