Systematic Application of UCIS to Improve the Automation on Verification Closure

Christoph Kuznik, Marcio F. S. Oliveira, Gilles Bertrand Defo, Wolfgang Mueller

Faculty of Electrical Engineering,
Computer Science and Mathematics
University of Paderborn/C-LAB
D-33102 Paderborn, Germany

{kuznik, marcio, bertrand, wolfgang}@upb.de

Abstract—**Interoperability between verification flows and tools is of particular importance for verification engineers and the EDA industry. While UVM and the Unified Coverage Interoperability Standard (UCIS) target the unified creation and reuse of verification environments, the flow from verification plan to testbench implementation and extraction of coverage data is still a time-consuming and error prone task, for which little automation support is available. In this article we present how to implement a coverage plan-driven functional coverage metric generation for SystemC verification environments, by means of UCIS and state-of-the art code generation and Model-driven Engineering (MDE) techniques.**

I. Introduction

As designs have grown, verification closure by means of coverage metrics is critical to measure the status and quality of the verification plan. The verification plan, usually a spreadsheet style document, defines the verification environment, the stimuli generation plan, the coverage plan and more. Moreover, as interoperability of flows and tools is of particular importance, highly interoperable methodologies for verification and coverage interchange have been introduced, such as the Universal Verification Methodology (UVM) [1], and the Unified Coverage Interoperability Standard (UCIS) [2]. While UVM provides the verification engineer with elements to build reusable testbenches with expected structure, UCIS defines a schema for creation, merge and export of coverage information across simulators and languages, eased by a convenient API.

In order to use those technical advances with the OSCI SystemC reference simulator we developed the System Verification Methodology (SVM) [3] in previous work, which is a verification methodology for the OSCI SystemC reference simulator inspired by UVM. Besides, SVM integrates also a previously developed functional coverage library [4].

Focusing on the verification environment construction and process, we propose the verification plan preparation and its exploitation to leverage from a systematic application of UCIS. Therefore, we

(1) introduce the systematic collection of coverage plan data, while preserving the language independence.
(2) utilize UCIS to store the coverage plan metrics for the DUV in an interoperable format — a UCIS *model*.

By means of a Model-Driven Engineering (MDE) technology the UCIS metrics model is mapped to a design model, combining both models for further analysis and code-generation. Afterward, the corresponding SystemC testbench infrastructure for the DUV is generated using our SVM library. During the simulation run the UCIS database is filled with actual coverage information, gathered from the simulation run.

The remainder of article is structured as follows. Section II will present fundamentals such as UCIS, our previous work and an overview about terms used in MDE. Section III will introduce our contribution in more detail. Section IV shows a detailed step-by-step explanation of the workflow by means of an example system. Related work will be referenced in section V. Section VI will discuss the limitations of our methodology and lessons learned from the application of UCIS, before we summarize and draw conclusions in section VII.

II. Fundamentals

A. Unified Coverage Interoperability Standard

The UCIS [2] has been developed to allow the coverage metric interchange of a variety of coverage producers, from statement coverage and functional coverage to formal. Therefore, an analysis on the information model of verification has been conducted by Accellera and a subset was selected and mapped to a data model, the UCIS data model, being able to represent a range of coverage information models used in practice. A standardized mapping, naming conventions and primary key management make the data objects universally recognizable. Besides, an API was defined that standardizes the way data is written or queried from this data model. The official Accellera UCIS v1.0 standard release provides:

- UCIS Specification v1.0, describing the API functions and the underlying data model.
- UCIS API Header File (.h), to allow custom implementations of coverage producers and consumers.
- UCIS XML schema, as formal definition of the interchange format.

The UCIS specification lists a number of use models. First, coverage producers may use UCIS to generate data. Second,

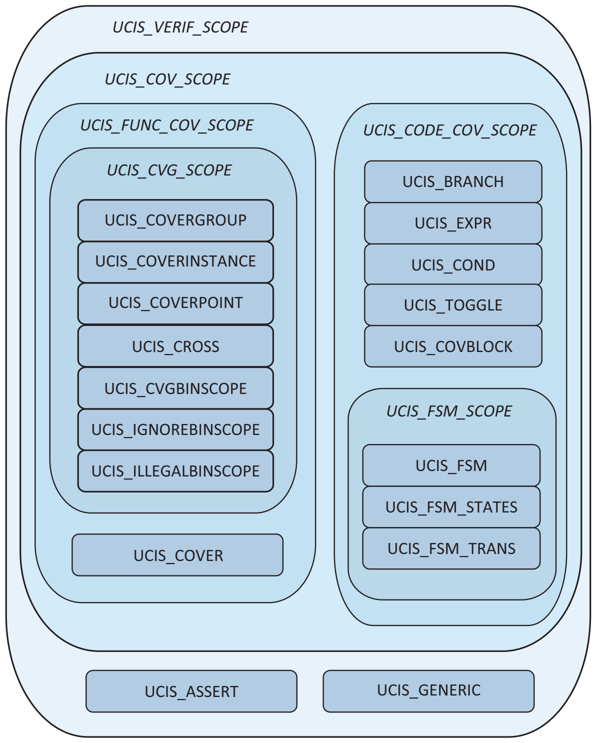


Fig. 1: The UCIS coverage scopes may contain coverage information from various coverage producers [2].

coverage consumers may facilitate the API and the interchange format to perform analysis tasks such as report generation or test plan update. Moreover, the activity to combine coverage results from independent simulation runs (temporal merge or spatial merge) or from different coverage producers (heterogeneous merge) is another specified use case. By means of the UCIS API, standardized interface names for creating, reading and merging coverage metric databases are provided. Besides, custom features can be implemented using API callback functions.

As UCIS may contain coverage information from various coverage data producers such as formal verification, static checks, assertions and data coverpoints it can be used to help in answering both the *Does it work?* and the *Are we done?* questions (verification closure). Figure 1 lists the coverage scopes as defined by the Accellera UCIS standard release v1.0. In section III we will introduce how UCIS can additionally improve the automation on verification closure by means of systematic coverage plan utilization. Moreover, we will focus on the `UCIS_CVG_SCOPE` coverage scope.

B. SVM - A verification methodology for SystemC

Despite SystemC is widely accepted for development at higher abstraction levels such as TLM 2.0, its verification capabilities are rather limited in comparison to other Hardware Design and Verification languages (HDVL) such as IEEE-1800 SystemVerilog or IEEE-1647 *e* as can be seen in Table I. Moreover, in the past verification methodologies were mainly introduced as SystemVerilog implementations, such as the Universal Reuse Methodology (URM) from Cadence, the Advanced Verification Methodology (AVM) from Mentor Graphics, and the Verification Methodology Manual (VMM) from Synopsys.

TABLE I: Selection of verification features of `IEEE-1800`, `IEEE-1647` and `IEEE-1666` compared.

	IEEE-1800 SystemVerilog	IEEE-1647 e	IEEE-1666 SystemC
Functional Coverage	+++	+++	x
Assertions	+++	+++	x
Constraint Solver	+++	+++	+ (SCV lib.)
Verification Meth.	+++	+++	x
TLM	++	+	+++
AOP	x	++	x
C-Software Simulation	simulator dependent	simulator dependent	+++

In order to use recent technical advances within the OSCI SystemC reference simulator we developed the SystemC-based System Verification Methodology (SVM) library [5], [3] and a functional coverage library for SystemC [4], [6] in previous work. SVM is based on the Open Verification Methodology multi-language release (OVM-ML), a donation from Cadence Inc. to the OVM community in February 2009 [7]. We refactored the base package and further improved it to reflect the improvements from the transition of OVM to the Universal Verification Methodology (UVM) standard [1]. We also extended the limited UVM for SystemC subset to offer the same expected structure as in SystemVerilog, e.g. allowing usage of stimuli generation facilities, sequences management and arbitration, command-line processing, etc. with the OSCI SystemC reference simulator. Moreover, a functional coverage library has been integrated into SVM and implements parts of the IEEE 1800-2009 SystemVerilog `covergroup` functional coverage metric to allow coverage-driven-verification (CDV) using the standard OSCI SystemC reference simulator.

Consequently, testbench automation approaches, e.g. code generation, for SystemC can now benefit from these improved verification capabilities. Based on the two introduced contributions we now intend to improve the automation on verification closure within the verification process with SystemC itself.

C. Model-driven Engineering (MDE)

As most part of a verification testbench is software source code containing classes, objects and function calls, we advocate the application of MDE, in order to improve the development of testbenches and the verification process. MDE was proposed to overcome the limitations of object technology, to rise the abstraction and deal with the increasingly more complex and rapidly evolving systems.

The basic concepts supporting the MDE principle are `system`, `model`, `metamodel`, and the relations between them, so that a model represents a system and conforms to a metamodel [8]. Such concepts were organized in 3+1 layers [8] as illustrated by means of the examples `C++` and `XML` in Figure 2.

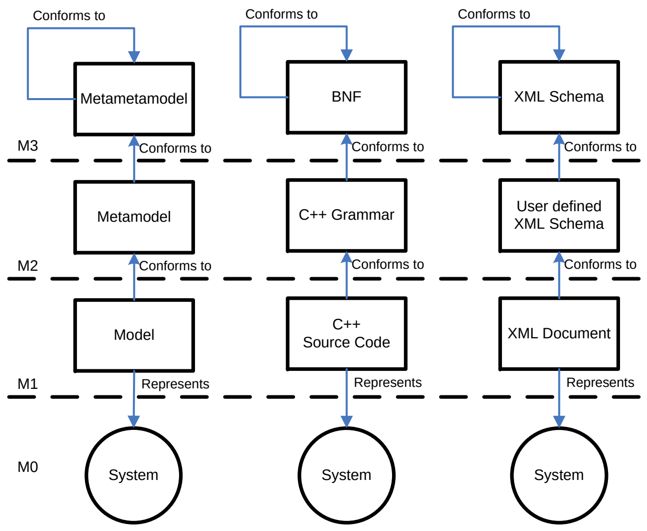


Fig. 2: Basic concepts, layered organization and relation of model and metamodel with `C++` and `XML` as examples.

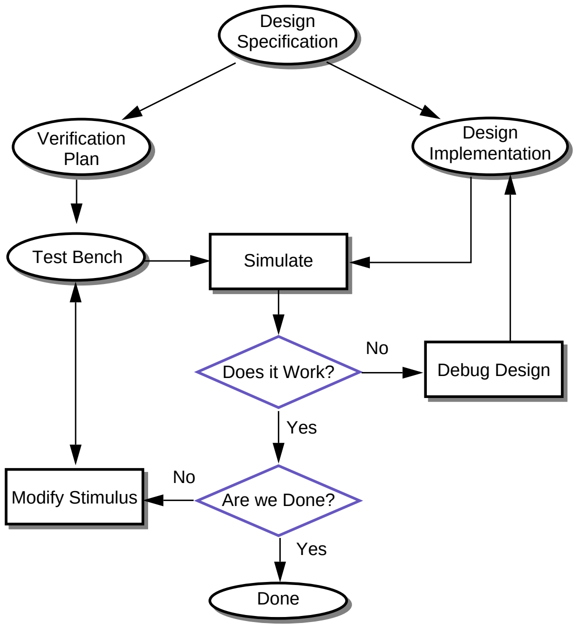


Fig. 3: An abstract view of a typical verification flow [11].

The UCIS metamodel

A metamodel is also a model, which is a reference model for other models, so that it defines classes of models that can be produced conforming to it. It is an abstraction, which collects concepts of a certain domain and the relations between these concepts. As such, the UCIS data model can be used to define the metamodel for the coverage domain. Therefore, we use the UCIS XML schema to generate the UCIS metamodel in our methodology.

Transformations

MDE models are operated through transformations, aiming at the automation of development activity. Such transformations define clear relationships between models [8] and usually are specified in a specialized language to operate on models. Following the description in [9], a model transformation means converting one or more source models to a target model, where all models must conform to some metamodel, including the model transformation itself, which is also a model.

Available tooling to enable MDE

MDE Technological frameworks [10] are tools to support common tasks for MDE independently from the application domain. Such tools rely on standards, in order to generalize the manipulation of models, providing facilities such as persistence, repository management, model transformation, model mapping (weaving), etc. They are the technological support for the MDE principles. For working with the UCIS XML and UCIS metamodel we use the framework provided by Eclipse Modeling Project[1] (EMP) to provide tool support for our methodology.

[1]http://www.eclipse.org/modeling/

III. UCIS FOR SYSTEMATIZED COVERAGE PLAN PROCESSING AND METRIC GENERATION

The verification plan, usually a spreadsheet style document, defines the verification environment, the stimuli generation plan, the coverage plan and more. For example, it defines which features of the design must be tested (*What to verify?*) in the feature plan and defines how the verification shall be conducted (*How to verify?*) in the coverage and checker plan, e.g. by means of a detailed functional coverage metric description. Therefore, the overall verification plan also comprises measurable metrics to determine the progress and completion of the verification process itself.

Unfortunately, the flow from verification plan to testbench implementation and the actual functional coverage metrics is still a time-consuming and error prone task, for which little automation support is available. Figure 3 depicts an abstract view of a typical development flow whereas verification plan definition and design implementation are conducted as parallel threads. Based on the verification plan, test cases and coverage metrics are defined and run together with the DUV in the testbench. If the simulation reveals design flaws the design is debugged to fix bugs. The overall regression testing process is performed until no further faults are detected. and a predefined (functional) coverage criteria is hit.

Especially, the left part of the development flow from design specification to verification and coverage plan definition, as well as the embedding of the latter in a verification environment, is not well standardized and has little automation support. Therefore, in this article we focus on the verification thread from figure 3, more precisely the flow starting from design specification, passing by verification and coverage plan until the integrated metric within the testbench and its re-use. Figure 4 highlights the current situation in most verification

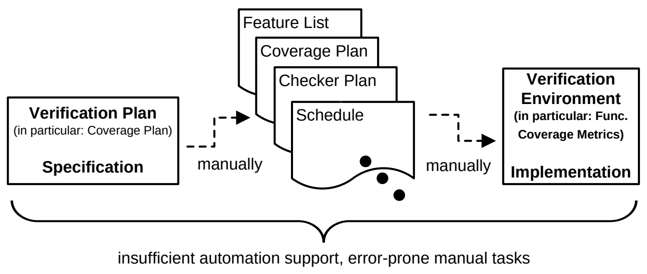


Fig. 4: Low automation support for coverage plan implementation due to informal manual conversions.

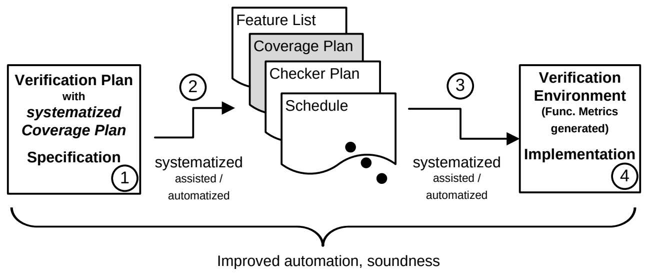


Fig. 5: Methodical Coverage Plan Definition increases automation and soundness.

processes. Here, during the verification plan specification the coverage plan is constructed manually as well as the actual functional coverage metric in the testbench environment.

To improve this process, we propose a more formalized coverage plan definition, as depicted in figure 5, which allows assisted and automatized transformations to be applied on it, hence, increasing productivity and soundness of the actual coverage metric in the testbench environment. In the individual steps we:

(1) enable the systematized coverage plan capture in a model-based fashion.
(2) apply transformations on this data to a cover plan intermediate model, which is in fact UCIS.
(3) bind the UCIS intermediate model to the design model by model mapping.
(4) generate appropriate functional coverage metrics for the testbench environment, based on the mapped model.

The next section will give a detailed explanation of the methodology process and the individual steps by means of a case study.

IV. Applying UCIS-based coverage plan methodology on an example system

In order to improve the verification process we extend the common methodology flow shown in the Figure 3 by adding intermediate UCIS-related steps in the process as can be seen in Figure 6. Supporting tools assist and automatize some of these steps or provide mechanisms to improve the **(i)** interoperability, **(ii)** automation, **(iii)** and reuse in the future. The shadowed areas highlight the verification thread steps in the flow and will be explained in the next subsections.

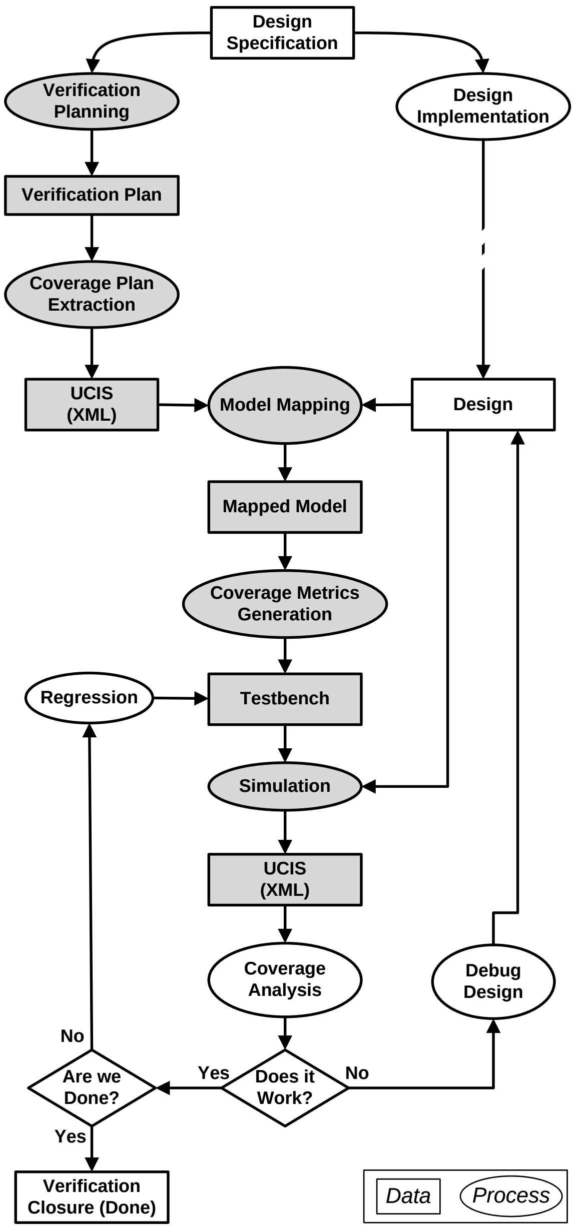


Fig. 6: Systematized coverage plan exploitation flow with UCIS.

Discussion of activities such as design implementation or design debug are not in focus of this article and will be omitted.

A. Example System

In order to demonstrate our methodology, we consider the verification of a functional SystemC model for an Adaptive Cruise Control (ACC) system. Basically, ACC is a vehicle comfort system which enhances the standard Cruise Controller by additionally controlling the distance between the drivers car and the front car, by adapting the vehicle speed. The Figure 7 illustrate the system functionality and Figure 8 shows a simplified architecture of the ACC model.

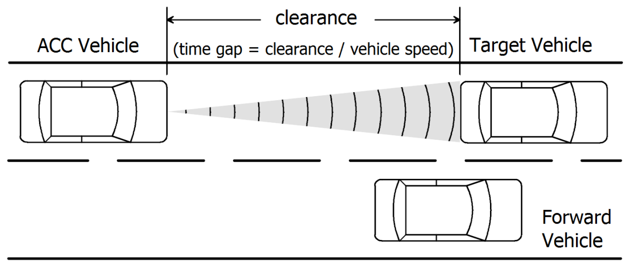


Fig. 7: Functionality of an Adaptive Cruise Control (ACC) system.

The main SystemC modules are:

(1) *SpeedController* module contains the adaptive part of the controller.
(2) *AccelerationController* module represents the standard cruise control component.
(3) *EngineController* is an abstract model of the vehicle.

In fact, *SpeedController* and *AccelerationController* implement the controller functionality of the system and the *EngineController* represents the plant to be controlled. The *SpeedController* is only active when a front vehicle is detected by the radar. Otherwise the desired speed set by the driver is forwarded to the *AccelerationController*. Therefore, as soon as a front vehicle is detected by the radar, the *SpeedController* eventually modifies the current speed depending on the desired distance. The desired distance is also provided by the driver.

The *AccelerationController* executes the control algorithm and delivers the control action to the *EngineController*. The *EngineController* performs the corresponding control action to calculate the current speed of the vehicle and sends it to the *AccelerationController* in return. Besides the introduced modules, the system actually consists of various additional modules, such as a radar model that performs detection of vehicles driving in the same lane. The closest vehicle in the same lane is identified as a target vehicle by the ACC system. The ACC maintains a safety distance to a target by actuating over the throttle or applying the brakes when necessary.

B. UCIS Application Assumptions and Requirements

As UCIS may contain coverage information from various coverage data producers such as formal verification, static checks, assertions and data coverpoints it can not only be used after simulation to store results but also as intermediate format for coverage metrics definition. Our approach and application of UCIS is based on the following assumptions and requirements:

- The Accellera UCIS v1.0 standard release is used.
- We focus on the functional coverage metric capture and generation, hence, on the `UCIS_CVG_SCOPE` scope.
- As simulation infrastructure the OSCI SystemC reference simulator is considered.
- Consequently, we expect a reference implementation of the UCIS API to be available for use cases that go beyond set/get of UCIS XML items, e.g. by EDA tools.
- All implementation activities shall be in conformance to existing EDA standards wherever possible.
- MDE technological framework, such as transformations and metamodels shall be hidden to the end-user wherever possible.

C. Design Implementation

Our proposed methodology assumes there is a design specification, whence a concrete design implementation is derived. The output of the design implementation process is the source code representing the DUV. Although our methodology does not impose restrictions to the design flow, we assume the design is specified in SystemC TLM or RTL. Moreover, a mechanisms to inject the SystemC design in our MDE technological framework is required. Such injection is required in order to analyze the design, mapping it to the UCIS model or generate new code. There are different ways to inject the design model, such as using parsers to read the SystemC source code and generate the model conforming to the respective metamodel. In [12] different front-end tools for this purpose are presented. Alternatively, other representations allow the processing of the design without requiring injection, such as IP-XACT by using tools based on IP-XACT [13] and its interchange format based on XML.

D. Verification Planning

The example system design specification contains a list of use cases whereof the verification plan is constructed in the verification planning phase. To define appropriate stimuli, we make use of the classification tree method (CTM) which provides tree-oriented decomposition of test scenarios into individual variable ranges of interest [14]. The detailed feature plan and verification plan information are omitted due to the focus of our approach on the coverage metric automation. The requirement for language independent metric processing must be reflected in the spreadsheet structure.

To implement our own spreadsheet data model we make use of the Requirement Interchange Format (ReqIF) defined by Object Management Group (OMG) [15]. Using ReqIF the requirements are stored in a data model, which can be handled by an MDE technological framework, simplifying our tool chain. However, support for other types than ReqIF is possible by injecting the requirement in the framework using different tools such as file parsers, e.g. text files defining tables in Comma-separated Values (CSV), or tables in the Microsoft Excel Spreadsheet XML.

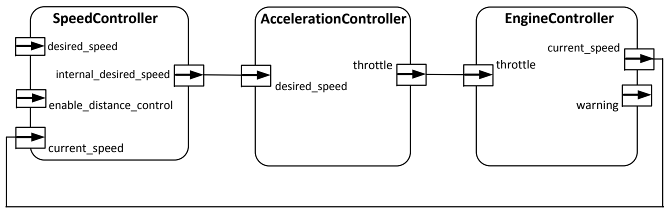


Fig. 8: Simplified architecture of the ACC model.

TABLE II: Excerpt of the coverage plan for `AccelerationController`.

Name	Range	Type	Weight	Goal
desired_speed	[10:100]	BIN	1	100
current_speed	[0:100]	BIN	1	100
desired_distance	[10:30]	BIN	1	100
current_distance	[0:150]	BIN	1	100
enable_acc	[0:1]	BIN	1	100
enable_dist	[0:1]	BIN	1	100

A filled table can be seen in table II. The table shows an excerpt of the coverage plan for our example system. The names in the table identify some important system input, namely *desired_speed* and *desired_distance* which specify respectively the speed and distance configured by the user. Other important values are the *current_speed* and *current_distance* calculated by the ACC system. The other two boolean values *enable_acc* and *enable_dist* allow one to turn on and off the adaptive control of the speed and the distance control respectively.

Note that information such as the actual variable specified in the design that must be connected to the coverage bin and the trigger condition do not need to be specified at this point. As such information is defined in the design and may not be synchronized with the verification plan, we propose to add such information during the mapping between the extracted coverage plan (as UCIS model) to the design model, as described in the next sections.

This step is equivalent to the interpretation performed by the engineer, when implementing the coverage metrics in the test bench, however we are providing tool-support. Moreover, the mapping step creates a *logical link* between the design and coverage plan that can be traced by computational tools, in order to update references or as in our proposal, to generate code.

E. Coverage Plan Extraction

The systematic specification of the coverage plan in the previous step, allows automatic data processing and extraction of valuable information. At this step, the relevant information for the coverage is automatically extracted from the verification plan and stored in a UCIS model. This model is filled with information about the specified metrics for coverage, such as coverage group, coverage description, and others elements. Figure 9, left pane illustrates part of the UCIS model extracted from the verification plan for our ACC case study.

The UCIS format is used here to store coverage information orthogonally to the design. As such, one mapping step is required to link the UCIS model to design entities before the testbench generation, as our UCIS model does not contain any design information.

F. Model Mapping

The verification plan does not contain information about the design, only references to design entities based on the design specification. Therefore, in the common verification flow presented in the Figure 3 an engineer interprets the verification plan and manually implements the test bench, by including the source code responsible for coverage data extraction during simulation and handle UCIS models. Such task consists of understanding the coverage intent and the design where it must be applied before implementing the testbench code.

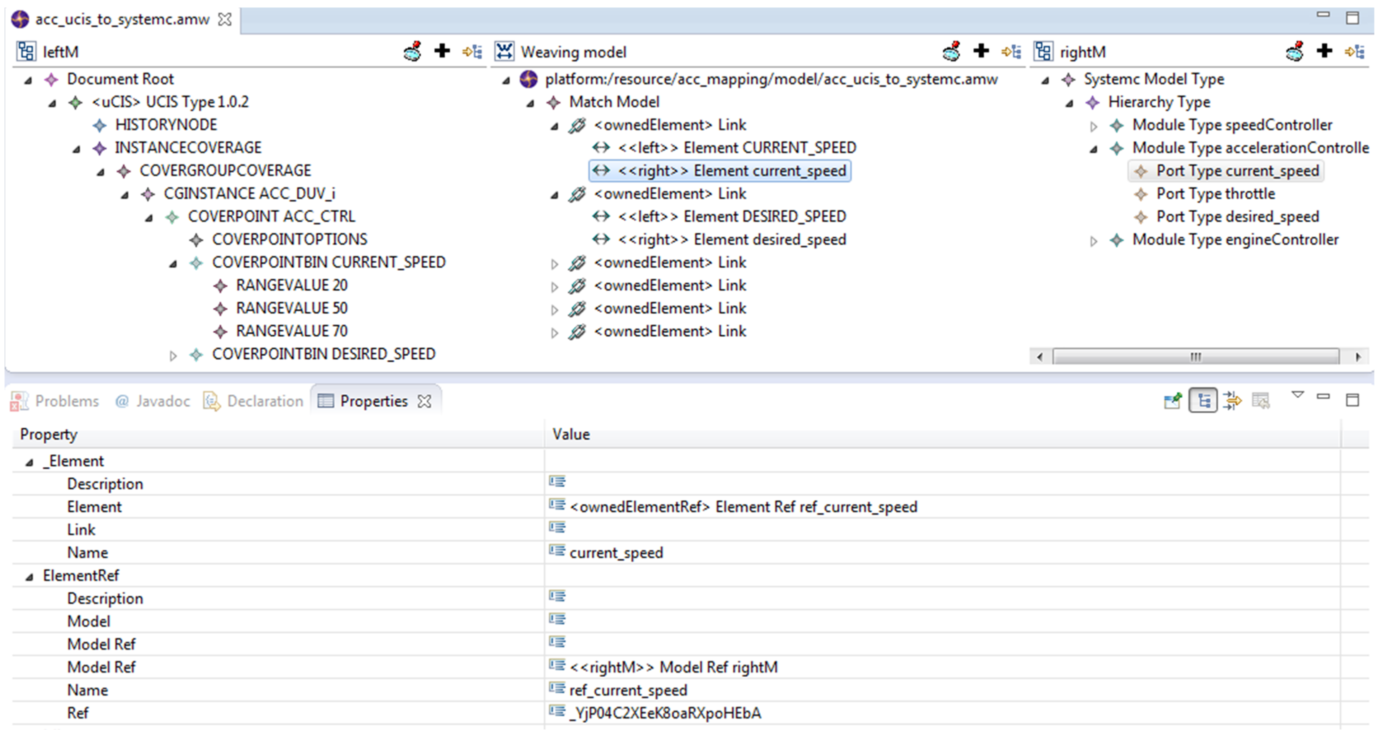
In our proposed methodology, the model mapping process receives a UCIS model and a design model as input. This process generates another model as output containing references to both input models. The information that could not be gathered from the verification plan can now be included in the UCIS model with tool support. Improving the UCIS model and linking it to the design model in the next step allows the automatic generation of the testbench source code that is responsible for the extraction of coverage data during the simulation and for the handling of UCIS models.

The model mapping tool support is provided by AWM [16]. This tools consist of a multiple pane editor, where the UCIS, the mapping model and the design model are displayed side-a-side. Dragging and dropping elements from UCIS and design model into the mapping model, an engineer build links between the models and at the same time is able to include additional information in the UCIS model, if it is required.

As the UCIS metamodel does not contain specialized constructs to reference external models and design elements, the mapping model is required to associate elements from UCIS model to the design ones. In order to keep the UCIS specialized on the coverage domain, the addition of constructs to link multiple models in UCIS metamodel is not desired. Moreover, by using the mapping approach the UCIS stays independent of design language, so that the UCIS model can be associated to SystemC, SystemVerilog or any other language.

G. Coverage Metrics Generation

This process enhances the implementation of coverage metrics, by generating automatically the source code used to extract the coverage metrics during the simulation. Considering the Figure 9, the generation process starts following the links defined in the mapping model (Figure 9 middle pane) to elements of the UCIS model (Figure 9 left pane), and generates the source code constructs related to it, such as acquiring the factory reference, creating the coverage database, and cover groups. The Listing 10 lines 2-8 show part of the code generated for such constructs. Moreover, the process generates code that refers to design elements, by following associated links between UCIS and design model (Figure 9 right pane) elements. For instance, code to create different type of bins as illustrated in the Listing 10 lines 18 and 22. A default report can also be generated inside of the `SVM Component`'s report callback. The generated code relies on the SVM library, which provides the API to implement SystemC testbenches in a UVM like fashion as well as functional coverage.


acc_ucis_to_systemc.amw

leftM
- Document Root
  - <uCIS> UCIS Type 1.0.2
    - HISTORYNODE
    - INSTANCECOVERAGE
      - COVERGROUPCOVERAGE
        - CGINSTANCE ACC_DUV_i
          - COVERPOINT ACC_CTRL
            - COVERPOINTOPTIONS
            - COVERPOINTBIN CURRENT_SPEED
              - RANGEVALUE 20
              - RANGEVALUE 50
              - RANGEVALUE 70
            - COVERPOINTBIN DESIRED_SPEED

Weaving model
- platform:/resource/acc_mapping/model/acc_ucis_to_systemc.amw
  - Match Model
    - <ownedElement> Link
      - <<left>> Element CURRENT_SPEED
      - <<right>> Element current_speed
    - <ownedElement> Link
      - <<left>> Element DESIRED_SPEED
      - <<right>> Element desired_speed
    - <ownedElement> Link
    - <ownedElement> Link
    - <ownedElement> Link
    - <ownedElement> Link

rightM
- Systemc Model Type
  - Hierarchy Type
    - Module Type speedController
    - Module Type accelerationControlle
      - Port Type current_speed
      - Port Type throttle
      - Port Type desired_speed
    - Module Type engineController

Problems | Javadoc | Declaration | Properties

Property	Value
_Element	
Description	
Element	<ownedElementRef> Element Ref ref_current_speed
Link	
Name	current_speed
ElementRef	
Description	
Model	
Model Ref	
Model Ref	<<rightM>> Model Ref rightM
Name	ref_current_speed
Ref	_YjP04C2XEeK8oaRXpoHEbA

Fig. 9: Multi pane editor, with an excerpt of ACC models: Coverage-UCIS (left), UCIS to design mapping (middle), and SystemC design model (right).

H. Simulation

To define appropriate stimuli, we made use of the classification tree method (CTM) during verification planning which provides tree-oriented decomposition of test scenarios into individual variable ranges of interest [14]. Moreover, for our specific case study this approach avoids the generation of meaningless stimuli. We combine this methodical breakdown of test scenarios with common constrained-random stimulation. Here, we build up-on the SVM library and apply the `MiniSat` based constraint solver CRAVE for SystemC [17].

During simulation of the ACC System the generated coverage metrics collected functional coverage information. Figure 11 shows an excerpt of the functional metric after simulation. In this report the hits represent the accumulated hits of the specified individual intervals.

I. UCIS Reader/Writer API Generation

At the beginning of the simulation the data contained in the UCIS model must be loaded. This model is updated during the simulation or at the end, depending on the coverage metric extraction strategy and if the results must be integrated in an existing UCIS model. To have the coverage results in an interoperable format its necessary to store the metric according to the UCIS schema definition. Therefore, we apply MDE to generate the API implementation from the XML schema for the Accellera UCIS v1.0 release. The tooling transforms the XML schema in an ECORE metamodel. Afterward, it generates a UCIS API for C++, by using EMF4CPP, and for Java, by using EMF. Such APIs are used during the simulation

```
// Init the factory
svm_pFac = svm_Factory::init();

// Acquire existing UCIS model
svm_pFac->setCoverageModel(acc_ucisModel);

// Specify metric, covergroups
cv_pCG = svm_pFac->newCovergroup(this, "ACC_DUV",
      "ACC_DUV_i");

cv_pCP = svm_pFac->newCoverpoint(cv_pCG, "ACC_CTRL");
cv_pCP->set_weight(1); // type_options
cv_pCP->set_at_least(100);
cv_pCP->set_goal(90);

(...)

// Specify metric, bin types
cv_pBa = svm_pFac->newBins(cov_pCP, "CURRENT_SPEED",
      AUTOBINS);
cv_pBa << range(10, 49) << range(50, 69) << range(70,
      100);
cv_pBa->connect(current_speed);

cv_pBb = svm_pFac->newBins(cov_pCP, "DESIRED_SPEED",
      AUTOBINS);
cv_pBb << range(10, 49) << range(50, 69) << range(70,
      100);
cv_pBb->connect(desired_speed);

(...)
```

Fig. 10: Excerpt of functional coverage metric for `AccelerationController`.

to manipulate the UCIS model but could be used as back-end in custom EDA tools, in order to improve the coverage related flows. The generated API implementation contains functions

```
BIN: ACC_SPEED_CTRL:desired_speed::: 20145 Hits
BIN: ACC_SPEED_CTRL:current_speed::: 21893 Hits
BIN: ACC_SPEED_CTRL:desired_distance::: 20772 Hits
BIN: ACC_SPEED_CTRL:current_distance:: 23383 Hits
BIN: ACC_SPEED_CTRL:enable_ac::: 414 Hits
...
```

Fig. 11: Excerpt of `ACC_CTRL` transaction coverage report.

to load/store entries from UCIS models, which corresponds to read/write on the UCIS XML, respectively. It also contains a factory to create elements of the UCIS metamodel. Functions such as set/get and add/remove are generated, as well as interfaces and implementation for all concepts in the metamodel. The generated API and EMF library also provide features to observe state changes in the model, control data transactions, interfaces and callbacks to customize the API. Figure 12 illustrates the generated API for C++.

The sample code shown in the Figure 12 starts registering the UCIS metamodel. This step allows tools to be aware of any change in the metamodel. In the Line 4 an `ecorecpp::parser` is defined. The parser knows the UCIS metamodel, because it was registered before, hence it is able to load the UCIS model in the Line 9.

In case Accellera changes the UCIS metamodel (or the XML Schema), a new reader/writer can actually be automatically generated from the metamodel and no changes in the read/write functions are required. Lines 11 and 12 get an instance of the `UCISPackage` and `UCISFactory`. This classes provide facilities to create and access instances of classes. Line 16 illustrates how an object is created using the factory concept and the Line 16 shows how the API is used to change the model. After all changes in the model are done, one can store the model into a XML database. Line 21 shows the call to the serializer, which writes the model in XML using the UCIS XML Schema. Finally, the API instance can be deleted in the Line 24.

```
1  (...)
2  // Registering the metamodel
3  ecorecpp::MetaModelRepository::_instance()
        ->load(UCISPackage);
4
5
6  // Loading the model from XML
7  ecorecpp::parser::parser parser;
8  DocumentRoot_ptr ucisModel =
        parser.load("acc_cov.xml")->as< DocumentRoot >();
9
10 (...)
11 UCISPackage_ptr ucisPackage =
        UCISPackage::_instance();

12 UCISFactory_ptr ucisFactory =
        UCISFactory::_instance();

13
14 // Create Instance Coverage
15 INSTANCECOVERAGE_ptr icoverage =
        ucisFactory->createINSTANCECOVERAGE();
16  icoverage->setName("InstCov"); /*@
17 (...)
18 // Serialize the model
19 ecorecpp::serializer::serializer
        ser("acc_sim_cov.xml");
20 ser.serialize(ucisModel);
21
22
23 // Delete the model
24 delete ucisModel;
25 (...)
```

Fig. 12: Sample of UCIS API generated from the UCIS XML schema.

V. Related Work

Data models for storing and merging coverage information from multiple coverage producers are common practice in EDA design flows and multiple vendors offer verification management solutions covering process management and test plan tracking. Nonetheless, prior to Accellera UCIS no interoperable data model was existing. As initial stimulus for UCIS development Mentor Graphics donated the UCDB technology. [18]. To the best of our knowledge so far there is no application of UCIS in early stages of design, such as systematized coverage plan processing. This may be related to the specific set of use cases currently specified for UCIS application, as well as a set of limitation of the current Accellera UCIS v1.0 release that will be discussed in section VI.

VI. Limitations and Lessons Learned

This section will list limitation of our prototype implementation as well as lessons learned with the usage of UCIS. Moreover, we intend to highlight challenges and potential pitfalls on building custom interconnections to the UCIS data model.

Implementation limitations

The current flow has several limitations. First, we only consider the `UCIS_CVG_SCOPE` coverage scope of UCIS for improved automation on verification closure due to the fact that we only had a functional coverage library available. In general, an extension of formalized capture, mapping and automation are also possible for other coverage scopes if suitable coverage producers are available. Moreover, we only focus on code-generation for the SystemC OSCI simulator. Besides, we expect the existence of a model of the SystemC design that is compatible with MDE tooling.

One-time effort for tool platform construction

The one-time effort to establish a working link from meta-model based coverage plan entries to generated coverage metric statements in the verification environment is moderate but requires expertise in the fields of model-driven engineering. In particular, the definition of an appropriate meta model for formalized coverage plan capture, e.g. by means of the OMG Requirements Interchange Format (ReqIF), and associated Eclipse tooling are likely to be out of scope for verification engineers. However, these models and transformation only need to be defined once. Afterward the usage of APIs is straightforward and even the building of code generators once the data model has been structured.

UCIS data model

From our point of view it is a promising direction that there is not only a common standardized data model for exchange and merge of coverage producers data, but also a more standardized way of processing the coverage plan from higher levels of abstraction. Here, the current UCIS v1.0 release is missing specific structures in the data model to allow fast adaption for testplan description such as generic trigger conditions for metrices, a mandatory information for code-generation for RTL design.

The experience from MDE community may contributes for those issues. A domain-specific language can be specified to address the condition expressions at different abstraction levels (RTL/TLM), by using existing languages from EDA community, such as SystemVerilog or defining a new expression language tailored to this problem.

UCIS schema and API

Despite the fact that the Accellera UCIS v1.0 standard release provides a header file describing the standardized API functions, a reference reader (or writer) implementation is not included. Moreover, the standardized API functions do not always match with the structure of the XML schema elements. Such a mismatch can be bothersome if no synchronization mechanism is provided once the UCIS model and API evolve.

Metamodeling tools such as ECORE, EMF4CPP and MDE standards can provide strong contributions to UCIS users such as automatic code generation based on the UCIS metamodel. Such a generation can act as synchronization mechanism between API and data model. Moreover, additional features such as model copy, merge, version control and others are available, once a metamodel is available.

VII. CONCLUSION

In this article we introduced an approach to systematize coverage metric generation based on methodical coverage plan data capture and machining, utilizing the Unified Coverage Interoperability Standard (UCIS). We formalized the flow from design specification to verification plan and coverage plan specification by means UCIS as intermediate format. Defining a verification methodology incorporating these UCIS related steps in the earlier phases of verification planning we can assist and automize functional coverage metric generation. Despite the fact that we concentrated on the `UCIS_CVG_SCOPE` functional coverage scope of UCIS, the approach in general is also applicable to other coverage scopes of UCIS - if suitable coverage producers are available. Moreover, although knowledge of model-driven engineering techniques and specific software centric tooling was necessary to define parts of the verification process steps, the actual end-user, here a verification engineer, does not necessarily need to be aware of the back-end flows once an initial one-time effort setup was conducted. Therefore, we see additional potential for the usage of standardized coverage models such as UCIS in earlier phases of the system design, in particular, the verification plan creation and exploitation phase, besides being the future standard for coverage data exchange between different vendor tool chains and flows.

ACKNOWLEDGMENTS

This work was partly funded by the German Ministry of Education and Research (BMBF) through the project SANITAS (16M3088I) [19], the DFG SFB 614, the ITEA2 projects VERDE (01S09012) and TIMMO-2-USE (01IS10034). We greatly appreciate the cooperation with the project partners.

REFERENCES

[1] Accellera Organization, Inc. (2012, May) Universal Verification Methodology (UVM). Accellera Organization, Inc. [Online]. Available: http://www.accellera.org/downloads/standards/uvm

[2] Accellera Organization, Inc. (2012, June) Unified Coverage Interoperability Standard (UCIS). Accellera Organization, Inc. [Online]. Available: http://www.accellera.org/downloads/standards/ucis

[3] M. F. S. Oliveira, C. Kuznik, W. Mueller, W. Ecker, and V. Esen, "A SystemC Library for Advanced TLM Verification," in *Proceeding of Design and Verification Conference (DVCON)*, 2012.

[4] C. Kuznik and W. Müller, "Functional Coverage-driven Verification with SystemC on Multiple Level of Abstraction," *Proceeding of Design and Verification Conference (DVCON)*, 2011.

[5] M. F. Oliveira, C. Kuznik, H. M. Le, D. Groe, F. Haedicke, W. Mueller, R. Drechsler, W. Ecker, and V. Esen, "The System Verification Methodology for Advanced TLM Verification," ser. CODES+ISSS '12.

[6] C. Kuznik and W. Müller, "Verification Closure of SystemC Designs with Functional Coverage," *16th North American SystemC User Group Meeting*, 2011.

[7] Cadence Design Systems, Inc. Open Verification Methodology Multi-Language (OVM-ML). [Online]. Available: http://www.ovmworld.org/

[8] J. Bezivin, "On the unification power of models," *Software and Systems Modelling*, vol. 4, no. 2, pp. 171–188, May 2005.

[9] D. Gasevic, D. Djuric, and V. Devedzic, *Model Driven Engineering and Ontology Development*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-00282-3

[10] R. France and B. Rumpe, "Model-driven Development of Complex Software: A Research Roadmap," in *2007 Future of Software Engineering*, ser. FOSE '07. Washington, DC, USA: IEEE Computer Society, May 2007, pp. 37–54.

[11] *Questa SIM Users Manual, v10.1a, Chapter 25 Coverage and Verification Management in the UCDB*. [Online]. Available: http://www.mentor.com/products/fv/questa-verification-platform

[12] K. Marquet, B. Karkare, and M. Moy, "A Theoretical and Experimental Review of SystemC Front-ends," in *FDL*, A. Morawiec, J. Hinderscheit, A. Morawiec, and J. Hinderscheit, Eds. ECSI, Electronic Chips & Systems design Initiative, 2010, pp. 124–129.

[13] "IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tools Flows," *IEEE Std 1685-2009*, pp. C1 –360, 18 2010.

[14] W. Müller, A. Bol, A. Krupp, and O. Lundkvist, "Generation of executable testbenches from natural language requirement specifications for embedded real-time systems," in *DIPES/BICC*, 2010, pp. 78–89.

[15] Object Management Group. (2011, April) Requirements Interchange Format (ReqIF) v1.0.1. [Online]. Available: http://www.omg.org/spec/ReqIF/

[16] M. D. Del Fabro, J. Bézivin, F. Jouault, E. Breton, and G. Gueltas, "AMW: A Generic Model Weaver," *Proc. of the 1ères Journées sur l'Ingénierie Dirigée par les Modèles*, 2005.

[17] F. Haedicke, H. M. Le, D. Groe, and R. Drechsler, "CRAVE: An Advanced Constrained RAndom Verification Environment for SystemC," in *Proceedings of the International Symposium on System-on-Chip 2012. October 11-12, Tampere, Finland.* IEEE, 2012. [Online]. Available: http://www.systemc-verification.org/

[18] W. Gude. Questa Verification Management TechTalk. Mentor Graphics Inc. [Online]. Available: http://www.mentor.com/player/2008/vm_techtalk/index.htm

[19] Collaborative verification along the entire value-added chain; "SANITAS" research project launched under management of Infineon. [Online]. Available: http://www.infineon.com/cms/en/corporate/press/news/releases/2009/INFXX200912-018.html