A New Test-Generation Methodology for System-Level Verification of Production Processes *

Allon Adir, Alex Goryachev, Lev Greenberg, Tamer Salman, and Gil Shurek

IBM Research - Haifa; Haifa, Israel
{adir,gory,levg,tamers,shurek}@il.ibm.com

Abstract. The continuing growth in the complexity of production processes is driven mainly by the integration of smart and cheap devices, such as sensors and custom hardware or software components. This naturally leads to higher complexity in fault detection and management, and, therefore to a higher demand for sophisticated quality control tools. A production process is commonly modeled prior to its physical construction to enable early testing. Many simulation platforms were developed to assess the widely varying aspects of the production process, including physical behavior, hardware-software functionality, and performance. However, the efficacy of simulation for the verification of modeled processes is still largely limited by manual operation and observation. We propose a massive random-biased, ontology-based, test-generation methodology for system-level verification of production processes. The methodology has been successfully applied for simulation-based processor hardware verification and proved to be a cost-effective solution. We show that it can be similarly beneficial in the verification of production processes and control.

Keywords: Production processes / Manufacturing processes, Test generation, Transaction-based modeling, UML/SysML.

1 Introduction

Modern production processes are becoming smarter and more complex. Cheaper, smarter sensors allow for more informative control of production systems [1]. The traditional end-of-line quality control can filter out defective products for recycling. On the other hand, in-process quality control provides the ability to take action on defective workpieces within the production process [2]. In-process control is largely based on the incorporation of various sensors in the various stages of the manufacturing chain. This leads to more complex possible production "paths" that include the possibility of fixing or adjusting a defective workpiece, repeated application of one or more stages, possible in-process product sorting based on observed features, and more.

* The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 285075.

The growth in complexity poses a challenge to the design and construction of robust and error-free production processes. Smart and automated verification activities are required to ensure the correctness of process designs. In general, behavioral models are useful for pre-construction simulation and testing as well as for post-construction adaptation. Finding design errors in simulation environments before creating the actual system is easier and cheaper than looking for them in the system itself, due to better observability and controllability of the system. In addition, the actual construction of a production system can be very costly, making it highly desirable to correct as many faults as possible in a design before it is implemented. Some existing modeling environments of production processes [3–11] were devised to enable visualization [4, 7], while others produce executable behavioral models for simulation purposes [3, 6, 8, 10, 11]. However, testing in these environments focuses on structural coverage and time-related verification.

Nevertheless, such simulation platforms can help assess the system behavior and performance in different "situations" for functional verification purposes. These situations (i.e., tests) are the key to effective deployment of the simulator. The common approach of manually operating the simulator suffers both from the relatively small number of tests that are eventually used and from the difficulty of directing the system to situations that may be worthy of testing. For example, creation of scenarios that involve events occurring far from the system inputs may require complex analysis of the system operation. Automatic test generators exist for simulation, but these focus mainly on simulating the expected typical behavior of the system and lack controls to direct the testing toward the rare functional corner cases that may be required in the test plan.

In this paper, we focus on automatic and directed massive test generation for simulation environments that allow for functional and performance testing. A good test generator produces tests that exercise the possible paths of the system under various performance- and stress-related conditions. In addition, using verification testing-knowledge can enhance the coverage achieved by the generated tests and bring the simulated system to corner cases or stress scenarios according to a given test plan.

Our approach uses two separate models of the system. The first model is an executable behavioral model that is used for simulation. The second model, on which our test-generation expert system is based, serves as a repository for accumulating knowledge on how to generate tests. This knowledge includes information about the system structure and properties, which is required for generating valid tests and also knowledge as to how to best test the production system. This is typically a derivation and extension of the knowledge accumulated for past verification efforts [12].

Our proposed methodology of test generation for simulation environments of production processes is drawn from the field of processor hardware verification. The great advancements in processor hardware verification technologies were made possible due to the high cost of bugs in hardware, which brought large investments to the field. It is also due to the increasing use of formal languages in

the hardware design process, which enabled increased automation in verification. Specifically, our test generation expert system uses the X-Gen test generator [13], originally developed and successfully used for testing IBM's processor hardware systems.

Our work is being conducted as part of the MuProD project [2] funded by the European Commission within the Seventh Framework Programme. MuProD is advancing solutions that avoid end-of-line (EOL) failures in production processes via intelligent and integrated in-process measurement techniques. These measurement techniques are immediately followed by reactions to generated defects, as they are detected. This novel type of production system can benefit much from our proposed test-generation methodology because of the large number of possible different paths and behaviors induced by the multiplicity of sensors, reworking stages, and in-process fault types.

Our proposed verification methodology also uses simulation monitors to track test coverage and identify possible problematic system behavior during the simulation of a test. The experiment reported in Sec. 6 used simulation-monitoring techniques to monitor the coverage of system simulations. These monitoring techniques were developed for the DANSE project [14], also funded by the European Commission within the Seventh Framework Programme. DANSE deals with the development of new methodology to support the evolving, adaptive, and iterative system-of-systems (SoS) life cycle models, including analysis and simulation support.

Our approach offers the following contributions to the verification methodologies of production processes: 1) Defining a testing ontology for production processes using (UML) [15] or (SysML) [16], following the concept described in [17]; 2) Performing functional and performance verification by massive random-biased testing; 3) Demonstrating how transaction-based testing can be applied to production processes; and 4) Enabling capture and accumulation of testing knowledge for functional and performance verification of production processes.

2 Methodology of Simulation-Based System-Level Verification of Production Processes

This section begins with a description of a working example of a production process design. We then describe our proposed methodology for system-level verification and show how it can be applied to the working example.

2.1 Working Example

The working example that accompanies this paper is a production process composed of 35 components. Most of these are generic components, including operators, buffers, sensors, tracks and track junctions (track joiners and splitters), a scrap heap, and product stocks. Some components are processing stages specific to our example production process (e.g., one shifter and one injector). The objects passing through the production process are packets. A packet contains

three attributes: an ID for unique identification purposes of the packet and two attributes with integer values named X and Y (which may represent some dynamic physical attributes of the manipulated packet). The topology of the production process is depicted in Fig. 1, where the solid lines portray possible paths for the flow of packets and the dashed lines portray the flow of sensory information.

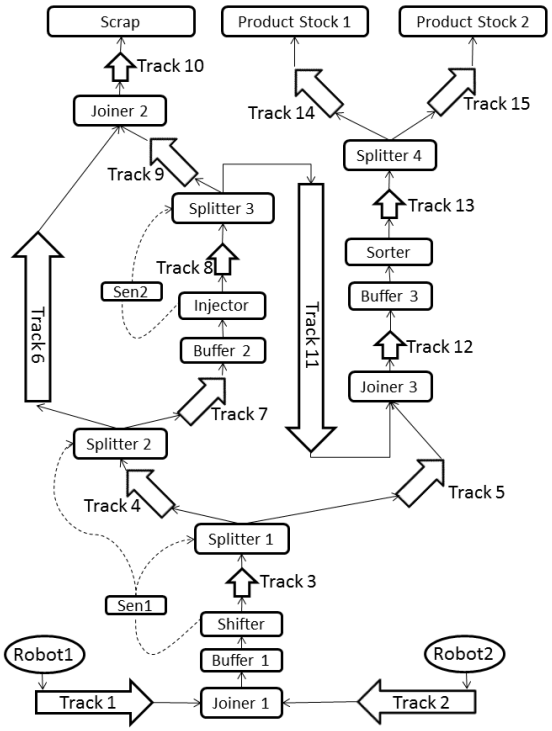

Fig. 1: A working example of a production process topology.

In this working example, two operators, called Robot 1 and Robot 2, put packets on tracks toward processing by a shifter stage, which shifts some constant amount from the Y attribute of a packet to the X attribute. However, during shifting some amount could be lost (i.e., leakage). A sensor identifies the leakage during shifting and sends the information to corresponding splitters designed so that three different continuations are possible for the packet. A path (through Track 5) for successfully shifted packets for which the leakage is tolerated (according to some tolerance threshold parameter), a second path (through Tracks 4 then 6) for packets that are ruined by excessive leakage, and a third path (through Tracks 4 then 7) for packets for which the leakage is not tolerated, but may be corrected by a re-work. The re-work for the latter path is performed by another processing stage called the injector. The injector adds some constant amount to the X attribute of the packet. However, the injected

amount might also suffer from leakage. Another sensor computes whether the leakage is small enough and whether or not it was able to bring the total leakage to a tolerated amount or not. If the re-work was successful, the packet is joined to the path leading to the product stocks (through Track 11). Otherwise, the packet is joined to the path leading to the scrap heap (through Track 9). Packets designated for the product stocks are sorted by another splitter according to the relation between their X and Y attributes. If X is larger than Y they arrive at the first product stock, otherwise they arrive at the second product stock.

2.2 Simulation-Based System-Level Verification Platform

The methodology under which our test generation solution operates is borrowed from the field of processor hardware verification (specifically verification of computer systems and SoCs) [13] and is described in Fig. 2. In this methodology, the verification engineer writes test templates according to a test plan. These test templates serve as the main input to the test generator. The test templates aim to generate tests that will drive the simulation towards desired states or scenarios in the design. The test generator is an expert system that includes both system-oblivious parts and system-specific parts. The test generator is based on a domain-specific ontology, in which an application engineer defines the terminology for specifying production systems and testing knowledge. Using this terminology, the application engineer then models a specific production process and its testing knowledge. The result of the test generation is a set of random-biased tests. These tests are fed into a simulator that simulates the design. The design's behavior while the test is running can be verified by various means. For example, verification could be done by using a reference model, or alternatively, by embedding simulation *checking monitors* to detect the occurrence of violations of system requirements. Such checking monitors give appropriate failure and pass reports. Another set of monitors, known as *coverage monitors*, can be used to detect the occurrence of the events targeted by the test plan. The coverage monitors produce coverage reports that are analyzed by the verification engineer. The verification engineer can be either satisfied with the testing results according to the test plan or can update the test templates to produce additional tests.

3 Modeling for Simulation and the Simulation Platform

Three types of components can be distinguished in the simulation platform that executes the tests (Fig. 3): designed components, behavioral components, and physical components.

The designed components are executable models that are direct representations of a designed element of the production system. These designs will later serve as the basis for implementation and, hence, finding problems in them is one of the main goals of the simulation-based testing.

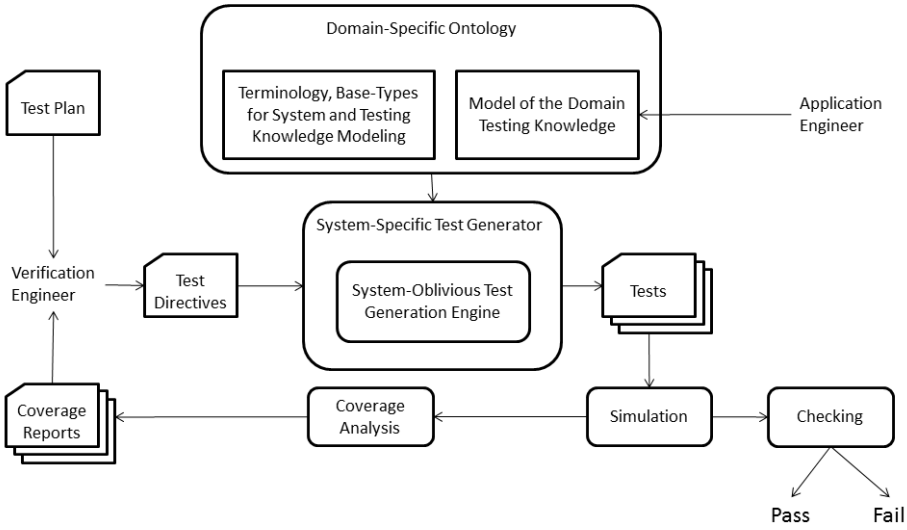

Fig. 2: High level description of the testing methodology for simulation based design verification.

The behavioral components are behavioral representations of some element of the production system. These components behave like the real system element, but are designed differently. Typically, they involve a much simpler software implementation that has the same behavior as the real element. These behavioral models are needed for full-system simulation but are not the object of the testing procedure. A bug found in a behavioral model merely reflects a problem in the simulation and not in the system to be implemented. The behavioral models can represent components that have not yet been designed; components that cannot be designed, such as users or human operators; or alternatively, components that have already been thoroughly tested. A simulation environment can start with a few designed components and many behavioral components (Fig. 3(a)) that get replaced with their respective designed substitutes as the system design matures (Fig. 3(b)).

The physical components can be integrated in the simulation using classic system-integration-lab [18], or hardware-in-the-loop (HIL) platforms. Some of the designed components can be progressively replaced by the corresponding implemented physical components (Fig. 3(c)).

A production process can be modeled using any existing modeling platform that supports model simulation [3, 6, 8, 10]. For this work we chose SysML for modeling production processes. SysML is a general-purpose modeling language for systems engineering applications. It is an extension of a subset of UML, which originally targeted software or software-intensive systems. These modeling languages are object-oriented in nature and can be translated into executable languages using code generators. The executable models can then be run for simulation purposes.

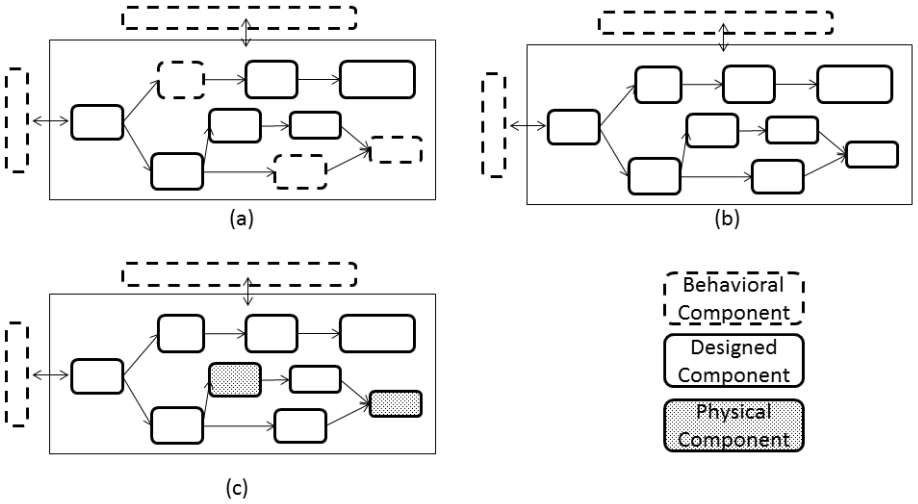

Fig. 3: High level description of the evolution of a simulation environment, where behavioral components are replaced with corresponding designed components and then by physical components as the production process model matures. (a) A production process model described using designed and behavioral components. (b) A production process described using designed components. (c) A production process described using designed components and integrating physical components. The remaining behavioral components after the model matures are input/output components and human operators.

We model components in a production process using SysML blocks. We model the behavior of components using statecharts and the interaction among component types using flowports and events. The topology of the production process is modeled in a SysML internal block diagram.

In addition to being able to transform the model into an executable program, the model should also be able to consume our generated tests. This means the behavioral components should be able to read the instructions given in a test and operate accordingly.

The software components of the simulated system are instantiated from a production-process ontology, which is basically an extendable and reusable library of component types. The component types are chosen either from an existing set of component types—such as tracks, track-junctions (splitters and joiners), buffers—or especially modeled for the specific production system such as models of in-line machining and sensors. The configuration of the model consists of configured instances of the component types and the topology of their connections. For example, a track can be configured to have a certain length and speed. In our working example, all tracks are configured to have a length with room for ten slots of packets, and they advance one slot every half a second.

4 Test Generation

We used X-Gen [13] to generate tests for production processes, such as the working example process shown in Fig. 1. X-Gen is a test-case generator originally developed and applied to computer systems and system on chip (SoCs). X-Gen adheres to the ontology-based paradigm [12]. The generator is partitioned into two separate layers. At the core of X-Gen is a system-oblivious test generation engine, which is capable of generating tests for a variety of systems. This generic engine, with the addition of the domain-specific ontology, is a system-specific test generator as shown in Fig. 2.

The abstract system model, which is a part of the domain-specific ontology, describes the system's components and its *interactions* (i.e., system scenarios that involve multiple components that collaborate to achieve some target). Testing knowledge is the verification expertise that biases the random test generation toward interesting areas for verification. Incorporating testing knowledge does not require explicit input from the user. An example of the testing-knowledge building blocks provided by X-Gen is its collision mechanism that biases tests toward the (possibly concurrent) reuse of certain system resources.

The topology of the system under test is described in a configuration file, while the verification scenario is defined in a test template file. The scenario definition is done using a proprietary special-purpose programming language developed for this task. The language allows verification engineers to define scenarios ranging from fully deterministic to totally random.

Once the test template is consumed by the test generator, a set of tests is created. These tests can be fed into the simulation to practice the different scenarios they represent in the model of the production process.

4.1 Test Template Definition Language

A test template describes the test characteristics required for specific verification goals. Through test templates, verification engineers can provide a full or partial specification of a required scenario and leave unspecified all the aspects that are not crucial to the scenario. This enables X-Gen to hit a targeted event in a large number of different ways. Thus, when a targeted scenario is defined by the verification engineer in loose terms, X-Gen, through its randomness and testing knowledge, can generate an interesting test around the loosely-defined scenario.

System interactions are the basic building blocks of the test template language. Verification engineers can constrain different interaction attributes, including the identity of the initiator and target(s) that participate in the interaction and their properties.

An interesting interaction in the context of production processes is one concerning a packet released by an operator and ending up in a sink, where a sink is either a scrap heap or a product stock. We call this interaction a *FullPath* interaction. When the operator or the sink are not specified as part of the interaction in the test template, then the decision is left for the random-biased generation engine to determine appropriate specific components.

The test template language contains several high-level statements used to group interactions or other high-level statements: `AllOf` statement generating all of its sub-statements; `OneOf` statement providing a weighted choice between its sub-statements; and `Repeat` statement generating multiple instances of its sub-statements. These high-level statements can be used to group single interactions or to group other high-level statements. Hence, a test template can be viewed as a tree with interactions as its leaves and high-level statements as its intermediate nodes.

In addition to the concepts described above, X-Gen's test-template language provides programming-language-like constructs that enables verification engineers to depict specific scenarios. These constructs include a rich expression language and support for variables with typed declarations, assignments, and constraining the selection of values for actors and properties. This enables the verification engineer to form practically any type of complex relationship among a set of interactions.

Consider the topology design of the working example shown in Fig. 1. Among the interesting behaviors to be tested in the production process are synchronized packets, i.e., packets that are released by the two different operators at the same time or a workload of packets arriving at the two different product stocks with some given ratio. A test template requesting tests that include both of the above-mentioned behaviors is shown in Fig. 4.

```
Integer: T
ComponentId: targetSink
AllOf
  Repeat(400)
    AllOf
      FullPath: {Robot1}; (T <- time); (targetSink <- target)
      FullPath: {Robot2}; (time <- T); (target <- targetSink)
  Repeat(200)
    OneOf
      FullPath: (target <- ProductStock1) weight=10
      FullPath: {target <- ProductStock2} weight=90
```

Fig. 4: Example test template.

The test template in Fig. 4, when consumed by a test generator, should be able to yield many random tests. In each such test, 800 packets will be released by the 2 operators, Robot 1 and Robot 2. Each operator will release 400 packets synchronized with the packets of the other operator and reaching the same destination sink. Note that X-Gen generates the interactions in the same order they appear in the test template. The resolution of variables follows the order of the interactions' generation.

Following that, the test will include the release of 200 packets from random operators, such that 10% of the packets will end up in Product Stock 1 and the remaining 90% in Product Stock 2.

4.2 X-Gen Generation Scheme for Production Processes

For a given test template, X-Gen's generation process can be divided into two layers: traversing the high-level statement tree and generating the interactions at its leaves. Statement tree traversal is done in a hierarchical manner, in which each statement is responsible for the generation of all of its sub-statements. An `AllOf` statement, for example, would generate all its sub-statements, while a `OneOf` statement would randomly pick one of its sub-statements and then generate it.

X-Gen generates an interaction by constructing and solving a constraint network, also known as a *constraint satisfaction problem* (CSP). Testing knowledge is incorporated into the CSP as soft constraints that are activated with some preset probability. The network is solved using a variant of the well-known *maintaining arc consistency* (MAC) algorithm [19, 20].

An interaction is generated in two stages. In the first stage called *path selection*, X-Gen randomly chooses an interaction initiator and target(s). This is done by forming a CSP to choose the path used by the interaction. We create a CSP variable representing the path. The domain of the CSP variable is the set of all paths that initiate from one of the allowed initiators and terminate at one of the allowed targets. Additional constraints may be used to enforce path restrictions; soft constraints may be added to bias toward interesting paths (e.g., always going "left" from a certain Splitter).

Once the path is chosen, a second stage, referred to as *property selection*, constructs a second CSP network. The variables in the CSP are the properties of the interaction, including the properties of a packet. Some components along the chosen path impose restrictions on the properties of the packet and some modify them. The restrictions and modifications are naturally modeled through constraints on respective components. Soft constraints are added to bias random choices to interesting corner cases.

4.3 Tests

A test for the design of a production process is a collection of interactions between component types. The generated test is refined by creating a text file containing specific configurations and instructions to the simulator.

When refined into a text file, a test includes a configuration of the system, its initial state, and sets of instructions for the behavioral components in the system. The configuration of the system is the assignment of values to the various parameters of the different components. The initial state describes the state of the system when the test starts running, e.g., the packets that already exist in-flight in the different stages in the system. The sets of instructions for the initiating and behavioral components are directions for how these components should generate activity in the production process. Each instruction includes a time stamp in which it is to be performed, a command specifying what activity is to be generated, and a set of arguments for that activity.

Consider the topology design in Fig. 1 of the working example and the test template in Fig. 4. A partial snapshot of a possible test that could be produced from the given test template is shown in Fig. 5. The configuration of the shifter is given in the parameter defining the shifted amount, which is set to 10, while the configuration of the first sensor is given in the parameter defining the tolerance threshold, which is set to 3. The synchronized interactions are specified in the template to cause the two operators to put two different packets in the production line at the same time and so that they end up in the same sink. This may yield the first instructions in Robot 1 and Robot 2 in the test. Both packets are released to their corresponding tracks 20 seconds (TS = 20) into the test, and both will arrive at the scrap heap due to untolerated and unfixable leakage.

```
INITIALIZATIONS: Operator:Robot1
------------------------------
B Put Packet=<ID=3,X=48,Y=67> TS=20
⋮
INITIALIZATIONS: Operator:Robot2
------------------------------
B Put Packet=<ID=5,X=77,Y=11> TS=20
⋮
INITIALIZATIONS: Stage:Shifter
------------------------------
C ShiftAmount 10
B Leak Leakage=<ID=3,amount=6>
B Leak Leakage=<ID=5,amount=7>
⋮
INITIALIZATIONS: Stage:Sensor1
------------------------------
C Tolerance 3
⋮
```

Fig. 5: An example test

5 Checking and Coverage Analysis

In simulation-based verification, the simulation of the generated tests aims to reveal errors in the design under test. The occurrence of an error during a test simulation can be detected in various ways. One method is to generate tests that include their corresponding expected results (intermediate and final). This method requires some reference model that is able to "run" the test and predict the expected results. When the test is simulated on the design simulator, the actual results are compared with the expected results from the reference model.

Another approach is to use simulation monitors. A simulation monitor is a software module that observes the progress of simulation and can detect and

report specified behaviors of the simulated model. Simulation monitors have diverse uses such as:

- *Checking monitors*: These monitors check for violations of system requirements as exhibited during simulation. These can originate from the system stakeholder requirements, from the designer, or from the verification engineer. They can include monitors to check for requirements relating to speed, power, heat, and more. They can also check for requirements relating to the functionality of the production system (e.g., that a production stage is performing its designated task correctly).
- *Coverage monitors*: These monitors check for the occurrence of scenarios targeted by the test plan. Coverage monitors are the main vehicle for assessing the progress or completeness of the testing phase. The coverage reports can help the verification engineer focus on uncovered areas or can lead to a managerial decision that enough testing has been done.
- Monitors that gather statistics for post-simulation analysis, such as an analysis of system performance, power consumption, scrap rates, and more.

In the following section, we compare the performance of different test generation approaches using simulation coverage monitors.

We show that our test generation method is able to reach significantly higher functional coverage than a random test generator with a smaller number of tests.

6 Experiments

A SysML model of a production process, as described in Section 2.1 and Fig. 1, was constructed to demonstrate the application of the proposed test-generation technology. The modeled production process topology includes 12 different end-to-end production paths. In addition, for demonstration purposes, we defined the following set of verification events for the production process as testing knowledge in the ontology:

A. **name:** *Almost total shift*
   definition: $Y \in \{0, 1\}$ after the shifter stage.
B. **name:** *Near-equilibrium at start*
   definition: $|X - Y| \leq 2$ when released by a robot.
C. **name:** *Near-equilibrium at end*
   definition: $|X - Y| \leq 2$ when entering a scrap or a product stock.
D. **name:** *Shifter tolerance leakage*
   definition: Leakage of the shifter is exactly the tolerance threshold.
E. **name:** *Injector tolerance leakage*
   definition: Leakage of the injector is exactly the tolerance threshold.

We set the verification goal to cover all the above five verification events for each of the 12 end-to-end production paths. This defines a coverage model with $12 \times 5 = 60$ coverage objectives. However, only 36 of the 60 coverage objectives are in fact possible in the defined production process. For example, event E can

occur only for paths that go through the injector, and only 6 such paths exist. Next, given this coverage model, the following test templates were created. Each template was set to generate a test with 3000 packets. The total number of covered objectives as a function of the simulated packets is shown in Fig. 6 for the 4 test templates. The templates are ordered from the most constrained to the least constrained. As expected, the less constrained test templates, though easier to specify, resulted in a slower coverage rate.

1. The first test template explicitly specifies a test for each of the 60 coverage objectives. The 12 paths were precisely specified and the 5 verification events were requested by invoking the corresponding controls from the testing knowledge database. The activation of the testing knowledge was a non-mandatory specification, which means that the event will occur in the test only if it is possible when considering the mandatory constraints, such as the test validity and the path specifications.
2. The second template specifies only the five verification events from the testing knowledge, while the end-to-end production paths were left to be selected randomly by the test generator.
3. The third template merely directs the generator to generate tests with random packets, while activating the default settings of the testing knowledge.
4. The fourth template is purely random, i.e., un-biased tests will be generated, constrained only to be valid production paths with no application of any testing knowledge.

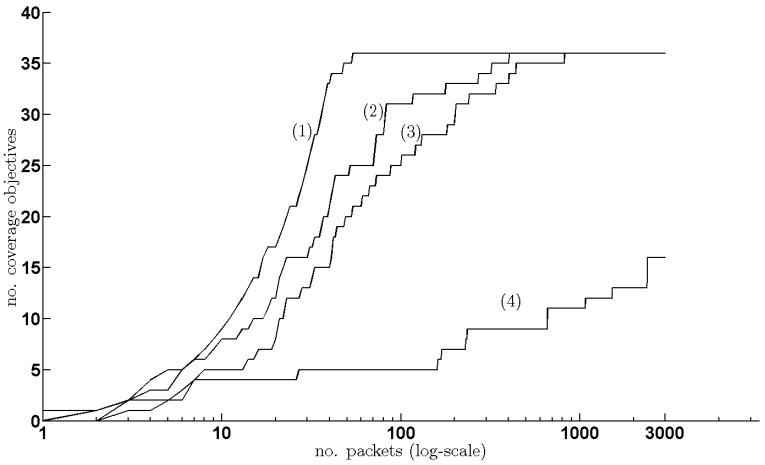

Fig. 6: Coverage progress for the four different test-generation approaches. (1) Directed towards all coverage goals. (2) Directed towards the verification events with random paths. (3) Random tests with default testing knowledge. (4) Random tests with no testing knowledge.

As shown in Fig. 6, the tests generated for the first template were the quickest to reach full coverage of the 36 coverage objectives. The second and third test templates took much longer to reach full coverage than the first. However, these less specific test templates have the advantage of giving more "room" for the test generator to apply its random testing knowledge and thus generate a greater variety of tests. X-Gen's test template specification language allows verification engineers to define templates ranging from fully deterministic to completely random. Yet even the completely random templates will be generated with the default random application of testing knowledge, which biases the random generation toward areas worthy of testing. In addition, Fig. 6 shows that the fourth template, which did not use any testing knowledge, made the slowest progress in coverage, as expected. In fact, some of the events were never covered, even after 3000 packets. These events have a naturally low probability of occurring but may still be important to test. An alternative to our approach would be to fill in these coverage holes by manually writing the missing test cases. However, this is obviously time-consuming and sometimes difficult to achieve. The events may refer to behaviors of the packet flow somewhere in mid-production, and therefore the test creation would require a detailed analysis of the manipulations carried out at the relevant processing stages.

7 Conclusions and Future Work

We presented a new test-generation methodology for system-level verification of production processes. Our methodology offers an alternative to manual and random test generation for functional and performance verification, by producing massive random-biased tests. The tests are generated from test templates stated in a rich high-level template language. The test generator uses a generic system-oblivious engine in addition to a domain-specific ontology and testing knowledge. We described an ontology of a production system using SysML, an industrial-standard modeling language. We built a simulator for this ontology, also using SysML, to run the generated tests. We presented an example of how to model a production process for the purpose of functional and performance testing. We demonstrated our methodology for generating tests that achieve coverage of a set of verification objectives and compared it to the coverage of a random test generator. We showed that our approach can achieve better coverage with a smaller number of tests than random test generation.

In the future, we plan to apply our methodology for test generation on a variety of types of production processes used by the industrial partners of the MuProd project. We expect that this methodology will prove beneficial to the field of production processes' design and verification.

References

1. Teti R. Jemielniak K., O'Donnell G., and Dornfeld D.: Advanced monitoring of machining operations, CIRP Annals - Manufacturing Technology 59, pp. 717–739 (2010)

2. MuProD - http://www.muprod.eu/
3. Köhler H., Nickel U., Niere J., Zündorf A.: Integrating UML Diagrams in Production Control Systems, In: International Conference on Software Engineering, pp. 241–251 (2000)
4. Rother M., Shook J.: Learning to See: Value-Stream Mapping to Create Value and Eliminate MUDA, Lean Enterprise Inst., Version 1.3., Cambridge, Mass., Lean Enterprise Institute (2003)
5. Nickel U., Niere J., and Zündorf A.: The FUJABA Environment, In: International Conference on Software Engineering, pp. 742–745 (2000)
6. Specht T., Drawehn J., Thränert, M., Kühne S.: Modeling Cooperative Business Processes and Transformation to a Service Oriented Architecture. In: 7th IEEE International Conference on ECommerce Technology, pp. 249–256 (2005)
7. T. Wen-xian and X. Yuan-yuan: A Production Process Mixed Modeling for Marine Diesel Engine Based on IDEF0 and Petri Net, Information Science and Engineering, 2008. ISISE '08. International Symposium on , vol.2, no., pp.773–777 (2008)
8. Zor S., Görlach K., and Leymann F.: Using Modeling Manufacturing Processes, In: Sihn, Wilfried (ed.); Kuhlang, Peter (ed.): Sustainable Production and Logistics in Global Networks - Proceedings of 43rd CIRP International Conference on Manufacturing Systems, pp. 515–522 (2010)
9. Campagna D. and A. Formisano: ProdProc - Product and Production Process Modeling and Configuration, In: 26th Italian Conference on Computational Logic, pp.261–279 (2011)
10. Organization for the Advancement of Structured Information Standards (OASIS),Web Services Business Process Execution Language Version 2.0 OASIS Standard (2007)
11. Colledani M., Terkaj W., and Tolio T.: Product-Process-System Information Formalization. In Tolio T. (Ed.), Design of Flexible Production Systems: Methodologies and Tools. Springer (2009)
12. Aharon A., Lichtenstein Y., and Malka Y.: Model-Based Test Generator for Processor Design Verification, In: Innovative Applications of Artificial Intelligence (IAAI) (1994)
13. Emek R., Jaeger I., Naveh Y., Bergman G., Aloni G., Katz Y., Farkash M., Dozoretz I., and Goldin A.: X-Gen: A Random Test-Case Generator For Systems and SoCs, In:7th IEEE International High-Level Design Validation and Test Workshop (HLDVT), pp.145–150 (2002)
14. DANSE - http://danse-ip.eu/home
15. http://www.omg.org/spec/UML/2.4.1/
16. http://www.omg.org/spec/SysML/1.3/
17. Bin E., Ghanayim A., Holtz K., Marcus E., Morad R., Peled O., Rimon M., Shurek G., and Tsanko E.: Ontology-Based Tools in the Service of Hardware Verification, In:22nd International Conference on Software Engineering & Knowledge Engineering , pp. 303–308. Knowledge Systems Institute Graduate School (2010)
18. Brahme D. S., Cox S., Gallo J., Glasser M., Grundmann W., Norris Ip C., Paulsen W., Pierce J. L., Rose J., Shea D., and Whiting K.: The Transaction-Based Verification Methodology. Cadence Berkeley Labs (2000)
19. Bin E., Emek R., Shurek G., and Ziv A.: Using constraint satisfaction formulations and solution techniques for random test program generation, In IBM Systems Journal 41(3), 386–402 (2002)
20. Mackworth A.: Consistency in Networks of Relations, In Artificial Intelligence 8(1), pp.99–118 (1977)