Engineering Functional Requirements of Reactive Systems using Synchronous Languages

Erwan Jahier* - Nicolas Halbwachs* - Pascal Raymond*
*CNRS/Verimag, 2 avenue de Vignate, Gières, France

***Abstract*—Automating the functional testing of reactive systems requires to provide a formal specification of the system environment which defines the admissible test inputs. It also requires a specification of the expected properties of the system in order to decide whether a test succeeds or fails. Engineering these formal specifications is a difficult task, as it is not part of the usual manual testing process. In this paper, we report some experiments that have been conducted within a project in collaboration with industrial developers of nuclear power plant control systems. In this project, automatic testing tools have been used for checking the correctness of reactive systems developed incrementally, using heterogeneous industrial engineering workbenches. But these tools appeared to be useful also for elaborating and refining formal, consistent, and accurate functional requirements.**

I. Introduction

The functional testing of reactive systems raises specific problems. In this paper, we adopt the usual point of view of synchronous programming, considering that the behavior of such a system is a sequence of atomic reactions — which can be either time-triggered or event-triggered, or both —. Each reaction consists in reading inputs, computing outputs, and updating the internal state of the system. As a consequence, a tester has to provide test sequences, i.e., sequences of input vectors. Moreover, since a reactive system is generally designed to control its environment, the input vector at a given reaction may depend on the previous outputs. As a consequence, input sequences cannot be produced off-line, and their elaboration must be *intertwined with the execution of the system under test* (SUT). Finally, in order to decide whether a given test succeeds or fails, the sequence of pairs (input-vector, output-vector) can be provided to an observer [1] which acts as an oracle at each reaction.

Let us illustrate this process with a very simple example: consider a device whose role is to regulate the water temperature of a tank, by opening or closing a gate that controls the arrival of hot water. A reaction consists in sampling the water temperature, comparing it to the target temperature, and sending an order to the gate. In a realistic input sequence, the temperature is assumed to increase at some rate when the gate in open, and to decrease when it is closed. Hence the input at a given reaction depends on the output sent at the previous reaction. The property to be checked could be that, if the target temperature did not change during a given delay, the temperature should belong to a given interval around the target temperature. This global property of the combination [system-environment] can be checked after each reaction by an observer, which must have an internal memory to count the delay from the last target change. Moreover, in order to properly cover this property, we want to generate sequences where the target temperature does not change too often.

In the past, we proposed languages and tools to automate this testing process [2]. The system under test is a black box; it can be any executable code, able to perform on demand a reaction, i.e., read inputs; do a step; provide outputs. The environment is modeled using dynamically changing constraints on inputs described using Lucky [3]; we now use a higher-level language named Lutin [4]. The oracle is provided as an observer in Lustre. The tool Lurette is then able to run automatically any number of arbitrarily long test sequences. Each step consists of (1) executing one (stochastic) reaction of the environment that provides inputs to the SUT (2) executing one reaction of the SUT with the chosen inputs, (3) executing one reaction of the oracle observer with the SUT inputs and outputs (and stopping the test if the checked property is violated), and (4) looping to (1) using the SUT outputs as environment inputs.

These tools have been applied in several projects. In this paper we report on the experiments conducted during the COMON project, which gathered Verimag with 3 industrial partners, each of which using its own development tool for designing specific parts of the control and supervision systems of nuclear power plants. During the project, it appeared that the methodology requires significant efforts to formalize both the assumptions made on the system environment and the required properties checked by the oracle. However, it appeared also that this formalization process is valuable not only for automating the tests, but also as an actual engineering of requirements, allowing early simulations and error detection. This is what is reported in the present paper.

A. Model-based development of correct systems

In order to establish the correctness of a system, one has to confront an implementation to a specification. When the specification is *formal*, that conformity checking can be mechanized.

Any formal specification is rooted on a natural language document. This formalization step is the most delicate, since it cannot be automated, and the most important, since all verifications are based on it. Once the functional requirements have been formalized, even the very first formal model of the final system can be verified automatically. This process is outlined in Fig. 1.

1) Analysis of the needs.
2) Specification of the functional requirements in natural language and via informal drawings.
3) **Formalization of these functional requirements.**
4) **Design of abstract, formal, and refinable models from the requirements.**
5) Design of a final implementation satisfying the requirements.

Steps 1, 2, and 5 are the usual software engineering steps. Steps 3 and 4 can be started in parallel by different teams. The first executable model obtained in step 4 is not complete. Typically, the hardware is abstracted away in the early stages. Ideally, Step 5 is the result of iterative refinements of step 4.

Fig. 1. Model-based development of correct systems.

Two kinds of formal models. The difference between the formal models obtained in steps Fig.1-3 and Fig.1-4 is that the former is only concerned with the "what", whereas the latter is also concerned with the "how" – even if the "how" is introduced step by step. More specifically, in step Fig.1-3, we write a model that just *decides* if the system is correct by observing a sequence of inputs and outputs. The models of step Fig.1-4 are more difficult to set up, since they are *computing* output sequences from input sequences. The advantage of going through an intermediate formal model is to make easier the (informal) demonstration of its correctness with respect to the informal specification of step Fig.1-2. We present in Sections II-A and II-B an automated test approach to check the correctness of models of step Fig.1-4 with respect to the one of step Fig.1-3.

Formalization languages. In order to be convincingly faithful, the formalization of Fig.1-3 has to be as concise and readable as possible; ideally there should be a one-one correspondence with the informal text. Of course concentrating only on the "what" makes it easier. In any case, good formalization languages help. They should be well-suited to the kind of systems under consideration. In particular, for reactive systems, we need languages where time and concurrency are first class concepts. A good set of libraries also makes a great difference. We try to demonstrate our languages and their suitability to define domain-specific libraries of timed properties in Sections III and IV.

Requirements engineering. The first three steps are the result of a human work that is impossible to automate. In Section II-E, we explain how the proposed tools and methodology can be used to complete the document of step Fig.1-2, and to detect inconsistencies.

B. The COMON project and its outcomes

The COMON project[1] consortium was made of Verimag and 3 industrial partners playing complementarity roles in the design of control and supervision systems for nuclear power plants: Atos-Origin designs supervision rooms and human interfaces; Rolls-Royce Civil Nuclear is in charge of critical safety systems; and Corys-Tess designs plant simulators. Each of these partners uses its own workbench: in-house tools for Atos and Corys, and Scade [5] for Rolls-Royce. The motivation for this consortium was to take advantage of the partners complementary to set up a model-based development framework based on early simulations, model refinements, and continuous integration.

In this project, we have demonstrated the use of our tools and languages, based on the synchronous paradigm [6], (1) to check the correctness of reactive systems developed incrementally, using heterogeneous industrial engineering workbenches (2) to elaborate consistent and accurate functional requirements. We have performed experiments where Lurette is controlling the 3 partners workbenches on a custom case study designed to be representative of each partner's usual activity.

It was our first experience with Lurette on such heterogeneous industrial systems. It was also the first time we used Lutin and Lurette together on a case study. Lutin has been extended with a new kind of modularity, which allows to write more efficient and more readable programs. We have also introduced in Lurette a simple concept of oracle coverage. Both this notion of coverage and the enhanced version of Lutin make Lurette a versatile tool to support model-based (or just incremental) developments with early simulation and validation.

C. Summary

The paper is organized as follows. In Section II, we describe the Lurette languages and methodology, and their new features. During the project, while formalizing requirements in Lustre and Lutin, we sketched up a library of generic oracles and stimulators. Most of those new library entries concerned time lasting properties that are typical of real-time control-command systems. Some of them are presented in Section III. Finally in Section IV, we report the experiments conducted in the COMON project, and give a few illustrative examples.

II. From automatic testing to requirements engineering

The reactive systems testing tool Lurette automates the SUT input data generation and the test decision using formalized functional requirements [2]. After a recall of its main components in Sections II-A and II-B, we propose a way to deal with coverage issues in Sections II-C and II-D. The introduction of this coverage mechanism, the use of Lutin, and the experiments performed in the COMON project revealed a new insight into the use of Lurette, as a tool to engineer precise, complete, and coherent functional requirements. This is argued in Section II-E, where we propose a complete tool-assisted methodology to support step by step development of correct reactive systems.

A. Formal outputs requirements (oracles)

Parts of the functional requirements deal with how the system should behave in some particular situations, such as raising an alarm when a threshold is exceeded. In our approach, these formalized requirements will be used as oracles to automate the test verdict, deciding if a sequence of inputs and

[1]The COMON project (2009-2012) was supported by the "Pôle de compétitivité" Minalogic, and funded by the French government, the city and the metropolis of Grenoble.

outputs conforms to the specification. This methodology can be seen as a light-weight way of confronting 2 implementations realized by 2 different teams. The advantage of using oracles over a second implementation is that oracles are easier to set up and maintain. Moreover, the fact that they use a different level of abstraction encourages transverse views which limit the risk that the same error is done twice.

With Lurette, oracles are written as observers in the synchronous data-flow language Lustre [6]. Of course any other synchronous language could be used [7], [8]. The concepts of time and parallel composition are constitutive of synchronous languages [9], and make them particularly suitable for the modular description of reactive systems oracles [10].

B. Formal inputs requirements (environments)

Another part of the functional requirements concern the assumptions made, sometimes implicitly, on the system environment. As a matter of fact, a reactive control system is seldom supposed to work correctly in any environment. A typical example is the one of a railway signaling system whose environment is composed of trains and tracks: the system cannot guarantee the absence of accidents without assuming that the trains follow tracks and stop at red lights. These assumptions, once explicitly expressed as constraints, will enable Lurette to generate pseudo-random realistic input sequences to the SUT.

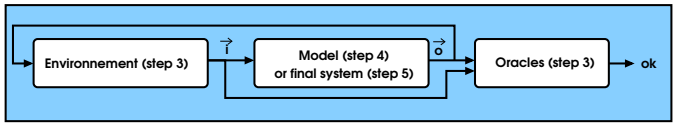

Fig. 2. Lurette data-flow. The 3 entities run synchronously. At each (discrete) instant, the environment model (step Fig.1-3) provides a (realistic) input vector (i) to the model (step Fig.1-4) or system under test (step Fig.1-5), which in turn provides an output vector (o) to its environment. Both vectors (i and o) feed the oracles (step Fig.1-3) that produce a verdict (ok/nok).

The Lurette data-flow is summarized in Fig. 2. The feedback loop between the SUT and its environment is specific to the test of reactive systems, that react to environments they try to control. Because a realistic environment must react to SUT commands, input sequences cannot be generated off-line, as it is done for testing other kind of systems. The language Lutin has been designed to program stochastic and reactive *stimulus generators* (or *stimulators*). Lutin programs are made of Lustre-like data-flow constraints assembled with regular operators (sequence, loop, non-deterministic choice) and exceptions that make easier the description of probabilistic sequential scenarios. Some examples are given in Sections III and IV.

C. Functional coverage

When performing functional testing, structural coverage criteria are insufficient to give insights about whether or not enough tests have been done. We also need coverage criteria attached to requirements. Consider for instance the following property, which expresses that, when a threshold is exceeded, and when the system is in its nominal mode, then an alarm must be raised:

```
(T>100 and nominal) => Alarm
```

There are several ways for the SUT to satisfy this property: (1) `T` can be smaller than 100 or (2) the system can be in a degraded (non-nominal) mode; (3) otherwise, `Alarm` must be true. From the coverage point of view, it is obviously the latter case that is interesting. Its seems fair to consider that this oracle is not covered as long as no simulation has been run where `T>100` and both `nominal` and `Alarm` are true. The case `T<100` is also interesting to cover, but can be attached to a property related to the absence of false alarms.

Hence, we define the *coverage of an oracle* as a set of Boolean conditions. A *run* (or a *trace*) of the SUT is a sequence of the SUT input/output vectors generated during a simulation. The oracle *coverage rate of a set of runs* is the rate of coverage conditions that have been true at least once during those runs. The coverage of a property is arguably part of its specification. If it is not the case, the persons in charge of formalizing requirements into oracles are in the best position to define the coverage at the same time.

In Lurette, in order to define the coverage of an oracle, one just needs to add additional Boolean variables to its output profile. By convention, the first output holds the oracle result, and the following outputs define the oracle coverage. Lurette updates the coverage rate from one execution to another (via a file). This coverage rate is reset each time either the oracle or the SUT is modified.

D. Better coverage via test scenarios

In order to cover the property of Section II-C, we need to drive the SUT in such a way that `T` exceeds 100 while maintaining it in nominal mode. More generally, achieving full coverage of conditions that depend only on SUT inputs is easy. This is more difficult when these conditions involve outputs or internal states of the system. One must set up scenarios that put the system in specific configurations, which generally requires a good expertise of the SUT (expertise that is also necessary for traditional testing methods). The design of Lutin was mainly motivated by the need to easily express elaborated constrained random and sequential scenarios; indeed, a purely data-flow language such as Lustre, which was used in early versions of Lurette, is not always convenient for describing sequential scenarios and to assign them probabilities. We illustrate in Section IV-D how Lutin can be used to increase the coverage.

E. The Lurette overall process

The design of the system, its oracles, and its stimulus generators is not a linear process. Several iterations are necessary, which we describe below, and outline in Fig. 3.

Refining the SUT. When an oracle is violated, it can be, of course, because of a design or coding error, which results in a erroneous SUT. Detecting such incorrect behaviors of the SUT is indeed the original motivation of all this infrastructure.

Refining oracles. An oracle violation can also be due to a wrong formalization. Despite the fact that sequence recognizers (oracles) are much simpler to develop than sequence generators (SUT), they are still the result of a human work and thus exposed to errors.

Refining ambiguous requirements. Lurette can also detect ambiguous requirements, when they are interpreted differently by the SUT and the oracle designers. It happened quite frequently during the project.

Refining inconsistent requirements. Formalizing requirements in a language such as Lustre that is equipped with a model-checker allows to detect inconsistencies, i.e., the absence of correct behaviors.

Refining imprecise requirements. Another reason that results in invalidated oracles is when they are based on imprecise requirements. One typical case encountered in the project was a requirement formulated as follows: "when `x` exceeds the threshold `t`, the alarm `a` should be raised". In a distributed system like the one of COMON, where sub-systems communicate over buses and networks, such a requirement will be immediately violated if interpreted literally. One should permit some reaction delay, and specify its bounds.

Refining incomplete requirements. Another very common source of oracle violations is a lack of completeness. A typical example, also encountered during the project, is a requirement that states that "the temperature of tank `t` should never exceeds `100` degrees", whereas the correct requirement was "the temperature of tank `t` should never exceeds `100` degrees when the system is in the nominal mode and the validity bit associated to its sensor is 1".

One outcome of the project is that the Lurette tool and methodology was actually helpful for debugging and refining requirements.

Refining scenarios. When the coverage is not complete, we must enrich the set of possible behaviors of the environment with new scenarios. Note that new scenarios may lead to properties violations, which lead to changes in the SUT (or in oracles); and changes in the SUT may change the coverage in turn.

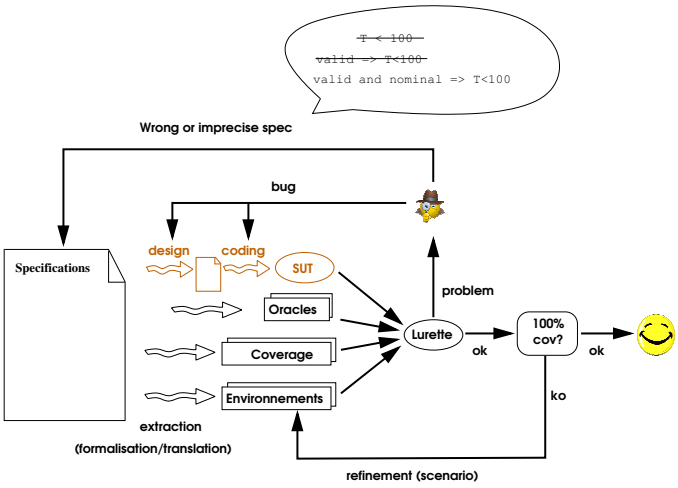

Fig. 3. The Lurette iterative process loops. Oracles and environments of the SUT are extracted from heterogeneous specifications. When an oracle is invalidated, it can be due to a design error, a coding error, or to a wrong or imprecise specification. Once the system is running without invalidating oracles, in order to improve the coverage rate, the tester needs to refine test scenarios.

F. Non-regression

The initial elaboration of fully formalized and covered requirements imposes a significant work. But obtaining a fully automatic testing is not the only advantage: it also provides automatic non-regression tests (almost) for free, when the first validated model is refined, or when the final system implementation changes. Of course, during system evolutions, the full coverage of requirements might be lost, requiring some additional work at the stimulators side. Requirements are also subject to changes, that need to be formalized and covered. But oracles and stimulators dealing with unchanged parts of the system can be reused as they are. Moreover it is unlikely to be the case for minor changes, which can therefore be tested automatically and continuously.

III. Crafting a library of generic oracles and stimulators

Some properties, involving combinations of Boolean variables and time, can be tricky to define. However, most of them are similar. This is why, in order to make easier the formalization of functional requirements, it is important to rely on a set of correct and well-documented libraries. We first present here, among the oracles and stimulators we defined to hold the COMON experiments, the ones that have a more general scope and that are therefore likely to be part of a library. The rationale is to show how such a library can be built easily, and also to illustrate how our languages are good at that. No prior knowledge of Lustre and Lutin should be necessary, as we paraphrase every program. We've also tried to start with simple programs in order to introduce language concepts smoothly.

Lutin and Lustre share the same basic principles: data-flow, logical time, stream operators, synchronous composition. They share the same syntax, since Lutin has been designed as an extension of Lustre with probabilities and regular expressions, to make easier the description of sequential stochastic scenarios. Indeed, many properties are used both for the definition of oracles and stimulators.

A. Formalizing quantitative-time properties

Many reactive systems properties involve time issues. Lustre is based on a *logical notion* of time (i.e., the sequence of program reactions), but in the COMON case study, many properties make use of *quantitative time*. Since the various subsystems activations are supposed to be periodic, the connection between logical and quantitative time is straightforward. However, it can be made easier thanks to a library of dedicated operators (called "nodes" in Lustre). In order to ease the formalization of such kind of temporal properties, a good set of dedicated nodes, with carefully chosen names, can help to convince that the formalization is faithful.

A first Lustre oracle. In order to convert quantitative time into logical time when dealing with periodic tasks, one just needs to know the activation period. For example, in order to make more readable the formalization of properties that refer to time intervals where Boolean signals remain true, one can define this kind of Lustre node:

```
node true_since_n_seconds(n: real; signal: bool)
returns (res: bool)
var timer : real;
let
  timer = n -> if not signal then n else
              max(0.0, pre(timer)-1000.0*cycle_time);
  res = (timer = 0.0);
tel
```

This node takes as input a real `n` that holds a number of seconds (that typically never changes), and a Boolean `signal`; the Boolean output `res` is set to `true` at all instants where `signal` was continuously `true` during the `n` preceding seconds. To define this node, we first define a local variable `timer` that is set to `n` at the first instant (`n ->`), and reset to `n` at each instant where `signal` is false. When `signal` is true, we simply decrease the `timer` of the constant `cycle_time`, supposed to hold the activation period in milliseconds. The `pre` operator gives access to the value of a variable at the previous instant.

The advantage of going through the `cycle_time` constant is to be able to express temporal properties regardless of the activation rate. We could easily do the same for non-periodic tasks, by passing time-stamp (`t:real`) as input, and by subtracting to the timer the time elapsed since the previous event (`pre(timer)-(t-pre(t))`).

Once declared, such an operator may be used in defining properties as oracles: for instance, to specify that, whenever the variable `x` has been invalid for 0.5 seconds, an alarm should be emitted, one would write:

```
p = true_since_n_seconds(0.5,invalid_x) => alarm_x
```

A first Lutin stimulator. The Lutin language is based on regular expressions. Literals are constraints expressed in a Lustre-like syntax that relate inputs, outputs and memories (`pre`). Constraints define the possible set of outputs for one logical instant. Instants can be chained using the concatenation (`fby`, standing for "followed by") and the Kleene star (`loop`) operators. The probabilistic (`|`) and the mandatory (`|>`) alternations are used to describe different possible scenario. As in Lustre, Lutin programs are structured into nodes that can be `run` in parallel, and operate over flow of logical instants. The following simple (and deterministic) Lutin node generates a flow of 100 logical instants where the integer variable `x` is equal to 42.

```
node main() returns(x:int) = loop [100] x=42
```

In contrast with a Lustre program which never terminates, a Lutin program may stop since there is no implicit outer `loop`. For stimulators also, it can be more convenient to think in terms of quantitative time. Thus we define a macro `minutes` that converts logical time into quantitative time, independently of the underlying cycle rate.

```
let minutes(x:int):int = x*60*1000/cycle_time
node main() returns(x:int) = loop [minutes(5)] x=42
```

B. Stimulating numeric variables

Now we demonstrate how numeric value generators can be defined in Lutin. We begin with a simple program that we refine, to introduce the language concepts one by one.

```
node gen_x_v1() returns (x:real) =
  loop 0.0<x and x<42.0
```

The node `gen_x_v1` generates an infinite sequence of real values uniformly distributed within the interval]0;42[. If it is more appropriate to bound the derivative, one just need to add the constraint `abs(x-pre(x))<a_bound`. Sometimes it makes more sense to maintain the drawn value for a certain amount of time:

```
node gen_x_v2() returns (x:real) =
  loop { (0.0<x and x<42.0) fby loop [20] x = pre x }
```

The node `gen_x_v2` generates a sequence of values such that: the first value is chosen randomly within]0;42[; that value is then maintained during 20 cycles. At instant 22, another random value is chosen, and so on forever, thanks to the outer `loop`.

```
node gen_x_v3() returns (target:real; x:real=0.0) =
  run target := gen_x_v2() in
  loop { x = (pre x + target) / 2.0 }
```

The node `gen_x_v3` generates 2 flows of reals: `target` is generated using the `gen_x_v2` node; and `x` tries to follow the value of `target` smoothly. The visualization of a run of this node is shown in Fig. 4.

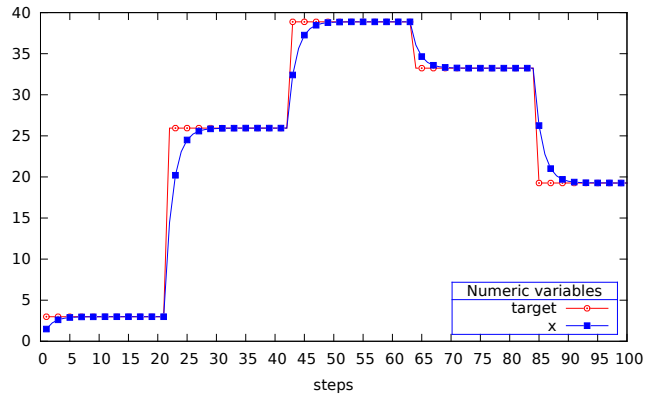

Fig. 4. Visualization of a 100 steps run of `gen_x_v3` (and `gen_x_v2` for the target).

```
node gen_x_v4()  returns (target:real; x:real=0.0) =
  run target := gen_x_v2() in
  exist px,ppx : real = 0.0 in
  loop {
      px = pre(x) and ppx = pre(px) and
      x = (px+target)/2.0+inertia*(px-ppx)
  }
```

The node `gen_x_v4` is a variant of `gen_x_v3` where `x` follows its target with some inertia. To do that, we introduce 2 registers `px` and `ppx` (initialized to 0.0) to store the 2 previous values of `x`. A run of this node is visualized in Fig. 5.

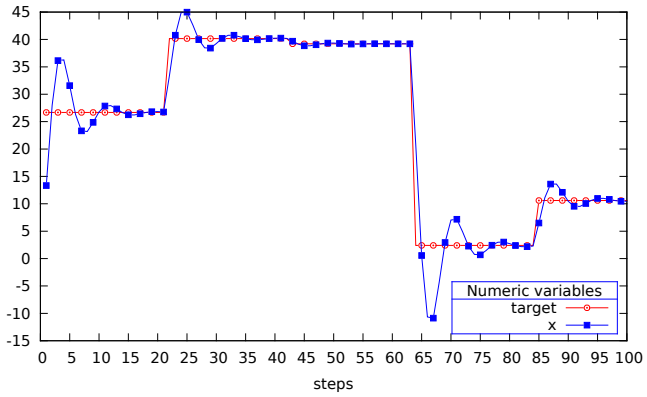

Fig. 5. Visualization of a 100 steps run of `gen_x_v4` with a `0.6` inertia.

IV. Some examples from the COMON case study

We now focus on oracles and stimulators that are specific to the COMON case study. Since a detailed description of the case study is not necessary to understand the examples, we only give a brief overview to help to understand the context.

A. The COMON case study

This case study has been developed from scratch during the project, with the objective to be representative of each partner's activity. It consists of a hydraulic circuit made of a heating and a cooling source, pipes, tanks, sensors, and actuators (cf Fig. 6). Some of these components are redundant. This circuit is regulated by classified (safety-critical) and unclassified control systems. The SUT monitors the system internal behavior (tank levels, temperatures, alarms) and it is driven by human operators (e.g., open a gate to produce more power). We developed several versions of this case-study, corresponding to different level of refinements. One is entirely simulated using the Alices workshop of Corys [11]. Then we have different versions where the other sub-systems are integrated one by one. The SCADA (Supervisory Control And Data Acquisition) of Atos is meant to be embedded in the control room of the power plant. The C code of Rolls-Royce generated by the Scade compiler is also meant to be embedded in the plant. At the end of this integration process, the only simulated parts were the physical process of the plant, and the unclassified automatisms (in the absence of partner specialized in this part in the consortium). This SUT has 58 Boolean and 22 numeric inputs (24 of them being devoted to fault injection); it has 748 Boolean and 108 numeric outputs. Note that they are not all outputs of whole the system; lots of them are at the interface of the 4 sub-systems (process, automata, classified automata, operator interaction).

Connecting Lurette, Alices and Scade was not difficult, since they share with Lurette the same logical (discrete) view of time. The connection with the Atos workbench required more work, since it only knows about real-time and reacting to events. In order to connect a time-triggered system with an event-triggered one, we have added a connection layer that performs sampling in one way, and monitoring to generate an event as soon as a value change in the other way. We choose to set at 4 hertz the communication rate between the 4 workbenches (i.e., 4 steps per second), which is a standard value in this domain. Hence, when the SCADA is connected to the test-bench, each entities should be able to react at this rate. Lutin generators can easily reach that rate, as the Lutin programs presented in this section are able to work at more than 2000 hertz on an ordinary PC. Lurette drives this heterogeneous platform by simulating the human operator behavior, and by injecting faults to circuit elements. Several testing experiments have been conducted, from integration tests to unit tests, going through sub-system tests[2]. The examples below are all extracted from those experiments.

B. States exploration via constraint-based scenarios

Since the SUT is based on a power plant simulator designed, among other things, to train operators, it is possible to set component failures. For instance, the generic gate provided by the Alices library can have more than 20 malfunctions. Failure injections such as the leaking percentage of a pipe, can be parametrized by numeric values, which can be controlled, for instance, by the stimulators presented in Section III-B.

In order to illustrate how constraint-based formalizations can be exploited to explore (pseudo-)randomly a large number of realistic environment scenarios, we propose to describe a subset of the failures generator used for the COMON case study (in the case study, we implemented 24 possible faults; but each Alices object could have several tens of faults, hence we could have had several hundreds of faults in the case study). It is a Lutin stimulator which aims at generating `n` possible failures while avoiding as much as possible to trigger some classified actions. Indeed, once a classified action is fired, the SUT is set into a safe mode, and setting it back to its nominal mode is a long and tedious process. A classified action must be fired, e.g., when at least one redundant element is faulty, or when at least three of quadruple-redundant element are faulty.

Let us consider for instance a dual-redundant and a quadruple-redundant sensor, whose fault statuses are held by 6 Boolean variables (`F1` to `F6`). By default in Lutin, the world is chaotic and unconstrained variables are chosen randomly. Hence, this simple Lutin node tosses at each step a value for each of the 6 possible faults:

```
node gen_failure() returns(F1,F2,F3,F4,P5,F6:bool)=
 loop { true }
```

In order to be able to analyze the consequence of a failure, it can be more convenient to make this stimulator less random, by preventing it to change of failures at each step. For example, we can force it to keep the chosen failures during between 2 and 5 minutes before performing another toss:

```
loop {
 true -- failures toss step
 fby loop [minutes(2),minutes(5)]
    F1=pre(F1) and F2=pre(F2) and F3=pre(F3) and
    F4=pre(F4) and F5=pre(F5) and F6=pre(F6)
}
```

Note that in Lutin, to state that a variable `F` should keep its previous value, we need to explicitly write a constraint

[2]Public materials (in French) coming from the COMON project including a video of the demonstration is available at http://comon.minalogic.net/.

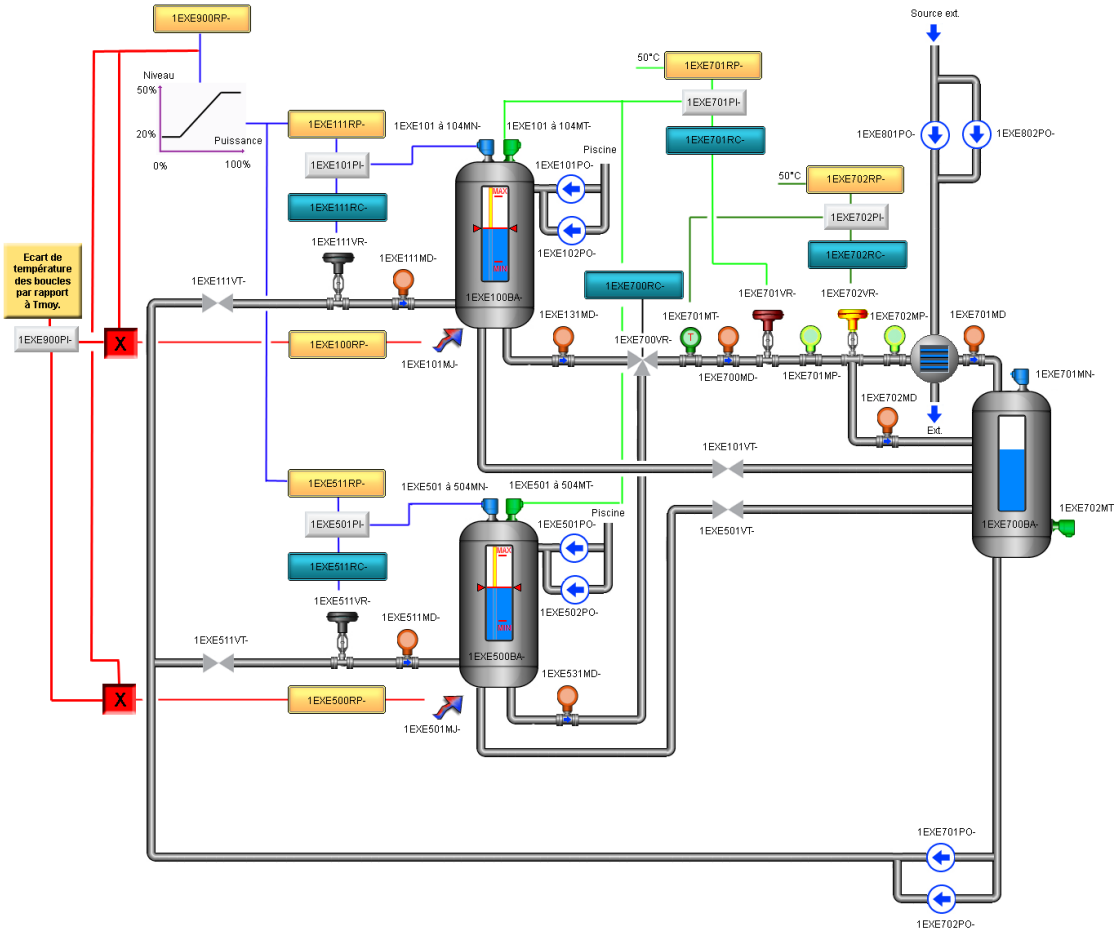

Fig. 6. The COMON case study hydraulic circuit, featuring pipes, tanks, gates, sensors, heating sources and heating sinks.

(`F=pre(F)`). In order to make easier the writing of such constraints, we can use the following freezing combinator (the `ref` keywords declares that the input have memory):

```
let frz(x: bool ref) = (x = pre(x))
```

In order to avoid as much as possible the firing of classified actions, we use a formal version of the firing conditions using the combinator `at_least_1_on_2` (resp. `at_least_3_on_4`) that is true if at least 1 (resp. 3) of its 2 (resp. 4) Boolean inputs is (are) true.

```
let classif_action = at_least_1_on_2(F1,F2) or
                     at_least_3_on_4(F3,F4,F5,F6)
in
```

Then we can use this constraint (instead of the `true` one) to generate random failures without firing classified actions:

```
loop { (not classif_action) fby loop ... }
```

One may also wish to define interactive sessions where the tester (or other Lutin programs) can choose the number `n` of failures to generate. To do that, let's add an integer input to `gen_failure`, and write the following program:

```
node gen_failure(n:int)
returns(F1,F2,F3,F4,F5,F6:bool) =
 let b2i(b:bool) = if b then 1 else 0 in
 let f_nb = b2i(F1)+b2i(F2)+b2i(F3)+
            b2i(F4)+b2i(F5)+b2i(F6)
 in
 loop {
   { |> f_nb = n and not classif_action
     |> f_nb = n
   }
   fby loop ...
```

The combinator `b2i` allows to define the number of failures `f_nb`. The mandatory choice operator (`|>`) states that a constraint should be chosen in priority, if it is satisfiable. Here, we try to generate `n` failures without firing classified actions. But of course, if `n` is greater or equal to 5, the constraint `f_nb = n and not classif_action` cannot be satisfied. In such cases, we use the constraint `f_nb = n`, which is always satisfiable.

If we sum-up the evolutions of the `gen_failure` node we've just described, plus the possibility to reset at any time the choice of failures (to make it more interactive) using an additional input (`reset:bool`), we obtain the following stimulator:

```
node gen_failure(n:int; reset:bool)
returns(F1,F2,F3,F4,F5,F6:bool) =
 let classif_action = at_least_1_on_2(F1,F2) or
                      at_least_3_on_4(F3,F4,F5,F6)
 in
 let b2i(b:bool) = if b then 1 else 0 in
 let f_nb = b2i(F1)+b2i(F2)+b2i(F3)+
            b2i(F4)+b2i(F5)+b2i(F6) in
 loop {
   {
     |> f_nb = n and not classif_action
     |> f_nb = n
   }
   fby loop [minutes(0),minutes(5)] {
     not reset and frz(F1) and frz(F2) and frz(F3)
               and frz(F4) and frz(F5) and frz(F6)
   }
 }
```

Whenever `reset` is true, `not reset` becomes false, and thus the inner loop constraint becomes false. Hence the control goes back to the outer loop, where the first cycle is devoted to another failures toss.

C. Monitoring the system stability

One of the most important properties of the system concerns its stability when no new order is performed by the operator: "In the nominal mode, after any order change, all sensor values must be stable after five minutes". One way of formalizing this requirement is to define the following Lustre node:

```
node check_stability(
   system_is_stable,nominal,no_order_changes:bool)
returns(ok,C:bool);
let
  C = true_since_n_seconds(300.0,no_order_changes)
      and nominal ;
  ok = (C => system_is_stable);
tel
```

This node has 3 Boolean inputs. The first one indicates if the system is stable, which means that no sensor value changed (too much) for a certain amount of time. The numeric tolerance on sensors and the necessary amount of time to consider the system as stable should be specified somewhere in order to be able to define the `is_stable` node (which is part of the library we've sketched up during the project). The second Boolean indicates if the system is in the nominal mode. The third one indicates if at least one order changes at the current instant (`Oi<>pre(Oi)`). The node `true_since_n_seconds` is the one of Section III-A. Using this node, formalizing this stability property is straightforward, which illustrates that faithfulness of a formalization relies on a good set of libraries.

As far as coverage is concerned, there are 2 ways to satisfy this property. First, the premise of the implication could be false; it suffices that the orders (controlled by Lurette) change at each cycle, or to never be in the nominal mode. Another way is to stimulate the system in such a way that no order change for at least 5 minutes in the nominal mode. The latter case is obviously more interesting, and the condition `C` is therefore an evident candidate for coverage cases. The variable `C` is an output of this oracle since is it the Lurette technical convention for defining coverage. Another thing that is necessary to cover this oracle is to check that at least one order changes during a run. One could even want to check that each possible order is changing at least once, and that after such a change, no order changes during at least 5 minutes. In order to define this coverage case for the order `order1`, one can proceed as in the following Lustre specification:

```
oder1_changed = false ->
  (pre(order1_changed) or (order1 <> pre(order1)));
Cov_order1 = true_since_n_seconds(300.0,
               no_order_changes and order1_changed);
```

We define an auxiliary variable `order1_changed` that is false at the first instant, and true if and only if `order1` changes at the current instant, or has changed in the past. Then we can define the coverage case `Cov_order1` using the `no_order_changes` variable, and the `true_since_n_seconds` node. This example illustrates that a language with modularity and time as first-class concept is necessary to define coverage cases conveniently. In order to fully cover all those conditions, generating random data will not be enough. Explicit scenarios, where some orders change from time to time, but not too often, are necessary. This is the purpose of the stimulator considered in next Section.

D. Specific scenarios to increase the coverage

In order to automatically test this open system, we need to close it by simulating its environment, that is to say, we need to model the behavior of the human operator. The operator can set various orders such as a request for a particular tank temperature, a pipe rate, or the opening percentage of a gate. Here we propose to focus on a particular numeric order: the temperature of a tank. When the operator performs such kind of orders, the system automatically open gates, to augment the heating temperature until the desired temperature is reached. But operator requests should respect some constraints. In particular, it is forbidden to request a too abrupt increase. The temperature order should be driven step by step, by requesting a bounded increase, waiting for the system stabilization, and so on until the targeted temperature is reached.

To define this virtual operator, we use a node that outputs an order `O` that changes step by step scoring a target. It takes as input a Boolean wire indicating if the system is stable, the `Target` to score, and a variable `O_init` used to provide an initial value to `O`.

```
node change_order_stepwisely(
         is_stable:bool; O_init,Target:real)
returns (O:real=O_init) =
 loop {
   -- (1) Waiting for the stabilization
   loop { not is_stable and O=pre(O) }
   fby -- (2) Choosing a new order O
     assert Abs(O-pre(O)) < a_bound in
      { |> O = Target
        |> if Target < pre(O) then O < pre(O)
                              else O > pre(O) }
   fby -- (3) Waiting for the change to take effect
   loop { is_stable and O=pre(O) }
 }
```

That node is made of an infinite loop that operates into 3 stages: (1) O keeps its previous value while waiting for the system stabilization. Indeed, the semantics of the Lutin loop construct is to loop as long as possible. Hence, when is_stable becomes true, the constraint becomes unsatisfiable, and the control passes in sequence to the right-hand-side of the first fby (2) Then O is chosen to get closer to Target, but not too rapidly. To do that, we first state that O should not change more than the constant a_bound (using the assert/in construct that propagates a constraint into a scope). Then we use the priority choice operator of Lutin (|>) that forces the use of its left-hand side constraint if it is satisfiable, and uses the right-hand-side otherwise. Hence, if pre(O) is close enough to Target, the constraint O=Target is used. Otherwise, O=Target is not satisfiable, and O will either increase (O>pre(O)) or decrease (O<pre(O)) to score Target. (3) Then we wait for the order change to be propagated into the system, which we detect using the is_stable falling edge. Now we can define our virtual operator, that chooses a new Target whenever the order reaches it:

```
node operator(O_init:real; is_stable:bool)
returns(O:real=O_init) =
exist Target:real in
 loop {
   --Choose the next target order
   0.0 <= Target and Target <= 100.0 and
   O = pre(O)
 fby -- try to reach the target step by step
   run O := change_order_stepwisely(
                is_stable, pre(O), pre(Target))
   in
   loop (Target=pre(Target) and O <> Target)
 }
```

This node is made of an infinite loop repeating 2 stages. The first stage lasts one instant and chooses a random value between 0 and 100 for Target (O keeps its previous value). During the second stage, Target keeps its previous value, and the order is set by the change_order_stepwisely node via the run statement. This statement creates a new reactive machine, having its own memory. It runs in parallel until the control exits its scope when the inner loop terminates, i.e., when O reaches the Target.

V. Related Works

An abundant literature deals with Model-based testing (MBT) in general [12], on embedded systems in particular [13]. The idea of MBT is to bypass the fact that complex system are not analyzable by rooting their analysis on formal models of the system. Then they use state-of-the art formal verification techniques (theorem proving, model-checking, constraint solving) to automate various kind of tests. Lurette oracles somehow falls into this category, as we rely on a SUT model. But this model is very abstract, and does not try to model the behavior of the SUT at all. Lurette, in particular in its current version featuring Lutin and oracle coverage, focuses on environment modeling. Therefore in the following, we only consider works that care about input generation and take into account the feedback effect, which is mandatory for testing reactive systems, thus also referred to as *reactive testing*.

The Model-Based Statistical Testing (MBST) approach [14], [15] consists in modeling the SUT as a Markov chain to describe its usage model. This model (an automaton labeled with probabilities [16]) can be traversed to generate realistic SUT inputs. Recent extensions of MBST [17] targets embedded systems by taking time and concurrency into account via the use of Petri-nets. Test input generation is not the primary objective when setting-up a faithful usage model, and can only generate a finite set of Boolean signals. Moreover, they are not meant to test borderline cases. Note that Lutin can be seen as a language to program stochastic processes, where the emphasis is put on simulation, whereas Markov chains were invented for system analysis (e.g., to figure out what are the transient or the recurrent states).

Lutess [18] is similar to Lurette. The main differences are that the environment is modeled in Lustre, and it requires a Lustre model of the SUT (to guide the input generation). Lutess also has a notion of property coverage [19], which is not an explicit formalization of some requirements as in our proposal. Lutess coverage is defined in terms of suspect states of the automaton resulting from the product of the SUT model, the environment model, and the property to check (i.e., the oracle). A state is *suspect* if there exists an outgoing transition that would lead to a bad state (violating the property) if the SUT does not behave correctly. Hence the coverage is a semantic consequence of the property, that relies on the SUT model faithfulness. More recently, members of the Lutess team have defined some concepts of structural coverage for data-flow Lustre/Scade programs [20], [21]. But note that, in the present article, we do not specifically target systems designed in Lustre/Scade and suppose that the SUT is a black-box.

GATeL [22] uses graph exploration and constraint solving techniques to generate input sequences that target a (formal) test objective. For SUT implemented in Lustre, GATeL should be able to use our coverage conditions and generate the SUT inputs sequences that cover them. Another manner to use white-box technique to automatically generate inputs that increase the coverage would be to do the same as in [23], where a graph exploration tool based on abstract interpretation was used in combination of Lucky (the predecessor of Lutin) to reach a particular symbolic state.

In this work, we consider the functional coverage as a part of the specification that should be explicitly defined. Some works use metrics to generate such kind of coverage automatically from requirements expressed in LTL [24].

Our approach fits particularly well with agile development methods [25], [26]. Three teams - development, oracle, stimulator - can work in parallel, from the initial stages of development. Actually the whole idea of model-driven development and virtual prototyping, where one tries to obtain early executable models of the system to implement, is in the spirit of agile methods.

VI. Conclusion

We propose a methodology, supported by synchronous programming languages and tools, to check systems correctness, and to engineer consistent and accurate functional requirements. The methodology particularly suits incremental and model-based development. We have illustrated this with

examples extracted from a case study designed by industrial partners to be representative of their concerns.

In some sense, this work justifies the relevance of formal verification, as it makes formal specifications usable even when the SUT is not analyzable because of state explosion. Hence this work can be seen as a mean to get a foot in the door, and encourage industry to use formal methods.

This work also demonstrates how synchronous technologies can be useful even when using non-synchronous workbenches.

REFERENCES

[1] N. Halbwachs, F. Lagnier, and P. Raymond, "Synchronous observers and the verification of reactive systems," in *AMAST'93*. Twente, The Nederlands: Springer Verlag, jun 1993.

[2] E. Jahier, P. Raymond, and P. Baufreton, "Case studies with Lurette V2," *STTT*, vol. 8, no. 6, pp. 517–530, 2006.

[3] P. Raymond, E. Jahier, and Y. Roux, "Describing and executing random reactive systems," in *SEFM'06, 4th IEEE International Conference on Software Engineering and Formal Methods*, Pune, India, sep 2006.

[4] P. Raymond, Y. Roux, and E. Jahier, "Lutin: a language for specifying and executing reactive scenarios," *EURASIP Journal on Embedded Systems*, vol. 2008, 2008.

[5] SCADE, http://www.esterel-technologies.com/scade/.

[6] N. Halbwachs, *Synchronous programming of reactive systems*. Kluwer Academic Pub., 1993.

[7] P. Caspi, G. Hamon, and M. Pouzet, *Real-Time Systems: Models and verification — Theory and tools*. ISTE, 2007, ch. Synchronous Functional Programming with Lucid Synchrone.

[8] L. Mandel and M. Pouzet, "ReactiveML, a reactive extension to ML," in *Proceedings of 7th ACM SIGPLAN International conference on Principles and Practice of Declarative Programming (PPDP'05)*, Lisbon, Portugal, Jul. 2005.

[9] A. Benveniste and G. B. (Ed.), "Another look at real-time programming," *Special Section of the Proceedings of the IEEE*, vol. 79, no. 9, Sep. 1991.

[10] G. Durrieu, H. Waeselynck, and V. Wiels, "Leto - a lustre-based test oracle for airbus critical systems," in *FMICS'08, L Aquila, Italy, September 15-16, 2008, Revised Selected Papers*, ser. LNCS, D. D. Cofer and A. Fantechi, Eds., vol. 5596. Springer, 2008, pp. 7–22.

[11] ALICES, http://www.corys.com/ALICES-Workshop--462.html.

[12] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, *Model-Based Testing of Reactive Systems: Advanced Lectures (LNCS)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[13] J. Zander, I. Schieferdecker, and P. J. Mosterman, *A Taxonomy of Model-based Testing for Embedded Systems from Multiple Industry Domains*. CRC Press, 2011, ch. 1, pp. 3–22.

[14] W. Dulz and F. Zhen, "Matelo - statistical usage testing by annotated sequence diagrams, Markov chains and TTCN-3," in *QSIC'03*. IEEE Computer Society, 2003, pp. 336–342.

[15] J. Carter, L. Lin, and J. Poore, "Automated functional testing of Simulink control models," in *Proceedings of 1st Workshop on Model-based Testing in Practice (MoTiP 2008)*, T. Bauer, H. Eichler, and A. Rennoch, Eds. Fraunhofer IRB Verlag, 2008.

[16] S. Prowell, "TML: A description language for Markov chain usage models," *Information and Software Technology*, vol. 42, no. 12, pp. 835–844, September 2000.

[17] F. Böhr, "Model based statistical testing of embedded systems," in *IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW'11)*, march 2011, pp. 18 –25.

[18] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon, "Lutess: a specification-driven testing environment for synchronous software," in *Proceedings of the 21st international conference on Software engineering*, ser. ICSE'99. New York, NY, USA: ACM, 1999, pp. 267–276.

[19] I. Parissis and J. Vassy, "Thoroughness of specification-based testing of synchronous programs," *IEEE 21st International Symposium on Software Reliability Engineering*, vol. 0, p. 191, 2003.

[20] A. Lakehal and I. Parissis, "Structural coverage criteria for Lustre/Scade programs," *Softw. Test., Verif. Reliab.*, vol. 19, no. 2, pp. 133–154, 2009.

[21] V. Papailiopoulou, A. Rajan, and I. Parissis, "Structural test coverage criteria for integration testing of lustre/scade programs," in *Proceedings of the 16th international conference on Formal methods for industrial critical systems*, ser. FMICS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 85–101.

[22] B. Marre and B. Blanc, "Test selection strategies for lustre descriptions in gatel," *Electr. Notes Theor. Comput. Sci.*, vol. 111, pp. 93–111, 2005.

[23] E. Jahier, B. Jeannet, F. Gaucher, and F. Maraninchi, "Automatic state reaching for debugging reactive programs," in *AADEBUG'03 – Fifth International Workshop on Automated Debugging*, Ghent, Sep. 2003.

[24] M. Staats, M. W. Whalen, A. Rajan, and M. P. Heimdahl, "Coverage metrics for requirements-based testing: Evaluation of effectiveness," in *Proceedings of the Second NASA Formal Methods Symposium*. NASA, April 2010.

[25] P. Abrahamsson, J. Warsta, M. T. Siponen, and J. Ronkainen, "New directions on agile methods: a comparative analysis," in *Proceedings of the 25th International Conference on Software Engineering*, ser. ICSE'03. Washington, USA: IEEE Computer Society, 2003, pp. 244–254.

[26] H. Baumeister, "Combining formal specifications with test driven development," in *XP/Agile Universe*, ser. LNCS, C. Zannier, H. Erdogmus, and L. Lindstrom, Eds., vol. 3134. Springer, 2004, pp. 1–12.