Information and Software Technology 57 (2015) 169–185

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Testing robot controllers using constraint programming and continuous integration

Morten Mossige [a,b,c,*], Arnaud Gotlieb [b,1], Hein Meling [c,2]

[a] ABB Robotics, 4349 Bryne, Norway
[b] Simula Research Laboratory, Lysaker, Norway
[c] University of Stavanger, 4036 Stavanger, Norway

ARTICLE INFO

Article history:
Received 24 March 2014
Received in revised form 19 September 2014
Accepted 19 September 2014
Available online 2 October 2014

Keywords:
Constraint programming
Continuous integration
Robotized painting
Software testing
Distributed real time systems
Agile development

ABSTRACT

Context: Testing complex industrial robots (CIRs) requires testing several interacting control systems. This is challenging, especially for robots performing process-intensive tasks such as painting or gluing, since their dedicated process control systems can be loosely coupled with the robot's motion control.
Objective: Current practices for validating CIRs involve manual test case design and execution. To reduce testing costs and improve quality assurance, a trend is to automate the generation of test cases. Our work aims to define a cost-effective automated testing technique to validate CIR control systems in an industrial context.
Method: This paper reports on a methodology, developed at ABB Robotics in collaboration with SIMULA, for the fully automated testing of CIRs control systems. Our approach draws on continuous integration principles and well-established constraint-based testing techniques. It is based on a novel constraint-based model for automatically generating test sequences where test sequences are both *generated* and *executed* as part of a continuous integration process.
Results: By performing a detailed analysis of experimental results over a simplified version of our constraint model, we determine the most appropriate parameterization of the operational version of the constraint model. This version is now being deployed at ABB Robotics's CIR testing facilities and used on a permanent basis. This paper presents the empirical results obtained when automatically generating test sequences for CIRs at ABB Robotics. In a real industrial setting, the results show that our methodology is not only able to detect reintroduced known faults, but also to spot completely new faults.
Conclusion: Our empirical evaluation shows that constraint-based testing is appropriate for automatically generating test sequences for CIRs and can be faithfully deployed in an industrial context.

1. Introduction

A complex industrial robot (CIR) is defined as a classical industrial robot with an additional control system attached to perform a given process. This additional control system is typical responsible for controlling the process, which is typically painting, gluing, welding, and so forth.

Developing reliable software for CIRs is a complex task, because typical CIRs are comprised of numerous components, including control computers, microprocessors, field-programmable gate arrays, and sensor devices. These components usually interact through a range of different interconnection technologies, for example, Ethernet and dual port RAM, depending on delay and latency requirements on the communication. As the complexity of robot control systems continues to grow, the development and validation of software for CIRs is becoming increasingly difficult.

The problem is even worse for robots performing process-intensive tasks such as painting, gluing, or sealing, since their dedicated process control systems can be loosely coupled with the motion control system. In particular, a key feature of robotized painting is the ability to precisely activate the process equipment along a robot's programmed path. However, many of the processes involved in robotized painting are relatively slow compared to the process of moving the mechanical robot. Consequently, advanced computation-based techniques have been set up to take advantage of knowledge of the slower physical processes to

[*] Corresponding author at: University of Stavanger, 4036 Stavanger, Norway. Tel.: +47 514 89 247.
E-mail addresses: morten.mossige@uis.no (M. Mossige), arnaud@simula.no (A. Gotlieb), hein.meling@uis.no (H. Meling).
[1] Tel.: +47 406 26 077.
[2] Tel.: +47 518 32 080.

http://dx.doi.org/10.1016/j.infsof.2014.09.009
0950-5849/© 2014 Elsevier B.V. All rights reserved.

compensate for these latencies. Validation of such a paint control system, called an Integrated Painting System (IPS), is therefore challenging. Current testing practices to reduce the number of software faults apply techniques such as the manual design of unit and integration testing, where both the test inputs and expected output are defined by validation engineers. Testing the IPS requires access to the physical layer to activate many of the painting robot's features. Much of the testing is based on running the full-scale system with a moving robot and measuring IPS outputs with instruments such as an oscilloscope. This results in long round-trip times and little automation. In addition, many of the tests produced for one configuration of the IPS cannot easily be reused to test another configuration, since manual test configuration is required. These techniques are labor intensive and error prone. Consequently, software faults may still be detected late in the IPS design process, often close to release date, leading to increased validation costs.

In this paper, we report on a methodology to fully automate the testing of ABB's CIR control systems. The work builds on initial ideas sketched in a poster presentation [1]. Our approach draws on continuous integration principles and well-established constraint-based testing techniques. It is based on an original constraint-based model for automatically generating test sequences that are both *generated* and *executed* as part of a continuous integration process. By performing a detailed analysis of experimental results over a simplified version of our constraint model, we determine the most appropriate parameterization of the operational version of the constraint model. This version is now deployed at ABB Robotics's CIR testing facilities and used on a permanent basis. This paper presents the empirical results obtained when automatically generating test sequences for CIRs at ABB Robotics. In a real industrial setting, the results show that our methodology is not only able to detect reintroduced known faults, but also to spot completely new faults. Our empirical evaluation shows that constraint-based testing is appropriate to automatically generate test sequences for CIRs and can be faithfully deployed in an industrial context.

1.1. Contributions

The contributions of the paper can be summarized as follows:

1. Our testing methodology introduces a new constraint-based mathematical model focusing on IPS timing aspects. The constraints are used to describe both normal behaviors of the IPS, as well as abnormal behaviors, so that it is possible to target error states when generating test cases. The model is generic and expressed using simple mathematical notions, which makes it reusable in other contexts.
2. A full-scale implementation of the model is presented with constraint programming tools [2]. The paper presents how the model is integrated in a live industrial setting to test the IPS. To the best of our knowledge, this is the first time a constraint model and its solving processes are used in a continuous integration environment to test complex control systems.
3. An empirical evaluation is conducted to analyze the model's deployment. During this evaluation, reinserted old, historical faults are found by this new approach, as well as new faults. Comparing this constraint-based approach with current IPS testing practices reveals that the time from a source code change to the time that a relevant test is executed is dramatically reduced.

1.2. Organization

We start by providing background information and presenting related work in Section 2. In Section 3 we introduce robotized painting. We describe some of the design choices made when developing ABB's paint control system and how these affect testing of the system. We present how the IPS is currently tested in Section 4. We describe the paint control systems' mathematical properties in Section 5 and, based on these properties, we present the constraints used as a basis for generating a model that can be used for test case generation in Section 6. In Section 7, we describe how the model is implemented and how it is integrated with a continuous integration system. We then present the results this new test strategy in Section 8. We present a thoroughly experimental evaluation of the model recommendations of how to use the model. In Section 9, we suggest ideas for improvement and further work.

2. Background and related work

The methodology proposed in this paper is tightly coupled with continuous integration and model-based testing (MBT). This section recalls the basics of continuous integration and gives a brief overview of the most recent advances in the field by looking at how continuous integration influences verification and validation activities. This section also reviews usage of MBT, with a particular focus on constraint programming in software testing.

2.1. Continuous integration

Continuous integration [3] is a software engineering practice aimed at uncovering software errors at an early stage of software development, to avoid problems during integration testing. Even if there is no general consensus of what continuous integration is exactly, a typical continuous integration infrastructure includes source control repository tools, automated build, build servers,[3] and test servers. Fitzgerald and Stol [4] describe continuous integration as "a process which is typically automatically triggered and comprises inter-connected steps such as compiling code, running unit and acceptance tests, validating code coverage, checking compliance with coding standards, and building deployment packages." There is therefore a common understanding that the time from a continuous integration cycle being triggered to a developer receiving feedback should be as short as possible [5,6]. Therefore, one of the key ideas behind continuous integration is to build, integrate, and test the software as frequently as possible. Developers working under continuous integration are encouraged to submit small source code changes to the source code repository instead of waiting and occasionally submitting larger sets of changes.

If we consider test execution part of a continuous integration cycle, various testing activities could, in principle, be included. For example, automatic test case generation, test suite minimization, or prioritization [7–11] could be included to reduce the time needed to execute a test suite without reducing the quality of the overall test process. Interestingly, Hill et al. [12] report on the inclusion of system execution modeling tools to test distributed real-time systems as part of continuous integration. However, to the best our of knowledge, very few results evaluate the impact of including more testing activities in continuous integration. Our work, incorporating systematic automated test case generation methodology in continuous integration, is a first step toward more automation in the software validation of complex software control systems.

[3] A build server is a machine that fetches source code from the source control repository and performs building, testing, integration, and so forth. All steps are carried out completely automatically and typically triggered by a source code commit or a timer.

2.2. MBT and constraint programming

The test strategy described in this paper relates to different validation and verification approaches. The discussion work is divided into three topics.

Correct-by-construction approaches: When a step-by-step refinement process is used to derive an implementation, a correct-by-construction system can be obtained. Systems designed by such approaches are typically generated by a formal specification model in which the system's correctness is guaranteed and formally proved.

MBT: This approach typically involves three major stages: (1) A specification model (e.g., UML diagrams) is first built for testing purposes. (2) Then, the model is used to automatically generate test inputs and test oracles. (3) Finally, the actual system can be run with the generated inputs and its results compared with automatically predicted outputs.

Constraint-based testing: This approach aims to use constraint solving technologies to derive test cases automatically from a piece of code or a model. The main challenges for this approach lie in the mathematical formulation of the code or model and tuning the constraint solving process.

2.2.1. Correct-by-construction approaches

Correct-by-construction methods are frequently used in the design of safety–critical systems in avionics or railway domains, but other application domains are also relevant. Zhao et al. [13] reports on the use of discrete-event systems (DES) [14,15] for the design of an event-triggered real-time distributed system related to the "eye vision" project. In this approach, called Programming Temporally Integrated Distributed Embedded Systems (PTIDES), multiple cameras are synchronized via IEEE 1588 [16,17] to take synchronized images. Since each camera has its own internal timing characteristics, taking a synchronized image requires addressing problems that are similar to those encountered in robotized painting. This PTIDES approach is appealing, since formalizing the event-triggered real-time distributed system would drive engineers to automatically correctly implement it.

However, even if the problems addressed in PTIDES share some similarities with the testing of CIRs, a major drawback is that the complete system is required, including all functional behaviors, to model the problem. For many industrial applications, obtaining such a model is challenging. When some parts of the system are delivered by third-party suppliers, the problem is even worse.

Industrial robots are usually considered representative of the larger class of Cyber-Physical Systems (CPS) [18,19], whose modeling is known to be challenging [20]. Broy et al. [21] formally verify a distributed real-time system used in the automotive field, using a de facto modeling notation for developing automotive controllers, namely, Simulink/State. Using this formal notation enables automatic model-based code generation, analysis, and verification of the control software systems. This of course, is an advantage of the approach, but, again, a formal model is required for each component. Note also that pushing the system under test into error states is not easy when developing a correct-by-construction approach. Formal models tend to capture only correct behaviors, refining these only until code generation.

Generally, correct-by-construction methods requires skill in writing mathematical proofs, which is uncommon among average software developers. In our industrial environment, this method is clearly out of scope.

2.2.2. MBT

MBT [22] is a part of model-based design and is thus related to the previously mentioned approaches. A UML model can be developed to specify the architectural parts of the system, together with manual coding of the implementation details. Then, generating test cases based on the model allows the validation engineer to check the correctness of the developed code. However, according to Utting and Legeard [22], a more common approach in MBT is to create a dedicated executable testing model. This approach is simpler because the complete behavior of the system does not need to be reflected by the model and details unrelated to actual testing can be ignored. However, writing a UML executable model is more demanding than writing a constraint model focused on particular aspects of the system, such as timing aspects. Support tools for MBT are also limited when it comes to including actual testing into a continuous integration environment. Another challenging aspect concerns including the test generation process into MBT tools [23].

Specifying variable ordering when generating test inputs is usually not possible, meaning that the control of the test generation time is limited. We later show that this is a critical factor in finding solutions in a reasonably allocated contract of time.

2.2.3. Constraint-based testing

Use of constraint programming for automatic test case generation has been around for a long time e.g. Gotlieb et al. [24], Marre and Blanc [25], Di Alesio et [26]. Gotlieb et al. [27] developed a constraint programming model for automatic test case generation for C programs. Similarly, Marre and Blanc developed GATeL [25], a constraint-based testing tool able to generate test cases for synchronous languages. In both these approaches, Prolog with constraints was used, along with techniques to fine-tune the search process. More recently, Di Alesio et al. [28] adopted a similar approach to stress-test real-time applications. The approach proposed in this paper differs in that none of these constraint models are included with a continuous integration process and none of the constraint solving processes are launched at testing time. Such integration requires that the constraint solving time be carefully controlled.

3. ABB's process control system

This section first briefly introduces ABB's IPS before presenting a general introduction to robotized painting and some of the challenges involved in controlling slow physical processes. We discuss some of the trade-offs in testing the IPS. We also look at some of the design choices taken when developing the IPS and how we can view the IPS in a more abstract way.

ABB's IPS is a standalone distributed control system usually used with a standard ABB robot controller, but it can also be used with non-ABB robots. The IPS is a collection of different real-time embedded controllers capable of performing one or more process-related tasks. Examples of such tasks can be the closed-loop control of air flow/air pressure, the closed-loop control of pump pressure in paint flow, the closed-loop control of high voltage for electrostatic charging, and various control systems for operation on valves and the supervision of sensors. The IPS can be used in many different configurations, ranging from a single controller for small paint robots to large systems with more than 20 controllers interconnected over an industrial-grade network.

In the following, we illustrate the principles of robotized painting with the IPS with a small example and introduce some of the challenges.

3.1. Example of robotized painting

In this example, the objective is to apply paint to an object, using a robot. The robot is shown in Fig. 1 and the fill area at the bottom left illustrates object to be painted. We assume that the robot is programmed to move in a straight line at a constant speed of 1000 mm/s. We also assume time starts at $t = 0$, when the robot motion starts.

The spray pattern to be applied starts at 500 mm and, since the robot is moving at a speed of 1000 mm/s, the final spray pattern should be ‘on’ 500 ms after the start, as shown in Fig. 1. Producing the desired spray pattern involves at least four different physical processes that must be combined to obtain the expected pattern. For the purpose of this example, we consider four physical processes: a motor running a paint pump, a valve connected to the spray head through which paint flows when the valve is open, and two different air flows that are used to shape the paint fog that comes out of the spray head.

To account for the motion of the robot, the different physical processes must be activated at the appropriate times. For instance, about 200 ms before the robot arrives at the point where the paint should be applied, the robot controller may send the following message to the IPS: $(B = 1, t_a = 500)$. This message means that the IPS should apply spray pattern number 1 at activation time $t_a = 500$ ms. The value $B = 1$ is simply a logical value describing a specific spray pattern. The IPS uses the value of B as an index in an internal lookup table that provides the physical value to be applied to the actuator outputs to produce the desired spray pattern. For this particular example, $B = 1$ *could* mean that the actuators controlling the pump and air flows 1 and 2 should provide 400 ml/m of paint and 250 Nl/m and 400 Nl/m of air, respectively. These parameters are, of course, user configurable.

The IPS will then calculate *when* each of the actuator outputs needs to be activated to produce the requested spray pattern at t_a. Since many of the physical processes involved in painting have significant physical delays, their actual activation must take place *before* t_a. For this example, the IPS calculates that the pump must be started 50 ms before t_a, while the valve must be opened 80 ms before t_a. For the two air flows, activation must take place 120 ms and 150 ms before t_a, respectively.

As is apparent from this example, the IPS needs to synchronize several actuator outputs, where each output has its own timing characteristic and may be located on different controllers. The timing characteristics for a specific actuator output depend on many factors, the most important of which is the magnitude of the change in output. Consider, for example, a pump; a large change will take longer to apply than a small change, due to the acceleration of the motor.

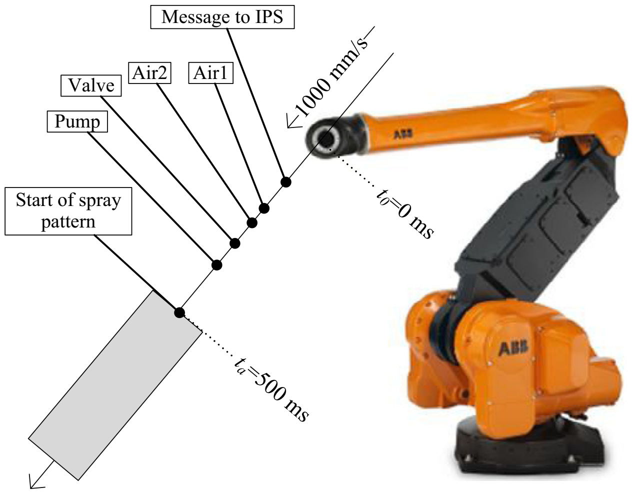


Fig. 1. To achieve the correct spray pattern at the starting edge of the object to be painted, different physical processes need to be activated with individual timing before the robot reaches the object.

3.2. Testing challenges

Offering a product with high levels of precision introduces several challenges in the development phase, among them being testing the system’s behavior with respect to its timing characteristics [29]. Testing the timing behavior of a centralized control system with a single clock can be challenging. However, the IPS is typically configured with a number of embedded controllers distributed across the robot system. These controllers run time synchronization protocols to keep their clocks synchronized. Still, testing the IPS timing behavior has proven to be a major challenge, mainly due to its distributed nature. Moreover, the degrees of freedom in configuring the IPS leads to further complexity in the testing phase, since a wide range of configurations must be tested. A natural consequence of these complexities is that automated testing has become a necessity.

The IPS is designed to be a highly flexible and configurable paint control system. Depending on the complexity of a customer’s solution, a robot is equipped with one or more embedded controllers running the IPS software.

The most complex configurations involve as many as 20 embedded controllers interconnected through an industrial-grade communication network. The main motivation for designing the IPS as a distributed system is to enable the different embedded controllers to be located physically close to the actual process that it controls. This enables fast control loops and is essential to make the system precise and accurate. The result of this design principle is that some of the controllers are placed at different locations on the robot, while others are located in a control cabinet close to the robot brain.

This design principle provides a powerful process solution but, complicates testing both due to the distributed nature of the IPS, and due to the fact that some of the embedded controllers can be located on movable and possibly hazardous robots.

3.3. Abstraction of the IPS

An abstract model of the IPS is shown in Fig. 2. As we can see, the robot controller communicates with an embedded controller, denoted the IPS master. This master connects to other embedded controllers through an industrial-grade network. Note also that all the embedded controllers are synchronized with respect to time. Since the robot controller and the IPS are synchronized, a function call to `gettimeofday()` executed on any embedded controller and the robot controller at the same time will return a synchronized time with microsecond precision. This accurate synchronization is one of the most important building blocks in the design of the IPS, since each embedded controller can schedule activation times for an actuator output, using the global clock.

4. Legacy test practices

In this section, we review some IPS testing practices, focusing on validating the accuracy of the time-based activation of actuator outputs. We discuss the benefits and drawbacks of these legacy testing practices before we outline the requirements for our automated test method.

A major challenge in testing a robot system is that it involves a physically moving part (the robot arm) that must be accurately synchronized with several external process systems. This quickly turns into labor-intensive procedures to set up and execute tests. Moreover, strict regulations with regard to safety must also be followed due to moving machinery and the use of hazardous fluids, such as paint [30].

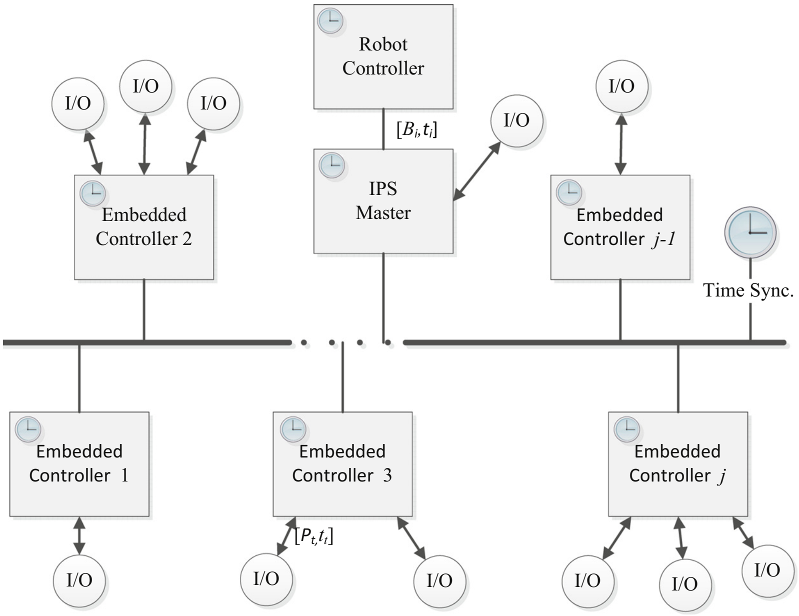


Fig. 2. Logical overview of the IPS. The IPS is interconnected by an industrial-grade network. All the embedded boards are synchronized by use of IEEE 1588 [16]. Each embedded controller is typically located inside the robot's control cabinet, at different locations on the robot arm, or in an external process control cabinet.

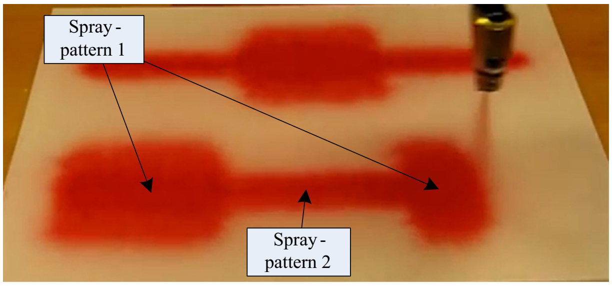


Fig. 3. Painting on paper allows for the visual inspection of the timing of different actuator outputs. However, the inspection must typically be performed by a paint process engineer in cooperation with a software engineer.

4.1. Painting on paper

To simulate a realistic application of spray painting with a robot, we can configure a paint system to spray paint on a piece of paper. An example of this is shown in Fig. 3.[4] This test includes both realistic robot movement and a complete and realistic IPS configuration. However, there are many drawbacks with this method. For instance, it involves quite a bit of costly manual labor to set up the test. In addition, the test can only be performed in a protected environment to prevent human exposure to dangerous paint fluids and gases. Finally, it is more or less impossible to automate this test, even after some initial configuration, as discussed below.

Due to its high cost, this type of test is typically performed during the final verification stage for a new product running the IPS software, such as a new air controller or a new pump controller. The test is also performed after a major refactoring of the IPS. Based on our experience at ABB Robotics, it is both extremely rare and difficult to find timing-related errors using this test method.

4.2. Activation testing with an oscilloscope

By reducing the IPS configuration to a single digital actuator output, without any fluid or air units and detecting trigger points using a proximity sensor, it is possible to run rudimentary synchronization tests on the IPS. Specifically, the test involves connecting the actuator output to an input channel on an oscilloscope and connecting the proximity sensor to another input channel on the oscilloscope. With this setup, the robot can be programmed to perform a linear movement passing over the proximity sensor, with the paint program set to activate at exactly that point. The robot thus generates a signal on its actuator output that should correspond exactly to the signal from the proximity sensor. By comparing the signal from the actuator output with the signal from the proximity sensor, it is possible to test many of the timing behaviors of the IPS.[5]

At ABB Robotics, this is one of the most executed tests aimed at uncovering synchronization problems, but it also requires manual labor to set up and execute the test runs. In addition, since it involves physical movement of the robot arm, a hazard zone must be established for the test. However, unlike the test described in Section 4.1, it can be executed without supervision and the test results can be inspected after test completion.

4.3. Running in a simulated environment

The IPS is designed to be portable to many microprocessor architectures and operating systems. It is even possible to run the IPS on a desktop system such as Windows. This advantageously allows much of the functional testing to be performed in a simulated environment, which reduces some of the need for time-consuming manual testing on actual hardware. However, testing against performance requirements is impossible in a simulated environment, due to the lack of real-time behavior in the simulator.

4.4. Summary of existing test methods

The test methods described above have several drawbacks. Test methods that use a real robot have the advantage of very realistic

[4] The video at http://youtu.be/oq524vuO5N8 also shows painting on paper.

[5] These two videos show activation testing using a proximity sensor and an oscilloscope, respectively: http://youtu.be/I1Ce37_SUwc and http://youtu.be/LgxXd_DN2Kg.

results, but they require slow, costly manual labor to set up the test and interpret the results. For the method described in Section 4.3, it is clearly possible to automate the setup and to some degree the result analysis. However, the method cannot be used to execute tests related to real time or synchronization between several embedded controllers. To cope with such tests, we need a new test method.

4.5. New test method

In the following, we outline the requirements for our new test method. The goals of the new method are automation, the reduction of manual labor, and reduction of the time required to detect errors introduced during development.

Automated: It should be possible to set up the test, execute the test, and analyze the results without human intervention.
Systematic: Tests should be generated automatically by a model rather than constructed by a test engineer.
Adaptive: Generated tests should automatically adapt to changes in the software and/or configurations and should not require any manual updates to the testing framework. This implies that tests should be generated immediately prior to their execution, using as input information obtained from the system under test.

5. Modeling the IPS

In this section we introduce a mathematical representation of the IPS. We first establish the mathematical relations within the IPS and show how these can be abstracted into a general-purpose model. We then show how the IPS can predict *when* to apply a change on an actuator output based on the activation time and the magnitude of the change. Finally, we discuss some of the interesting constraints and scenarios the IPS must be able to handle and show how they can be formulated as mathematical constraints and integrated into the model.

5.1. IPS channels

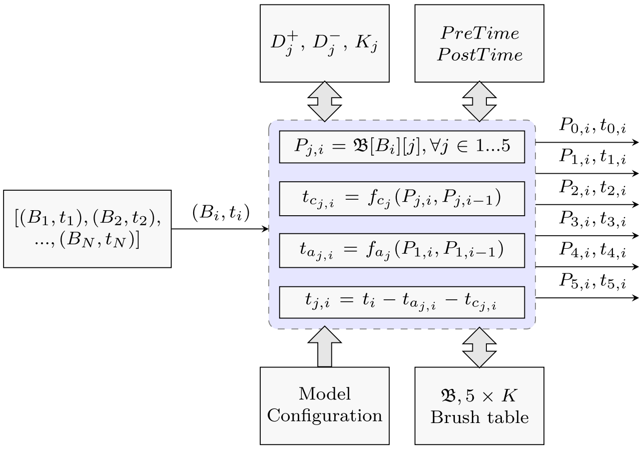
Before we introduce the mathematical model of the IPS, we need to introduce the concept of a *channel* used in the IPS.

As previously mentioned, the IPS can be configured in different ways, depending on the complexity of the process. One way to configure the IPS is by using channels. A channel is simply an abstraction that represents how a specific spray pattern is generated. Each channel is responsible for controlling *one* physical process, for example, air or paint, involved in generating a spray pattern. The current IPS supports up to five channels plus a special internal channel (channel 0) that is reserved for controlling the paint valve in the spray applicator. In the abstract model of the IPS shown in Fig. 4, each channel is shown as an output of the model.

5.2. Mathematical model

Abstractly, the IPS can be modeled as shown in Fig. 4. The input to the IPS is represented by a sequence of spray patterns along with their desired application times, that is, a sequence of (B_i, t_i)-tuples, denoting the ith spray pattern B_i and its application time t_i. This sequence corresponds to the commands sent by the robot controller. The output of the model represents the physical values for each channel j, along with their activation times, $(P_{j,i}, t_{j,i})$. In the following, we describe the mathematical relations for the transformation $(B_i, t_i) \mapsto (P_{j,i}, t_{j,i})$.

To model the physical processes they represent, each channel has its own set of configuration parameters, which are used as input to the timing calculation for the channel: D_j^+, D_j^-, and K_j in Fig. 4 and explained further in Section 5.6. The IPS can also compensate for timing disturbances between the different channels. This functionality is controlled by the parameters *PreTime* and *PostTime*. Finally, we have a *brush table* $\mathfrak{B}$ that is consulted to perform the transformation $B_i \mapsto P_{j,i}$.



D_j^+, D_j^-, K_j
PreTime
PostTime
$[(B_1, t_1), (B_2, t_2), ..., (B_N, t_N)]$
(B_i, t_i)
$P_{j,i} = \mathfrak{B}[B_i][j], \forall j \in 1...5$
$t_{c_{j,i}} = f_{c_j}(P_{j,i}, P_{j,i-1})$
$t_{a_{j,i}} = f_{a_j}(P_{1,i}, P_{1,i-1})$
$t_{j,i} = t_i - t_{a_{j,i}} - t_{c_{j,i}}$
$P_{0,i}, t_{0,i}$
$P_{1,i}, t_{1,i}$
$P_{2,i}, t_{2,i}$
$P_{3,i}, t_{3,i}$
$P_{4,i}, t_{4,i}$
$P_{5,i}, t_{5,i}$
Model Configuration
$\mathfrak{B}, 5 \times K$ Brush table

Fig. 4. Abstract mathematical overview of the test model.

All of the parameters mentioned above are treated as constants in a production installation. However, for the purpose of generating test sequences for the IPS model, these parameters are turned into variables that may change. Finally, the model configuration part of Fig. 4 contains configuration parameters describing how to generate the IPS test model. These parameters typically include the length of the test sequence and the type of test scenario.

5.3. Brush table

As mentioned earlier, the robot controller will send a new activation message with the value B_i, identifying a specific spray pattern. Internally in the IPS, B_i is used as an index in the brush table. The content of the brush table determines the actuator output for each channel, which is used to produce the desired spray pattern. This lookup function is expressed as follows:

$$P_{j,i} = \mathfrak{B}[B_i][j], \quad \forall j \in 1 \dots 5 \tag{1}$$

where $\mathfrak{B}$ is a brush table with five columns, one for each channel, and $|\mathfrak{B}|$ rows, representing the different spray patterns. For the internal channel 0, the output is derived from the value of channel 1, according to Eq. (2).

$$\begin{aligned} P_{0,i} &= 1 \quad \text{if } P_{1,i} > 0 \\ P_{0,i} &= 0 \quad \text{if } P_{1,i} \leq 0 \end{aligned} \tag{2}$$

This means that the valve controlled by channel 0 will open if channel 1 has a positive output. Moreover, a negative value on channel 1 corresponds to a special configuration for loading paint into a canister, meaning that the valve of channel 0 should be closed. Thus, it is important that channels 0 and 1 are tightly synchronized to prevent excess pressure on the hoses that carry paint, which could otherwise cause them to rapture.

5.4. Channel activation time

We now explain how to compute the activation times for each channel, $t_{j,i}$, from the desired spray pattern activation time, t_i, received from the robot controller. Eq. (3) shows how this calculation is performed:

$$\begin{aligned} &\forall j \in 0 \dots 5, \quad \forall i \in 1 \dots N \\ &t_{j,i} = t_i - t_{a_{j,i}} - t_{c_{j,i}} \\ &\quad = t_i - f_{a_j}(P_{j,i}, P_{j,i-1}) - f_{c_j}(P_{j,i}, P_{j,i-1}) \end{aligned} \tag{3}$$

where N is the size of the input sequence. The air delay $t_{a_{j,i}}$ and channel delay $t_{c_{j,i}}$ used in this equation are computed using Eqs. (4) and (5), respectively.

Note that the resulting time $t_{j,i}$ depends on the change between the actuator output $P_{j,i}$ and the previous output $P_{j,i-1}$. As we discuss later, each channel also has its own set of parameters that are used in this calculation.

5.5. Timing influence between channels

As mentioned earlier, some of the IPS channels will influence the timing of other channels. For example, turning on or off the paint channel (channel 1) will disturb the timing of the air channels (channels 2–4). To compensate for this disturbance, an *air compensation function* f_a is added to the air channels:

$$f_{a_j}(u, v) = \begin{cases} \mathit{PreTime} & \text{if } u = 0 \wedge v \neq 0 \\ \mathit{PostTime} & \text{if } u \neq 0 \wedge v = 0 \\ 0 & \text{otherwise} \end{cases} \quad \forall j \in 2 \dots 4 \tag{4}$$

where *PreTime* and *PostTime* are considered constant configuration parameters (see also Table A.4 in Appendix A).

5.6. Timing on isolated channels

Each channel has its own set of parameters that can used to adjust its timing characteristics. This timing is calculated using the *channel compensation function* f_c, shown in Eq. (5). A channel can be configured to have either a fixed delay or a delay that is linearly related to the change of $P_{j,i}$. A fixed delay is typically used for digital outputs that control valves, while a linear delay is typically used for outputs that control motors and air flows. For a linear delay, the time needed to adjust the output value depends on the magnitude of the change; a large change takes longer:

$$f_{c_j}(u, v) = \begin{cases} D_j^- \cdot \left(\frac{v-u}{Max_j - Min_j}\right)^{K_j} & \text{if } u < v \\ D_j^+ \cdot \left(\frac{u-v}{Max_j - Min_j}\right)^{K_j} & \text{if } u > v \\ 0 & \text{otherwise} \end{cases} \quad \forall j \in 0 \dots 5 \tag{5}$$

where $K_j \in \{0, 1\}$ is used to enable or disable the linear delay component. The terms D_j^+ and D_j^- are considered constant configuration parameters (see also Table A.4).

6. Test scenarios with constraints

With our mathematical model at hand, we now describe scenarios that can arise when multiple spray patterns are activated in succession. Accordingly, we identify mathematical constraints that can be used to generate test sequences to produce such error scenarios.

We divide the scenarios into two main categories. The first category expresses how the IPS behaves in a normal operational state. The second category represents scenarios in which the IPS is pushed into either an erroneous state or a state with reduced performance. These scenarios are summarized in Fig. 5 and discussed in detail in the following sections.

6.1. Normal scenario

During normal, non-erroneous behavior, the robot controller sends commands to the IPS and the IPS activates outputs according to the following constraints, respectively, both corresponding to Fig. 5a:

$$\begin{aligned} &\forall i \in 1 \dots N, \\ &t_i - t_{i-1} \geqslant \mathit{MinBrushSep}, \\ &t_i > t_{i-1}, \quad t_i \geqslant 0, \\ &B_i \neq B_{i-1}, \quad B_i \in 0 \dots |\mathfrak{B}| \end{aligned} \tag{6}$$

$$\begin{aligned} &\forall j \in 0 \dots 5, \quad \forall i \in 1 \dots N \\ &t_{j,i} - t_{j,i-1} \geqslant \mathit{MinTrigSep}, \\ &t_{j,i} > t_{j,i-1}, \quad t_{j,i} \geqslant 0 \end{aligned} \tag{7}$$

where *MinBrushSep* and *MinTrigSep* refer to two configurable parameters that are entered into the model prior to generating a test sequence. These constraints are especially efficient in generating test sequences with a corresponding configuration and oracle to validate that the IPS is behaving as expected under non-erroneous conditions. During comparison between the outputs generated by the IPS and the oracle generated by this scenario, we specifically look for missing output events and missing brush events.

6.1.1. Burst

An extension of the normal behavior scenario can be achieved by constraining the time span on either a set messages in the input sequence or a set of output activations. This makes it possible to force a *burst* of messages or activations within a limited time period. The constraints for a burst on an input sequence and a burst for an output channel are formalized, respectively, as

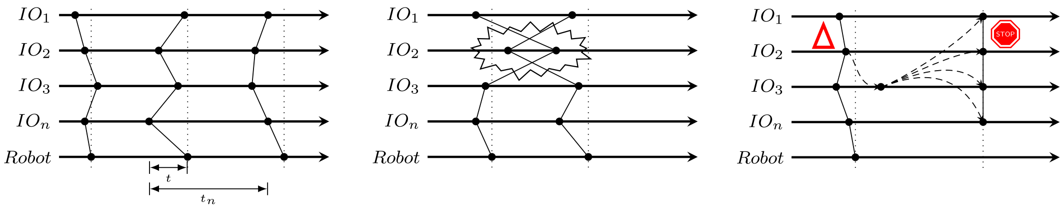
$$t_e - t_{e+\mathit{BurstLen}} \leqslant \mathit{BurstTime} \tag{8}$$

$$t_{c,e+\mathit{BurstLen}} - t_{c,e} \leqslant \mathit{BurstTime} \tag{9}$$

Both *BurstLen* and *BurstTime* are configurable input parameters in the model (see Table A.4).

6.2. Overlap scenario

Overlapping events are probably one of the most interesting scenarios that can be generated, as shown in Fig. 5b. This scenario is best explained with a simple example. Assume that one actuator



(a) Normal scenario. (b) Overlap scenario. (c) Shutdown scenario.

Fig. 5. A collection of error scenarios that the model can generate. Horizontal lines represent time and a black dot represents the activation of an output. A specific *spray pattern* is a collection of output activations, visualized by a line connecting the black dots.

output is configured with $K = 0$, $D^+ = 10$, and $D^- = -10$. Consider two events, where the first resulted in an activation schedule $P_{t1} = 0$ and $t_{t1} = 10$ for the actuator output IO_2. The second message is $P_{i2} = 1$ and $t_{i2} = 15$. Assuming that the current time (`gettimeofday()`) is less than 10, it is easy to see that $t_{t2} = t_{i2} - D^+ = 15 - 10 = 5$. As this example illustrates, an event received *later* can result in an activation time *before* events already scheduled for activation.

The IPS could generally handle such an overlap scenario in one of two ways. One possibility is to schedule the new event before the current event, resulting in the activation sequence $((P_{t2} = 1,\ t_{t2} = 5),\ (P_{t1} = 0,\ t_{t1} = 10))$. However, this approach has a serious safety flaw. Assume that the last event was some form of shutdown command, for example, to open a valve due to overpressure. Then the supervisor system would observe the actuator in an unexpected state.

Another option is to retain the old t_t and just replace the P_t value in the queue with the newly calculated P_t, resulting in a schedule $((P_{t2} = 1,\ t_{t1} = 10))$. We thus ensure that the actuator ends up in a state expected by our supervisor system. This corresponds to the approach taken by the IPS.

In real robot applications, there are many sources for this particular overlap scenario, the most common being that a customer wishes to increase the speed of the robot and thus moves the activation time of two events closer together. The standard behavior for the IPS is to report this in an error message to the user and resolve the schedule as described above:

$$\begin{aligned} &t_{c,e} - t_{c,e+1} \geqslant MinOverlapTime, \\ &t_{c,e+1} - t_{c,e-1} \geqslant MinTrigSep, \\ &t_{c,e+2} - t_{c,e} \geqslant MinOverlapTime \end{aligned} \tag{10}$$

where $t_{c,e}$ represents the activation time for a specific channel c and event e. Note that *MinOverlapTime* and *MinTrigSep* are considered positive constants given as input when a test sequence is generated (see also Table A.4).

6.3. Shutdown scenario

The shutdown scenario is important to validate that the IPS is able to shut down safely in specific error cases. Depending on the IPS's configuration, forcing one of the output channels to fail may cause the IPS to initiate a controlled shutdown. This shutdown procedure must be performed in a special sequence, taking care to avoid pressure buildup in hoses, which could otherwise lead to rupturing them. This scenario is illustrated in Fig. 5c and its constraint is specified as

$$P_{c,e} = IllegalVal \tag{11}$$

where *IllegalVal* is a configurable input parameter in the model (see Table A.4) that causes the IPS to initiate a shutdown.

6.4. Minimizing test execution time

As stated previously, the actual test sequence sent to the IPS is a sequence of timed events $(B_1, t_1, \ldots, B_N, t_N)$. When the test sequence is executed, each (B_i, t_i) pair is sent to the IPS at time t_S, such that $t_S + t_\delta \leqslant t_i$. This means that the IPS receives each pair (B_i, t_i) around t_δ before the activation time. In practice, the value of t_δ is typically around 200 ms. Consequently, the execution time of a complete test cycle lies in the area of the time of the last t_i, that is, t_N. By minimizing the value of t_N, we gain the ability of executing more tests within a given time interval as we discuss in Section 8.4.

7. Implementation

This section explains how the model is implemented, deployed, and used in ABB's production-grade test facility. We also discuss some of the design choices made during the model's deployment.

7.1. Test setup

This section describes the steps involved in setting up a continuous integration-based test facility for generating and executing tests. Test execution is typically triggered by a build server upon a successful build of the IPS software. These steps are illustrated in Fig. 6 and explained below.

1. *Build:* The software is scheduled to be built every night. In addition, a developer can trigger a manual build or a build can be triggered by a check-in to the source control repository.
2. *Upgrade:* All embedded controllers are upgraded with the newly built software. This is one of the most important tests performed, one where catastrophic, hard to find errors are often be detected. Typically, these can cause the new software to throw an exception or simply freeze.
3. *Configure:* In this step, the IPS is configured according to configurations retrieved from the source control repository. This configuration can be either a specific qualified setup of one of the different configurations that a customer can buy or a configuration specially made for testing purposes.
4. *Query and solve model:* A set of basic *smoke tests* [22] is then executed before the constraint model is launched for test case generation. By feeding data retrieved from the new configuration into the constraint model, together with properties retrieved from the IPS, we ensure that the gener-

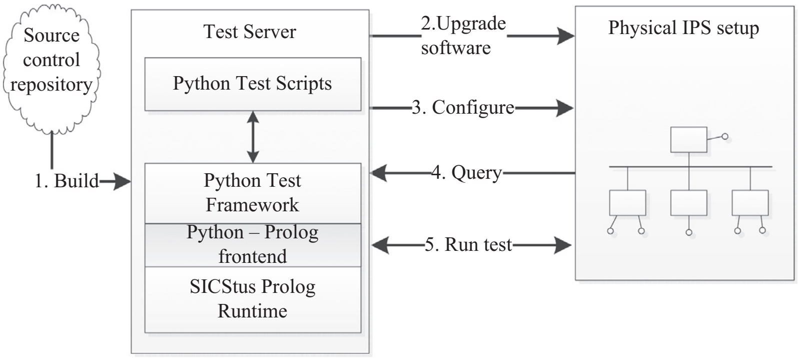
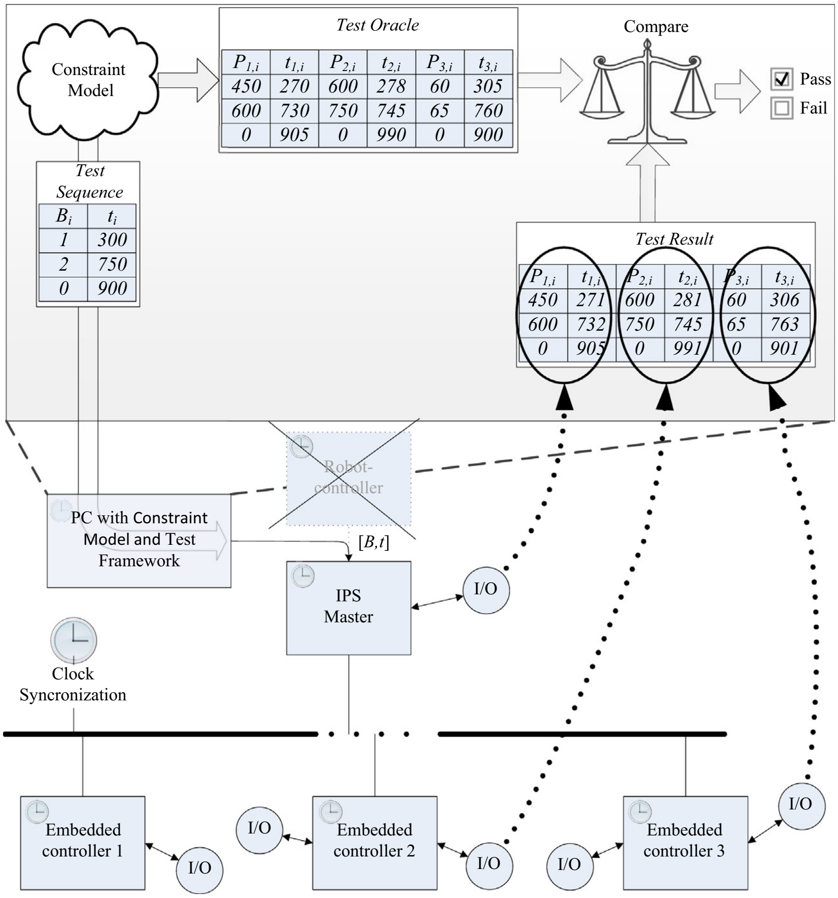


Fig. 6. Integration between the test server and IPS. The test server typically receives a new build from a build server, upgrades all embedded boards, performs tests, and publishes the results for the developer. The numbers correspond to the explanation given in Section 7.1.



Test Oracle

$P_{1,i}$	$t_{1,i}$	$P_{2,i}$	$t_{2,i}$	$P_{3,i}$	$t_{3,i}$
450	270	600	278	60	305
600	730	750	745	65	760
0	905	0	990	0	900

Test Sequence

B_i	t_i
1	300
2	750
0	900

Test Result

$P_{1,i}$	$t_{1,i}$	$P_{2,i}$	$t_{2,i}$	$P_{3,i}$	$t_{3,i}$
450	271	600	281	60	306
600	732	750	745	65	763
0	905	0	991	0	901

Fig. 7. How a complete test is executed. The constraint model generates the test sequence, the configuration of the IPS, and the oracle. The configuration is applied to the IPS and the test sequence is executed. The oracle is then compared with actual measurements before a pass/fail is determined. Currently 11 different variations of this setup are being executed in parallel at ABB Robotics.

ated tests are kept in sync with the current software and configuration. Further details about this just-in-time test generation (JITTG) are discussed in Section 7.2.

5. *Run test:* Finally, the actual test is executed by applying the generated test sequence and comparing the actuator outputs with the model generated *oracle*. Fig. 7 shows this last step in more detail.

In ABB's production test facility, each generated test sequence is executed on 11 different configurations, including execution on different hardware and software generations of the IPS and on both VxWorks and Linux as the base operating system for the IPS. The test framework is written in Python [31] and supports parallel test execution as long as resources are not shared. This allows for a significant reduction in the time needed to run the test sequence on many different configurations, compared to running them one at a time, in sequence.

7.2. JITTG

As discussed in Section 5, many parameters in the model must be specified before the model can be solved. Some of these parameters come from configuration files used to configure the IPS and some can be extracted by querying a newly built IPS. Common to both sets of parameters is that the resulting model will differ if the parameters change. This means that the model is tightly coupled to what is fetched from the source control repository. Consequently, we decide to generate and solve the model at testing time, as opposed to solving the model once and adding the resulting model to the source control repository, corresponding to what Utting et al. [32] call on-line and off-line testing, respectively. The choice of on-line versus off-line testing is a trade-off. The main advantage of JITTG is that there is a lower probability of falsely reporting an error due to a mismatch between the generated model and the real system. However, an important concern then becomes the time needed to solve the model. If the model is solved once and used many times, a solving time of several hours is reasonable. However, with JITTG the solving time becomes crucial. The models solved so far have a solving time of less than a few minutes.

7.3. Model implementation

To convert our mathematical model into an executable model out of which test sequences and test oracles could be extracted, we use *Constraint Programming* (CP) [2].

Constraint programming is a well-known paradigm introduced 25 years ago to solve combinatorial problems in an efficient and flexible way [33]. Typically, a constraint programming model is composed of a set of variables *V*, a set of domains *D*, and a set of constraints *C* and constraint resolution aims to find *solutions*, that is, assignments of *V* to values that belong to *D* such that all the constraints *C* are satisfied. Finding solutions is the purpose of the underlying constraint solver, which applies several *filtering techniques* [33] to prune the *search space* formed by all the possible combinations of values in *D*. A nice feature of constraint programming is the ability to call the constraint solver incrementally, during program execution. Consequently, most constraint programming solvers are embedded into various programming languages, including Java, C++, and Prolog, or dedicated modeling languages, such as OPL, Comet, and Zinc [34].

In practice, constraint models developed to solve concrete and realistic combinatorial problems usually contain complex control conditions (e.g., conditionals, disjunctions, recursions) and integrate optimized and programmable search procedures. The flexibility and versatility of constraint programming are recognized as a competitive advantage over other, more rigid approaches [2].

However, solving the mathematical model could have been possible by using other techniques, such as SAT or SMT solving [35], search-based test data generation [36], or Mixed Integer Programming (MIP) [37]. These techniques were examined and discarded for the following reasons:

1. The selected technique had to be flexible enough to accommodate the many alternatives in the dynamic configuration of the IPS. MIP techniques are very powerful for handling conjunctions of linear constraints [37], but handling disjunctive constraints (i.e., non-linear constraints) is much more problematic. Constraint programming offers a high degree of flexibility to handle disjunctive constraint systems, including the use of backtracking, *reification*, or *constructive disjunction* [34].
2. Time-constrained optimization is essential to use the technique in an industrial context and to build a cost-effective testing method. SAT and SMT solving are amazingly efficient at handling Boolean and non-Boolean constraint satisfiability problems [35], but they are not tuned to solve optimization problems (e.g., minimizing a cost function in a given contract of time). Even if extensions exist to handle constraint optimization problems (e.g., Max-SAT), usual SAT- or SMT-solvers do not necessarily implement these extensions. On the contrary, constraint programming integrates time-aware optimization methods on discrete combinatorial problems in its foundations, which makes it more flexible to tackle optimization problems within an industrial process [34].
3. Since the model is used to predict the expected outputs of the IPS processing of a timed-event sequence, exact methods are mandatory. Despite the efficiency of search-based test data generation techniques [36], the absence of a guarantee of the satisfiability of the constraints (e.g., no possible detection of unsatisfiability or no guarantee of the determination of satisfiability for complex constraint sets) was regarded by us as a sufficient reason to discard these techniques. On the contrary, constraint programming offers a theoretical guarantee on the assessment of satisfiability [33]. We should also mention that, since industrial adoption was set up as an essential goal, we felt that deterministic methods would be more appropriate than probabilistic approaches of constraint solving to convince engineers.

It is worth noticing that CP solvers are usually hosted by a programming language e.g., Prolog, Java or C++. Thus, they have to be flexible to facilitate their integration into applications, and incremental, i.e., constraints can be submitted at different stages of the parsing process. The constraint model can be structured by using high-level programming features such as predicate or method invocation, recursive and virtual calls, and backtracking or inheritance.

We implemented our mathematical model using the finite domain constraint solving library of SICStus Prolog, called `clpfd` [38]. This library is well maintained and up-to-date with respect to the last advances in constraint programming solving, which was a sufficient reason to select it for industrial adoption. The `clpfd` solver is fully hosted and integrated within the Prolog programming language and is called incrementally during Prolog program execution.[6] To integrate the model with ABB's existing test framework, we also built a front-end layer in Python. This front-end layer can be used by test engineers with no prior knowledge of constraint programming or Prolog and also allows us to integrate with our existing build and test servers based on Microsoft Team Foundation Server. A schematic overview of the architecture is shown in Fig. 6.

8. Empirical evaluation

The constraint model introduced in Section 5 has been thoroughly evaluated to validate its ability to generate test sequences for CIRs in a realistic industrial environment. Our objective was to quantify the benefits and drawbacks of introducing a new testing strategy in a continuous integration process, after having deployed it within ABB's testing facilities.

This section presents the main research questions (RQs) (Section 8.1) addressed so far in our empirical evaluation. It details the experimental results and their analysis (Section 8.2). It evaluates several threats to the validity of the results and discuss their importance (Section 8.6). Finally, this section concludes with an analysis of several lessons learnt when deploying this approach in an industrial environment (Section 8.8).

8.1. Research questions

The introduction of a new test strategy (i.e., a constraint-based model) into a strong validation process always raises many research questions regarding its adoption. Our empirical evaluation addressed three main research questions, covering the following.

RQ1 (efficiency of the search heuristics): Questioning the efficiency of the constraint model to generate test sequences is of primary importance. Among several parameters, the selection of search heuristics turned out to be a key factor of the strategy's efficiency. Observing that different search heuristics can lead to completely distinct results, we conducted a systematic comparative study of several representative search heuristics to respond to this research question.

RQ2 (model scalability): The scalability of the model to generate realistic test sequences is also a main question. To introduce the constraint model into a continuous integration environment, managing the model solving time was crucial. Evaluating this solving time for different settings appeared to be the best way to evaluate the model's scalability.

RQ3 (Adoption in an industrial environment): Finally, evaluating the capabilities of the model to find previously found bugs and also its ability to uncover new faults in an industrial, realistic validation process was also considered a crucial research question. In response to this research question, we determined that the only way was to put the constraint model to work for a period of time and evaluate its potential through a systematic analysis. Essentially, we saw this work as mandatory to prepare the model for industrial adoption on a larger scale.

8.2. Experimental setup

In response to the three research questions, we developed two different constraint models. The first model, denoted CM_1, is a highly configurable and general model that includes several

[6] We used a compiled version of the model.

measurement and analysis tools. The model CM_1 is mainly made for use from within the SICStus environment. The second model, denoted CM_2, is highly tuned and optimized for the industrial production environment. It is callable from an external Python framework and contains all the functions to generate realistic test sequences and test oracles. To answer RQ1, we configured one experiment that systematically analyzed all possible combinations of variable orderings for defining the search heuristics combined with different configurations of the model. The goal of this experiment was to identify a search heuristic that could be further tested on the CM_2 model. In the second experiment, we used the results from the first experiment on the CM_2 model to answer RQ2.

In the following, we give a detailed account of our observations and findings.

8.3. RQ1, Experiment 1

Our first experiment is divided into three sub-experiments, using three distinct configurations: {*SeqLen*, $|\mathfrak{B}|$, *Channels*, *MinTrigSep*, *MinBrushSep*} = {7, 3, 3, 3, 1} for Exp1a. For Exp1b and Exp1c we use respectively {10, 5, 5, 3, 1} and {20, 10, 5, 1, 1}. Since experiment Exp1b and Exp1c are just slight variations of Exp1a, we present only the final results for these, while for Exp1a we also present detailed setup and execution results.

Experiment Exp1a uses a minimal configuration with three channels, $j \in [1, 3]$, as illustrated in Fig. 8, yet is complex enough to provide significant and meaningful results. Each channel has the following characteristics: $Min_j = 0,\ Max_j = 3$, and $D_j^+,\ D_j^- \in [-3, 3]$. The brush table has $|\mathfrak{B}| = 3$ rows and, since there are three channels, $\mathfrak{B}$ becomes a 3×3 matrix, as shown in Fig. 8. At runtime, the model can freely choose a linear or a fixed delay for each channel j, using $K_j \in [0, 1]$ (see (5)). For all three channels, this adds up to the following sequence of variables that need to be *labeled* by the constraint solver:

$$C = \left(D_1^+, D_1^-, K_1, D_2^+, D_2^-, K_2, D_3^+, D_3^-, K_3\right)$$

In the context of a constraint solver, the term *labeling* denotes the process of selecting a value from the legal *domain* of a variable and assigning it to the variable such that all constraints are fulfilled.

Note that we use parentheses to denote ordered sequences and brackets to denote unordered sets. For a constraint solver, the order in which the variables are labeled is of crucial importance for efficiency.

The variables in $\mathfrak{B}$ take on the values in the range $[Min_j, Max_j]$. Finally, we specified that none of the channels should slave channel 1, as explained in Section 5.5; that is, we set $PreTime = 0$ and $PostTime = 0$. We also set $MinTrigSep = 3,\ MinBrushSep = 1$, and $MinOverlapTime = 1$. The expected input for Exp1a is a sequence of index-time pairs denoted $(B_1, t_1, \ldots, B_7, t_7)$. Each pair (B_i, t_i) is sent as an individual input to node **M** in Fig. 8. Fig. 9 shows an example of an *optimal* solution found by our method for Exp1a. In this case an *optimal* solution means that the constraint solver has found the lowest possible value for t_7 while still satisfying the constraints.

In Exp1a, the expected test sequence is of length $N = 7$ and the goal was to elicit an overlap on **C2** between events 5 and 6, as shown in Fig. 5b and described in Section 6.2. Thus, we engineered the experiment to elicit an overlap scenario. We chose this scenario because it is the most difficult to obtain.

Another goal with experiment 1 was to find the shortest test sequence able to elicit the error scenario, since minimizing the duration of the test sequence allows test engineers to run more tests. Consequently, our constraint model is used in combination with a time-aware cost optimization process, where the goal is to minimize t_N, the duration of the test sequence, in a given contract of time. We used a timeout value of 180 s of computation time for all three sub-experiments.

In this context, an *optimal* solution is an assignment of values to all the variables such that all constraints are satisfied and t_N is minimized. If sufficient time is allocated, the minimization process can provide an optimality certificate. In most cases, this certificate is not required and the process returns an optimal or sub-optimal solution without any certificate. For a solution without a certificate, there is no way to evaluate the distance to the true optimal value of the cost function. If insufficient time is allocated, the solver sometimes reports a failure, indicating that it has been unable to find a solution. These cases are obviously the most problematic ones.

As mentioned earlier, the order in which the variables and values are selected for labeling is a critical parameter for the efficiency of the constraint-solving process in `clpfd`. In this experiment, we defined search heuristics based on distinct choices of the variable and value selection.

8.3.1. Variable selection heuristics

Based on previous definitions, we propose various sequences with distinct variable orderings. We first consider the four possible orderings between B_i and t_i, denoted as follows:

$$\begin{aligned}
\overline{B_i t_i} &= (\ldots, B_i, t_i, B_{i+1}, t_{i+1}, \ldots)\\
\overline{t_i B_i} &= (\ldots, t_i, B_i, t_{i+1}, B_{i+1}, \ldots)\\
\overline{B_i} &= (B_1, \ldots, B_i, \ldots)\\
\overline{t_i} &= (t_1, \ldots, t_i, \ldots)
\end{aligned} \tag{12}$$

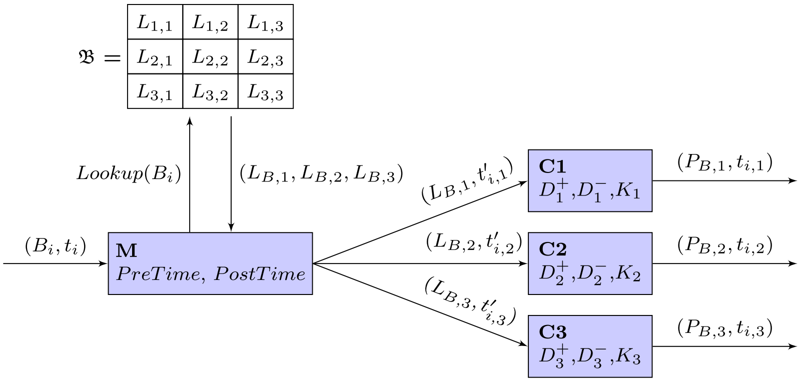
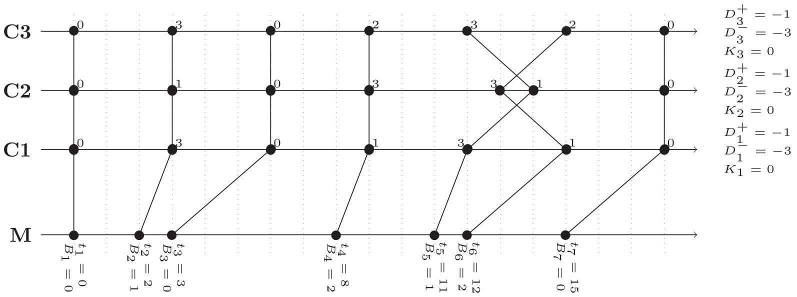


Fig. 8. Logical overview of experiment Exp1a. The experiment includes the use of three actuator outputs (channels), a test sequence of length 7 (*SeqLen*), and a lookup table size of size 3 ($|\mathfrak{B}|$).



$$\mathfrak{B} = \begin{matrix} 0 & 0 & 0 \\ 3 & 1 & 3 \\ 1 & 3 & 2 \end{matrix}$$

$MinBrushSep$	1
$MinTrigSep$	3
$MinOverlapTime$	1
$PreTime$	0
$PostTime$	0

Fig. 9. An *optimal* solution for Exp1a. The cost function used by the solver is *minimize*(t_7), where the optimal solution is $t_7 = 15$. The number next to each black dot (•) represents the value of the actuator output to apply at that time instance. The configuration of each actuator output is to the right of each actuator output's time axis ($D^+, D^-,$ and K).

We now define the following two sequences of variables from $\mathfrak{B}$ in vector form:

$$L = \text{vec}(\mathfrak{B}) = (L_{1,1}, L_{1,2}, L_{1,3}, L_{2,1}, L_{2,2}, L_{2,3}, L_{3,1}, L_{3,2}, L_{3,3}, \ldots)$$
$$L^T = \text{vec}(\mathfrak{B}^T) = (L_{1,1}, L_{2,1}, L_{3,1}, L_{1,2}, L_{2,2}, L_{3,2}, L_{1,3}, L_{2,3}, L_{3,3}, \ldots)$$

If we now combine all the sequences of variables and define all the combinations of sets such that each set contains exactly the same variables but the *sequences* that each set contains are different, we obtain

$$\begin{matrix} G_1 = \{C, L, \overline{B_i t_i}\} & G_3 = \{C, L, \overline{t_i B_i}\} & G_5 = \{C, L, \overline{t_i}, \overline{B_i}\} \\ G_2 = \{C, L^T, \overline{B_i t_i}\} & G_4 = \{C, L^T, \overline{t_i B_i}\} & G_6 = \{C, L^T, \overline{B_i}, \overline{t_i}\} \end{matrix} \quad (13)$$

where G_1, G_2, G_3 and G_4 are sets of cardinality 3, while G_5 and G_6 are of cardinality 4. Considering all possible combinations of these sets yields $4 \cdot 3! + 2 \cdot 4! = 72$ distinct possibilities. Note that all the resulting orderings are pairwise distinct. In our experiments Exp1a, Exp1b, and Exp1c, we systematically explored the results on these 72 distinct search heuristics.

8.3.2. Value selection heuristics

To find solutions with `clpfd`, each variable has to take on a value in its domain. Exhaustively exploring the domain can be realized through several strategies e.g., starting from the middle of the domain, picking a value at random from the domain. For the sake of simplicity, we only explored the following two simple strategies.

up: If $x \in [a, b]$, then explore the domain from the smallest value to the largest (i.e., $x = a,\ x = a + 1, \ldots, x = b$).
down: If $x \in [a, b]$, then explore the domain from the largest value to the smallest (i.e., $x = b,\ x = b - 1, \ldots, x = a$).

Other value selection heuristics were briefly explored without finding significant improvements, so we concluded that these two strategies were the most important to evaluate.

8.3.3. Result for Experiment 1

To classify the results on the 72 measurements, four different categories were defined, from the most useful to the least interesting:

Category 3 (sub-optimal) still represents interesting heuristics, since a solution is found but, since optimality is not reached, this category is less interesting than Category 2. Note that to distinguish between Categories 2 and 3, we have to know the optimal value of the cost function in advance. This is possible for the simple problems in experiment 1, but not in experiment 2 or whenever the model is used in production.

Fig. 10 shows a detailed depiction of all executions of Exp1a, where the four categories are represented. For Category 1, the graph shows the time needed to find an optimal solution, while for Categories 2–4, timeout is reached. These categories are grouped together and classified through a qualitative difference. These results are summarized in Table 1.

8.3.4. Analysis of Experiment 1

For Exp1a, we obtain in total 42 heuristics where no solution is found, six heuristics where the search finds an optimal solution without any proof certificate, and finally 24 heuristics where an optimal solution is found and proven to be optimal. Furthermore, from Fig. 10, among the 24 successful heuristics, the time needed to find and prove optimality ranges from 0.7 s to 149.4 s. Generally speaking, the two graphs in Fig. 10 show that the value selection heuristics **up** is more interesting than **down**. This results from the fact that when selecting first largest values for all the variables, longer sequences are privileged.

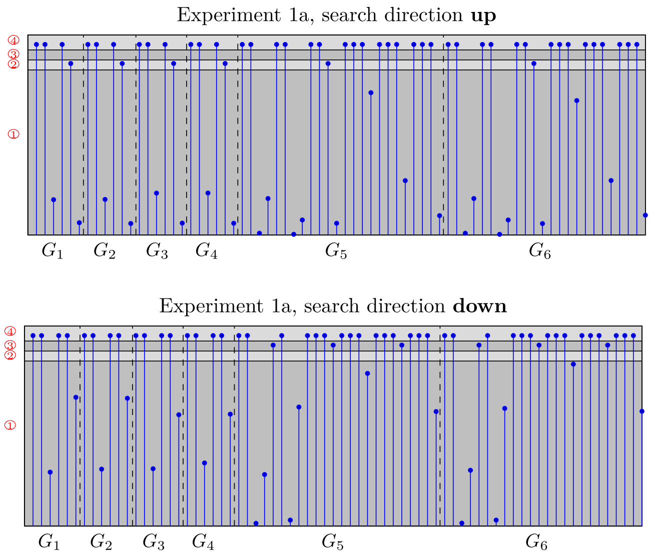


Fig. 10. A graphical presentation of the results for Exp1a from Table 1. The colored circles on the left axis correspond to the descriptions in Table 1. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 1
Summary of results for experiment 1, where we classify the search heuristics into four categories. The timeout was set to 180 s for all experiments. A more detailed graphical presentation of Exp1a is given in Fig. 10.

	Search direction					
	Up			Down		
	Exp1a	Exp1b	Exp1c	Exp1a	Exp1b	Exp1c
① Optimal	24	6	0	20	2	0
② Optimal, timeout	6	0	2	0	0	2
③ Sub-optimal	0	7	2	6	18	2
④ No solution	42	59	68	46	52	68
Total	72	72	72	72	72	72

①. *(Optimal):* An optimal solution is found and an optimality certificate is obtained within the contract of time, that is, optimality is proven.
②. *(Optimal, timeout):* An optimal solution is found but no certificate is provided, that is, optimality is not proven.
③. *(Sub-optimal):* A sub-optimal solution is found but the search timed out. This means that optimality is neither reached nor proven.
④. *(No solution):* No solution is found within the contract of time.

We found that only two variable selection search heuristics perform acceptably for the three sub-experiments, namely, $(L^T, \overline{B_i}, C, \overline{t_i})$ and $(L, \overline{B_i}, C, \overline{t_i})$, the first having a slight advantage. Even if these two heuristics do not always give the best results in terms of CPU time, they can both be used in combination with the **up** value selection heuristic for the three sub-experiments. This result can be explained by the fact that first giving the values for the brush table (i.e., L^T or L) drastically reduces the size of the search space by withdrawing numerous choice points originating from the table. The remaining sequence $(\overline{B_i}, C, \overline{t_i})$ is also important, but most probably for technical reasons involving the shape of the constraints. Selecting one of these two heuristics answers RQ1 by providing a solid foundation for the analysis of search heuristics in the context of constraint-based test sequence generation. Based on these results, we selected $(L^T, \overline{B_i}, C, \overline{t_i})$ for the second part of our empirical evaluation, dedicated to answering RQ2 and RQ3.

8.4. RQ2, Experiment 2

To answer RQ2, we examine how scalable the proposed model is with respect to the heuristics discovered in experiment 1. Scalability in the context of timed test sequence generation for CIRs can be understood as (1) determining the largest test sequences the model is able to generate within a reasonable time, (2) determining the impact of the brush table size on the time needed to generate a test sequence, and (3) determining the optimal contract of time to be allocated to the minimization process.

To answer these questions, we ran experiments with $|\mathfrak{B}| = (10, 15, 20)$ and $SeqLen = (50, 100, 150, 200, 250, 300)$, which yields 18 different configurations. Each configuration was systematically executed using all timeout values in the range $[2, 30]$ s, in addition to 60 s, 120 s, 180 s, and 600 s. For each timeout value, the ability to find a solution and the value of t_N that was found were reported.

8.4.1. Analysis of Experiment 2

Fig. 11 relates the test sequence duration, t_N, to the solving time, t_s, for 15 different configurations. Note that the model could not be solved for $|\mathfrak{B}| = 10$ in combination with large values of $SeqLen$ within the time contract of 600 s. For this reason, only three results are reported for $|\mathfrak{B}| = 10$, namely, those where $SeqLen \leqslant 150$. For $|\mathfrak{B}| = 15$ and $|\mathfrak{B}| = 20$, we got results for all combinations of $SeqLen$. Note also that all executions, except for $|\mathfrak{B}| = 10$, $SeqLen = 150$, provided a sub-optimal solution within 10 s. In fact, most of the executions generated a first solution in less than three seconds.

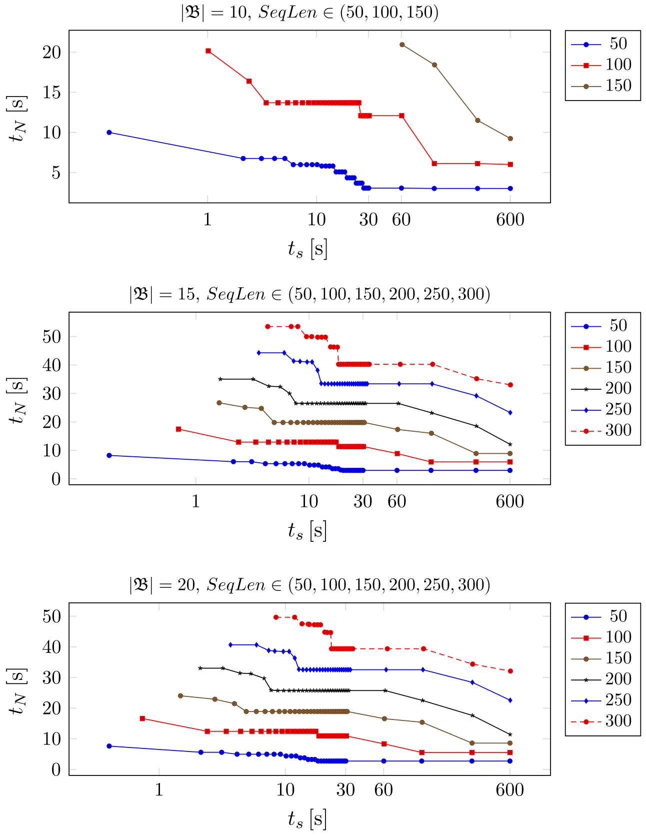


Fig. 11. How well the model minimizes the duration of the test sequence, t_N, if more time is added to the solving process, t_s.

This result is encouraging for our desired deployment in a continuous integration environment. On the one hand, a test sequence where t_N is minimized is highly desirable but, on the other hand, allocating a very long contract of time to reach this objective is counterproductive in continuous integration, since this will result in a reduction in the number of tests that can be executed. The tradeoff relation can be precisely computed and represented as follows, with a test efficiency factor E that tells us how much time can be spent in the solving phase to obtain as many changes in B_i as possible:

$$E = \frac{SeqLen}{t_N + t_s} \tag{14}$$

In Fig. 12, we plot the efficiency factor for all the tested configurations. As the plot shows, the maximum efficiency is obtained after two seconds to 12 s of solving time. Thus, if the model is generated and solved solely for *a single execution*, there is no benefit running the solver longer to obtain a better solution. As an example, consider the case with $|\mathfrak{B}| = 10$ and $SeqLen = 50$. For this case, the first value found is $t_N = 9.99$ s and, by running the model an additional 30 s, we obtain a solution that executes in only $t_N = 3.05$ s, that is, a 30% reduction from the first solution. Clearly, this is wasted effort if the solution is used only once. However, if the generated model and the solution is meant to serve multiple consecutive test runs, it may be advantageous to run the solver longer to further reduce t_N.

In conclusion, unless a test sequence can be reused multiple times, there is not much to gain from extending the solving phase.

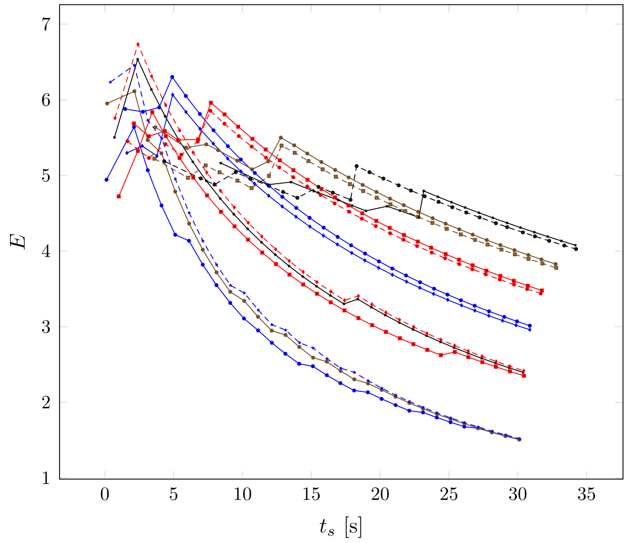


Fig. 12. Efficiency factor E for the executions in Fig. 11. When the model is run in a continuous integration environment, there is little to gain from running the model more than around 10 s.

8.5. RQ3, deployment, and industrial adoption

We now address our last research question, whether our proposed model can be implemented in a real industrial setting. We divide this question into three parts:

- Is the model able to detect new errors?
- Is the model able to detect old errors that were reintroduced into the IPS?
- Does the proposed JITTG framework behave as expected?

8.5.1. New errors detected

This section describes the errors found immediately after we introduced the new model. These are errors that were in the IPS for some time and were only detected by the new model. We found a total of three previously unknown errors in the IPS. Two of the errors were directly related to the behavior of the IPS, while the last was related to how a PC tool presents online diagnostics for a live system.

8.5.2. Detection of old errors

To further validate the robustness of the model, a collection of old, previously detected errors were reintroduced into the source code with the intention of verifying that the model was able to detect the errors. The selected errors were chosen by searching ABB's bug tracking system, by interviewing ABB's test engineers, and through discussions with the IPS's main architect. Most of the errors were originally discovered at customer sites while staging a production line or after the production line was set into production. The chosen errors are mainly related to timing errors of painting events and several of the errors can be classified as errors that appear when the IPS is part of a large configuration with many components.

The chosen errors are summarized in Table 2. This table shows historical data on how long it took to detect the error, how long it took to fix the error, and how long it took to validate that the error had in fact been fixed. Note that these numbers cannot be accurately specified; they represent reasonable estimates. In particular, errors related to *how long* a bug has been in the system are difficult to estimate. However, by interviewing the main architect of the IPS and the lead test engineer, we have high confidence in the numbers presented.

Table 2
Historical data on old bugs that were reintroduced to test the model.

Bug#[a]	Time in system[b]	Time to solve[c]	Time to validate[d]
44432	5–10 years	1–2 h	1 day
44835	5–10 years	2–4 days	1 day
27675	6–12 months	1–2 months	1–2 weeks
28859	6–12 months	2–3 months	2–3 weeks
28638	4–6 months	1–2 weeks	2–3 weeks

[a] The bug number in ABB's bug tracking system.
[b] How long the bug was present in the IPS before it was discovered. Numbers are based on estimates.
[c] How long it took from the time the bug was discovered until it was fixed.
[d] How long it took to validate that the bug had actually been fixed. For many bugs, this involved testing time spent at customer facilities.

8.6. Threats to validity

In this section, we discuss threats to validity for our experiments and how we address these. A possible threat to conclusion validity (i.e., when factors that can influence the conclusion drawn from the experiments) lies in the absence of a systematic analysis of all possible search heuristics in response to RQ1. Actually, we adopted a systematic analysis for variable selection heuristics by examining all 72 possible combinations of variable orderings, but we only compared two heuristics (up/down). In response to RQ2, we selected only a subset of possible parameter settings. Therefore there is another threat to conclusion validity, since nothing guarantees that another specific setting might exhibit different results. To reduce this threat, we adopted parameter settings that are realistic for the application of the constraint model in question and we responded to RQ2 by using CM_2, which is the production model. Note also that our empirical evaluation, in response to RQ3, is realized in a production environment, which considerably reduces any concern about conclusion validity.

An external validity threat of our empirical evaluation concerns the generalization of the results. Indeed, the models we developed for the experiments (i.e., CM_1 and CM_2) are specific to ABB's IPS timed sequence generation problem and cannot be easily generalized to other test generation problems. Such a threat is common in any software engineering empirical study and cannot really be reduced without applying the technology to other case studies. However, general-purpose constraint modeling languages and tools, such as SICStus Prolog and its `clpfd` library [38], address this threat and permit us to draw some generalizable perspectives from this work.

8.7. Comparison of test methods

As previously mentioned, our new MBT strategy cannot entirely replace current testing methods, but it represents an excellent supplement for identifying bugs at a much earlier stage in the development process. Nonetheless, we can still compare the different methods quantitatively. Table 3 shows the results of our comparison. As we can see from this table, our new test strategy provides a huge improvement in the number of activations that can be tested within a reasonable time frame, which is not possible with existing testing methods. If we also include automation in all aspects of testing, our strategy performs much better than our current test methods. However, it is important to note that our new method does not involve a mechanical robot and this must be regarded as a weakness.

8.8. Lessons learnt

Based on experience from about one year of live production in ABB Robotics software development environment, we report the following lessons learned, based on experience gathered through

Table 3
Comparison of constraint-based testing versus current test methods.

	Activation w/oscilloscope	Paint on paper	Constraint-based test
Setup time[a]	1–2 h[e]	3–4 h[e]	1–2 min[f]
Activations per test[b]	1	5–10	>100
Repetition time[c]	5 s	10 min	<1 s
Interpretation time[d]	<1 min[e]	2–4 min[e]	<1 s[f]
Synchronized with mechanical robot	Yes	Yes	No
Can run standalone after initial setup	Yes	No	Yes

[a] Setup time is defined as the time it takes to configure a test. This time includes upgrading the software, configuring the IPS, and loading the test.
[b] The number of physical outputs that are verified with respect to time in one test.
[c] Time needed to repeat two identical tests.
[d] Time needed to inspect and interpret the result.
[e] Manual task performed by a test engineer.
[f] Automated task performed by a computer.

development and deployment of the test framework and discussions with test engineers:

Higher confidence when changing critical parts: Based on developer feedback, there is now less worry about applying changes to critical parts of the code. Previously, such changes involved significant planning efforts and had to be coordinated with the test engineers responsible for executing tests. With the new testing framework in place, it is easy to apply a change, deploy a new build with the corresponding execution of tests, and inspect the results. If the change causes unwanted side effects, the change is rolled back to keep the build "green."

Simple front-end, complex back-end: By using Python [31] as the front-end interface to the constraint solver and keeping the interface that a test engineer is exposed to as simple as possible, we can utilize personnel with a minimal computer science background. Both Francis et al. [39] and de la Banda et al. [40] recognize that constraint programming has a steep learning curve. Even with training limited to introduction to the famous classical problems such as SEND + MORE = MONEY and the N-Queens problem [2], the test engineers have received enough training to use the constraint solver from Python without major problems.

Less focus on legacy and manual tests: A positive side effect of introducing MBT is that the focus in the organization has shifted from a great deal of manual testing toward more automatic testing. Even for products beyond the scope of this paper, the introduction of fully automatic test suites has inspired other departments to focus more on automatic testing.

Putting everything in the source control repository: In our work, we never perform *any* installation on *any* build server.[3] After a build server is installed with its continuous integration software, absolutely everything is extracted from the source control repository, as recommended in [3]. By strictly adhering to this philosophy, it is possible to utilize large farms of build servers. For example, ABB Robotics has access to large farms of build servers located in Norway, Sweden, India, and China and it is possible to schedule builds on these servers without any prior installation of special build tools. This is also the case for the new constraint programming-based tool presented in this paper. We consider the effort to develop, deploy and fully integrate our constraint-based testing tools quite demanding, but very efficient in the long run.

Keeping tests in sync with the source code and hardware: The combination of adding everything to the source control repository and JITTG is that we experience fewer problems with tests generating false errors due to a mismatch. We still have other test suites that do not have this tight integration and these tests can therefore occasionally produce false errors. The main advantage of this synchronization is experienced if a roll-back to an older version is required. In this case both the production source code and the test code is reverted to the older version.

9. Conclusions

In this paper, we present a new testing strategy for validating the timing aspects of distributed control systems for CIRs. A constraint-based mathematical model is given to automatically generate test cases through constraint solving and continuous integration techniques. The model is fully implemented and deployed within an industrial continuous integration environment. Interestingly, the constraint-based model is solved online as part of the continuous integration process. We call the online solving process JITTG.

Using JITTG guarantees that software, configuration, and hardware are kept in sync with the generated test cases. To our knowledge, this is the first time a constraint-based model using JITTG has been deployed in a continuous integration environment. The paper also answers three research questions, using the results of a thorough empirical evaluation obtained from testing a CIR system. Using a generic model that omits some technicalities, we find an ideal parameterization for constraint solving concerning variable and value ordering heuristics.

This ideal parameterization is then used on a production-grade model that is deployed at ABB's testing facilities and empirically evaluated during the validation of CIRs. This evaluation reveals that our testing strategy could not only find reinjected old faults found in previous test campaigns, but could also discover new faults. By observing that the time taken to generate a single test case in the continuous integration process typically ranges from two seconds to 13 s, we demonstrate that our strategy is faster and more effective than current test methodologies used at ABB. However, it is worth noting that our empirical evaluation does not include moving robots as part of the evaluation, which would be necessary to fully convince stakeholders of the takeaway value of our approach.

A weakness of our approach is related to the absence of guarantees with respect to model coverage. In other words, the generated test sequences does not necessarily cover every possible transition between different spray patterns. Even if this is not an industrial requirement, we believe that improving our strategy to achieve a certain test coverage is clearly an interesting research perspective.

In addition, investigating the use of our constraint-based model for other applications, such as robotized gluing, sealing, or welding, is also part of further work.

Acknowledgments

This work is funded by the Norwegian Research Council under the Industrial PhD Program (222010), the Certus SFI grant (http://www.certus-sfi.no), and ABB Robotics.

Appendix A. Notation

In Table A.4 we summarize the notation used in the mathematical model for the IPS.

Table A.4
Notation for the parameters in the production model.

Parameter	Test control parameters
N	The size of the input sequence
i	The ith sequence, $i \in [1, N]$
j	Channel number $j \in [1, 5]$
$\lvert\mathfrak{B}\rvert$	The number of different spray patterns in the model, or entries in the lookup table $\mathfrak{B}$
e	A subscript e specifies at which sequence i a scenario should start
c	A subscript c specifies on which channel j a scenario should appear
$MinBrushSep$	The minimum time between two spray pattern changes, $t_i - t_{i-1} \geqslant MinBrushSep$
$MinTrigSep$	The minimum time between two actuator output changes for some channel j
$MinOverlapTime$	The minimum time an overlap should be in the overlap scenario
$BurstTime$	The minimum time a burst of changes should last in the burst scenario
$BurstLen$	The number of changes to use in the burst scenario
IllegalVal	Value to use for the shutdown scenario
	Parameters from the robot controller
B_i	The value ith spray pattern in the test sequence
t_i	The time of the ith spray pattern in the test sequence
	Global parameters in the IPS
$PreTime$	Disturbance time between channel 1 and channels 2–4 for $P = 0 \rightarrow P > 0$
$PostTime$	Disturbance time between channel 1 and channels 2–4 for $P > 0 \rightarrow P = 0$
$\mathfrak{B}$	Brush table with $\lvert\mathfrak{B}\rvert$ rows; each row has five tuples
	Parameters for each channel
Max_j	The maximum value channel j can have
Min_j	The minimum value channel j can have
D_j^+	Parameter used to calculate timing for increasing value of the output on channel j
D_j^-	Parameter used to calculate timing for decreasing value of the output on channel j
K_j	Boolean value deciding whether or not a channel should user linear delay calculations
$P_{j,i}$	The activation value for the ith output on channel j
$t_{j,i}$	The activation time for the ith output on channel j

References

[1] M. Mossige, A. Gotlieb, H. Meling, Test generation for robotized paint systems using constraint programming in a continuous integration environment, in: IEEE Int. Conf. on Soft. Testing, Verif. and Valid (ICST'13), Poster Presentation, 2013, pp. 489–490, http://dx.doi.org/10.1109/ICST.2013.71.

[2] K. Marriott, P.J. Stuckey, Programming with Constraints: An Introduction, MIT Press, 1998.

[3] M. Fowler, M. Foemmel, Continuous Integration, 2006. <http://martinfowler.com/articles/continuousIntegration.html> (accessed 13.08.13).

[4] B. Fitzgerald, K.-J. Stol, Continuous software engineering and beyond: trends and challenges, in: First Workshop on Rapid Continuous Software Engineering (RCoSE) Co-located with ICSE, vol. 14, 2014.

[5] S. Dösinger, R. Mordinyi, S. Biffl, Communicating continuous integration servers for increasing effectiveness of automated testing, in: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ACM, 2012, pp. 374–377.

[6] A. Orso, G. Rothermel, Software testing: a research travelogue (2000–2014), in: Proceedings of the IEEE International Conference on Software Engineering (ICSE), Future of Software Engineering, 2014.

[7] H. Do, G. Rothermel, A. Kinneer, Empirical studies of test case prioritization in a JUnit testing environment, in: Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on, IEEE, 2004, pp. 113–124.

[8] T.Y. Chen, M.F. Lau, A new heuristic for test suite reduction, Inform. Softw. Technol. 40 (1998) 347–354.

[9] D. Marijan, A. Gotlieb, S. Sen, Test case prioritization for continuous regression testing: an industrial case study, in: Software Maintenance (ICSM), 2013 29th IEEE International Conference on, IEEE, 2013, pp. 540–543.

[10] J.S. Bell, G.E. Kaiser, Unit test virtualization with vmvm, in: Companion Proceedings of the 36th International Conference on Software Engineering (ICSE), ACM, 2014.

[11] D. Hao, L. Zhang, X. Wu, H. Mei, G. Rothermel, On-demand test suite reduction, in: Proceedings of the 2012 International Conference on Software Engineering, IEEE Press, 2012, pp. 738–748.

[12] J.H. Hill, D.C. Schmidt, A.A. Porter, J.M. Slaby, CiCUTS: combining system execution modeling tools with continuous integration environments, in: Engineering of Computer Based Systems, 2008. ECBS 2008. 15th Annual IEEE International Conference and Workshop on the, IEEE, 2008, pp. 66–75.

[13] Y. Zhao, J. Liu, E.A. Lee, A programming model for time-synchronized distributed real-time systems, in: 13th IEEE Real Time and Embedded Technology and Applications Symposium, 2007. RTAS '07, 2007, pp. 259–268. <http://chess.eecs.berkeley.edu/pubs/325.html>.

[14] P. Ramadge, W. Wonham, The control of discrete event systems, Proc. IEEE 77 (1989) 81–98.

[15] C.G. Cassandras, S. Lafortune, Introduction to Discrete Event Systems, Springer, 2008.

[16] K. Lee, J. Eidson, Ieee-1588 standard for a precision clock synchronization protocol for networked measurement and control systems, in: In 34th Annual Precise Time and Time Interval (PTTI) Meeting, 2002, pp. 98–105.

[17] S. Johannessen, Time synchronization in a local area network, IEEE Control Syst. 24 (2004) 61–69.

[18] R.R. Rajkumar, I. Lee, L. Sha, J. Stankovic, Cyber-physical systems: the next computing revolution, in: Proceedings of the 47th Design Automation Conference, ACM, 2010, pp. 731–736.

[19] L. Sha, S. Gopalakrishnan, X. Liu, Q. Wang, Cyber-physical systems: a new frontier, in: Machine Learning in Cyber Trust, Springer, 2009, pp. 3–13.

[20] E.A. Lee, Cyber physical systems: design challenges, in: Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on, IEEE, 2008, pp. 363–369.

[21] M. Broy, S. Chakraborty, S. Ramesh, M. Satpathy, S. Resmerita, W. Pree, Cross-layer analysis, testing and verification of automotive control software, in: Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on, IEEE, 2011, pp. 263–272.

[22] M. Utting, B. Legeard, Practical Model-Based Testing: A Tools Approach, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[23] M. Barnett, M. Fähndrich, K.R.M. Leino, P. Müller, W. Schulte, H. Venter, Specification and verification: the spec# experience, Commun. ACM 54 (2011) 81–91.

[24] A. Gotlieb, B. Botella, M. Rueher, Automatic test data generation using constraint solving techniques, in: Proc. of Int. Symp. on Soft. Testing and Analysis (ISSTA'98), 1998, pp. 53–62.

[25] B. Marre, B. Blanc, Test selection strategies for lustre descriptions in gatel, Electron. Notes Theor. Comput. Sci. 111 (2005) 93–111.

[26] S. Di Alesio, S. Nejati, L. Briand, A. Gotlieb, Stress testing of task deadlines: a constraint programming approach, in: Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on, IEEE, 2013, pp. 158–167.

[27] A. Gotlieb, T. Denmat, B. Botella, Goal-oriented test data generation for pointer programs, Inform. Softw. Technol. 49 (2007) 1030–1044.

[28] S. Di Alesio, S. Nejati, L. Briand, A. Gotlieb, Stress testing of task deadlines: a constraint programming approach, in: Int. Symposium on Soft. Reliability and Engineering (ISSRE'13), Research Track, Pasadena, CA, USA, 2013.

[29] A.W. Ulrich, P. Zimmerer, G. Chrobok-Diening, Test architectures for testing distributed systems, in: Proceedings of the 12th International Software Quality Week, 1999.

[30] European Parliament and Council of the European Union, Directive 2006/42/EC on machinery, 2006.

[31] G. Rossum, Python Reference Manual, Technical Report, Amsterdam, The Netherlands, 1995.

[32] M. Utting, A. Pretschner, B. Legeard, A taxonomy of model-based testing approaches, Softw. Test. Verif. Reliab. 22 (2012) 297–312.

[33] P. Van Hentenryck, Constraint Satisfaction in Logic Programming, MIT Press, 1989.

[34] F. Rossi, P.v. Beek, T. Walsh, Handbook of Constraint Programming (Foundations of Artificial Intelligence), Elsevier Science Inc., New York, USA, 2006.

[35] L. De Moura, N. Bjørner, Z3: an efficient SMT solver, in: TACAS'08/ETAPS'08: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer-Verlag, 2008, pp. 337–340.

[36] P. McMinn, Search-based software test data generation: a survey, Softw. Testing, Verif. Reliab. 14 (2004) 105–156.

[37] I. IBM, ILOG Labs, IBM CPLEX: High-Performance Software for Mathematical Programming and Optimization, 2006. <http://www.ilog.com/products/cplex/>.

[38] M. Carlsson, G. Ottosson, B. Carlson, An open-ended finite domain constraint solver, in: Proceedings of the9th International Symposium on Programming Languages: Implementations, Logics, and Programs: Including a Special Track on Declarative Programming Languages in Education, PLILP '97, Springer-Verlag, London, UK, 1997, pp. 191–206. <http://dl.acm.org/citation.cfm?id=646452.692956>.

[39] K. Francis, S. Brand, P.J. Stuckey, Optimisation modelling for software developers, in: Principles and Practice of Constraint Programming, Springer, 2012, pp. 274–289.

[40] M.G. de la Banda, P.J. Stuckey, P. Van Hentenryck, M. Wallace, The future of optimization technology, Constraints (2013) 1–13.