# Testing Robotized Paint System Using Constraint Programming: An Industrial Case Study

Morten Mossige, Arnaud Gotlieb, Hein Meling



# Testing Robotized Paint System using Constraint Programming: An Industrial Case Study

Morten Mossige[1,3], Arnaud Gotlieb[2], and Hein Meling[3]

[1] ABB Robotics, Norway. morten.mossige@no.abb.com
[2] Simula Research Laboratory, Norway. arnaud@simula.no
[3] University of Stavanger, Norway. hein.meling@uis.no

**Abstract.** Advanced industrial robots are composed of several independent control systems. Particularly, robots that perform process-intensive tasks such as painting, gluing, and sealing have dedicated *process control systems* that are more or less loosely coupled with the motion control system. Validating software for such robotic systems is challenging. A major reason for this is that testing the software for such systems requires access to physical systems to test many of their characteristics.
In this paper, we present a method, developed at ABB Robotics in collaboration with SIMULA, for automated testing of such process control systems. Our approach draws on previous work from continuous integration and the use of well-established constraint-based testing and optimization techniques. We present a constraint-based model for automatically generating test sequences where we both *generate* and *execute* the tests as part of a continuous integration process. We also present results from a pilot installation at ABB Robotic where ABB's Integrated Process Control system has been tested. The results show that the model is both able to discover completely new errors and to detect old reintroduced errors. From this experience, we report on lessons learned from implementing an automatic test generation and execution process for a distributed control system for industrial robots.

## 1 Introduction

Developing reliable software for Complex Industrial Robots (CIRs) is a complex task, because typical robots are comprised of numerous components, including computers, field-programmable gate arrays (FPGAs), and sensor devices. These components typically interact through a range of different interconnection technologies, e.g. Ethernet and dual port RAM, depending on delay and latency requirements on their communication. As the complexity of robot control systems continuous to grow, developing and validating software for CIRs is becoming increasingly difficult. For robots performing process-intensive tasks such as painting, gluing, or sealing, the problem is even worse as their dedicated process control systems is loosely coupled with the robot motion control system. A key feature of robotized painting is the ability to perform precise activation

of the process equipment along a robot's programmed path. At ABB Robotics, Norway, they develop and validate Integrated Painting control Systems (IPS) for CIRs and are constantly improving the processes to deliver products that are more reliable to their customers.

Current practices for validating the IPS software involve designing and executing manual test scenarios. In order to reduce the testing costs and to improve quality assurance, there is a growing trend to automate the generation of test scenarios and multiplying them in the context of continuous testing. To facilitate this automation, we abandon the use of physical robots and instead run our tests on the IPS controllers with outputs connected to actuators. We monitor these outputs and compare with the expected results for the different tests.

In this paper, we report on our work using Constraint Programming (CP) over finite domains to *generate* automatically timed-event sequences (i.e., test scenarios) for the IPS and *execute* them within a Continuous Integration (CI) process [1]. Building on initial ideas sketched in a poster [2] one year ago, we have developed a constrained optimization model in SICStus Prolog `clpfd` [3] to help test the IPS under operational conditions. Due to online configurability of the IPS, test scenarios must be reproduced every day, meaning that indispensable trade-offs between optimality and efficiency must be found, to increase the capabilities of the CI process to reveal software defects as early as possible. While using CP to generate model-based test scenarios is not a completely new idea [4,5], it is to our knowledge, the first time that a CP model and its solving process been integrated into a CI environment for testing complex distributed systems.

**Organization.** The rest of the paper is organized as follows: Section 2 presents some background on robotized painting, with an example serving as a basis for describing the mathematical relations involved ; Section 3 describes ABB Robotics' current testing practices of the IPS and the rationale behind our validation choices ; Section 4 presents test objectives and scenarios ; Section 5 explains how the model is implemented and how it is integrated in in the CI process ; Section 6 discuss the errors found using the new model and compare our new approach with existing testing methods. Finally, Section 7 present the lessons learned from using the new testing approach and outlines ideas for further work, before we conclude the paper.

**Notation.** Throughout this paper the following notations is used: A symbol prefixed with a $^*$, as in $^*SeqLen$, is to be regarded as a constant integer value given as *input* to the CP-model. An underlined symbol, as $\underline{D^+}$, is to be regarded as an integer *generated* by the CP-model, i.e. a result of the solving process.

## 2 Robotized Painting

This section briefly introduces robotized painting. A robot system dedicated to painting typically consists of two main parts: the robot controller, responsible for moving the mechanical arm, and the IPS, responsible for controlling the paint process. That is, to control the activation and deactivation of several physical

processes such as paint pumps, air flows, air pressures, and to synchronize these with the motion of the robot arm.

A *spray pattern* is defined as the combination of the different physical processes. Typically, the physical processes involved in a spray pattern will have different response times. For instance, a pump may have a response time in the range 40-50 ms, while the airflow response time is in the range 100-150 ms. The IPS can adjust for these differences using sophisticated algorithms that have been analyzed and tuned over the years to serve different needs. In this paper, we focus on validating the timing aspects of the IPS.

### 2.1 Example of Robotized Painting

We now give a concrete example of how a robot controller communicates with the IPS in order to generate a spray pattern along the robot's path. A schematic overview of the example is shown in Figure 1, where the node marked *robot controller* is the CPU interpreting a user program and controlling the servo motors of the robot in order to move it. The example is realistic, but simplified, in order to keep the explanations as simple as possible.

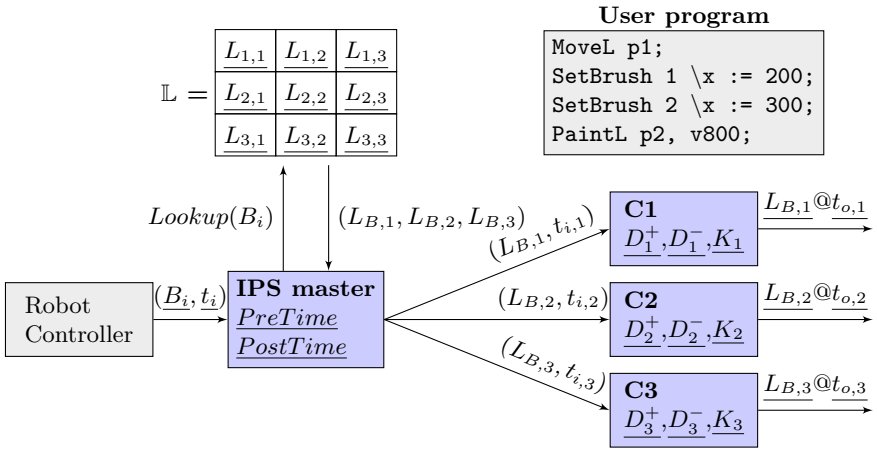



Fig. 1: Logical overview of a robot controller and the IPS.

The program listing of Figure 1 shows an example of a user program. The first instruction `MoveL p1` moves the robot to the Cartesian point `p1`. The next two `SetBrush` instructions tells the robot to apply spray pattern number 1 when the robot reaches $x = 200$ on the $x$-plane, and to apply spray pattern number 2 when it reaches $x = 300$. Both `SetBrush` instructions tell the IPS to apply a specific behavior when the physical robot arm is at a given position. The last instruction (`PaintL`) starts the movement of the robot from the current position `p1` to `p2`. The `v800` argument of `PaintL` gives the speed of the movement (i.e., 800 mm/s).

We now assume that the path from `p1` to `p2` results in a movement from $x = 0$ to $x = 500$. The robot controller interprets the user program ahead of the actual physical movement of the robot, and can therefore estimate *when* the robot will be at a specific position. Assuming that the movement starts at time $t = 0$, the robot can compute based on speed and length of path that the two `SetBrush` activations should be triggered at $t_1 = 250$ ms and $t_2 = 375$ ms.

The robot controller now sends the following messages (a.k.a. *events*) to the IPS master: $(B_1 = 1, t_1 = 250), (B_2 = 2, t_2 = 375)$, which means apply spray pattern 1 at 250 ms, and spray pattern 2 at 375 ms. The messages are sent around 200 ms before the actual activation time, or at $\approx 50$ ms for spray pattern 1, and at $\approx 175$ ms for spray pattern 2. These messages simply converts position into an absolute global activation time. Note also that the IPS receives the second message before the first spray pattern is bound for execution, which means that the IPS must handle a queue of scheduled spray patterns.

**IPS master:** When the IPS receives a message from the robot controller, it first determines the physical outputs associated with the logical spray pattern number. Many different spray patterns can be generated based on factors like paint type or equipment in use. In the IPS, each spray pattern is translated into 3 to 6 different physical actuator outputs that must be activated at appropriate times, possibly different from each other. In this example, we use three different physical actuator outputs.

Figure 1 shows the three different actuator outputs (**C1, C2, C3**). The output value of each actuator output for a given spray pattern is resolved by using a *brush table* ($\mathbb{L}$) (simply a lookup table). In this example, $\mathbb{L}(B_1 = 1)$ returns $(L_{1,1}, L_{1,2}, L_{1,3})$, while $\mathbb{L}(B_2 = 2)$ results in $(L_{2,1}, L_{2,2}, L_{2,3})$. The IPS master will now pass these values to each actuator output along with its activation time, which may be different from the original time received from the robot controller ($t_i$). This possible modification can be formalized as follows:

$$t'_i = \begin{cases} t_i - \underline{PreTime} & \text{if } L_{1,B_{i-1}} = 0 \wedge L_{1,B_i} \neq 0 \\ t_i - \underline{PostTime} & \text{if } L_{1,B_{i-1}} \neq 0 \wedge L_{1,B_i} = 0 \end{cases} \quad (1)$$

What equation (1) shows is that the activation time of each actuator output may be adjusted by a constant factor ($\underline{PreTime}$, $\underline{PostTime}$), depending on changes from other actuator outputs. This is done because small adjustments may be necessary when there is a direct link between the timing of different actuator outputs. In our example, the timing on **C2** is influenced by changes on **C1**. A practical example of such compensation is to change the timing on an air actuator output when a paint-pump is turned on or off.

**Activation of actuator outputs:** By still referring to Figure 1, we now present how a message sent from the IPS master to a single actuator output is handled. Let us assume that message $(L, t_i)$ is sent, and that the current actuator output is $L'$. Since painting involves many slow physical processes, the actuator output compensates for this by computing an adjusted activation time $t_o$, that accounts for the time it takes the physical process to apply the change.

The IPS can adopt two different strategies to compute this time compensation. The first consists of adjusting the time with a *constant* factor: $\overline{D^+}$ for positive change, and $\overline{D^-}$ for negative change. The second one consists in using a *linear* timing function to adjust the time *linear* to the change of the physical output.

Equation (2) combines these strategies into a single compensation function, where ${}^*Min$ (resp. ${}^*Max$) is the physical minimum (resp. maximum) value possibly handled[4] by some actuator output.

$$t_o = t_i - \begin{cases} \overline{D^-} \cdot (\frac{L-L'}{{}^*Max-{}^*Min})^{\overline{K}} & \text{if } L' < L \\ \overline{D^+} \cdot (\frac{L'-L}{{}^*Max-{}^*Min})^{\overline{K}} & \text{if } L' > L \\ \overline{0} & \text{otherwise} \end{cases} \tag{2}$$

**Physical layout of the IPS:** Figure 1 only shows the logical connections in a possible IPS configuration. In real applications, each component (**IPS master, C1, C2, C3**) may be located on different embedded controllers, interconnected through an industrial-grade network. As such, the different components may be located at different physical locations on the robot, depending on which of physical process it is responsible for.

## 3 Testing IPS

In this section, we summarize some of the testing approaches that have been used to test the IPS, focusing on validating the accuracy of time-based activation. We discuss the benefits and drawbacks of these legacy-testing approaches, before we introduce our automated testing approach.

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

### 3.1 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 Figure 2[5]. This test includes both realistic movement of the robot while running a complete and realistic setup on the IPS. However, there are many drawbacks to using this method, where one is high cost of much manual labor to set up, and hazardous environment due paint fluid exposure. It is also more or less impossible to automate this test, even after the initial setup is done.

[4] These values are determined by the physical equipment involved in the paint process (pumps, valves, air, etc.).

[5] This video `http://youtu.be/oq524vuO5N8` also shows painting on paper.

A typical test execution will be to perform a single paint-stroke as shown in Figure 2, followed by manual inspection of the result. This inspection will typically require both a software engineer and a paint process engineer to interpret the results. Because of this manual inspection that need to be done between each paint-stroke, the possibility of automating this test method is limited. Such a test is typically performed during the final verification phase for a new product running the IPS software, such as the air controller of a paint controller. The test is also executed as part of major revisions of the IPS. Based on our experience at ABB Robotics, it is extremely rare that errors related to timing are found when running such a test.

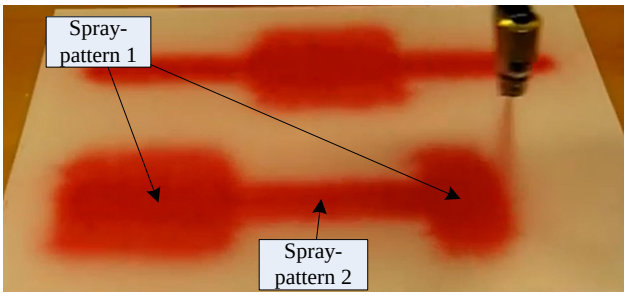



Fig. 2: Painting on paper.

### 3.2 Activation Testing with an Oscilloscope

By reducing the IPS setup to include only a single digital actuator output, without any fluid or air, it is possible to perform some of the basic synchronization tests on the IPS. This can be done by connecting the digital output to an input channel on an oscilloscope, and the output from a proximity sensor to another input channel. The robot is then programmed to perform a linear movement passing over a proximity sensor, with the paint program set up to do an activation at exactly that point. Thus generating a signal which should be in sync with the signal from the proximity sensor. By comparing the signal from the digital output with the signal from proximity sensor, it is possible to test much of the timing behaviors of the IPS[6].

At ABB Robotics, this is one of the most frequently executed tests aimed at uncovering synchronization problems. However, also this test require manual labor to set up and to execute the test runs, and since it also involves physical movement of the robot arm, a hazard zone must be established for this test. However, different from the test described in Section 3.1, it can be executed without supervision, and the test results can be inspected after test completion.

[6] Two videos showing activation testing using a proximity sensor and oscilloscope: `http://youtu.be/I1Ce37_SUwc` and `http://youtu.be/LgxXd_DN2Kg`.

### 3.3 Running in a Simulated Environment

The IPS is designed to be portable to many microprocessor platforms and operating systems. It is even possible to run the IPS on a desktop system such as Windows. This has the advantage that much of the functional testing can be performed in a simulated environment. This has reduced 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 lack of real-time behavior in the simulator.

### 3.4 Summary of Existing Test Methods

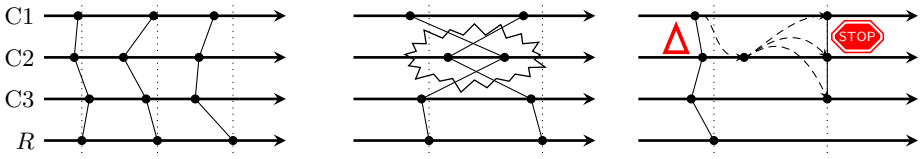
There are several drawbacks with the test methods described above. The methods that use real robots gives very realistic results, but require costly and slow manual labor to set up the test, and to interpret the results. For the method described in Section 3.3 it is clearly possible to automate both setup and result analysis. However, it cannot be used to execute tests related to real-time and synchronization between several embedded controllers.

### 3.5 A New Test Strategy

In the following, we outline some of the requirements for our new test strategy. The rationale behind the new requirements are mainly to reduce manual labor, and to be able to detect errors earlier in the development process: *1*) Automated: It should be possible to set up the test, execute the test, and analyze the results without human intervention. *2*) Systematic: Tests should be generated automatically by a model, rather than being contrived by a test engineer. *3*) 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.

## 4 Constraints and Scenario

With the mathematical relations described in Section 2 as background, we now describe the constraints used to generate test sequences. The constraints can be divided into two main categories. The first category expresses how a complete IPS behaves in a normal situation. That is, the IPS is not forced into any erroneous state, or other corner cases. The second category represents constraints where the model generates specially selected error scenarios, where the IPS is either pushed into an erroneous state, or where performance limitations are being tested. These constraints are summarized in Figure 3, and discussed in detail in the following sections.




(a) Normal. (b) Overlap. (c) Kill.

Fig. 3: Test scenarios considered as test objectives. The horizontal axis represent time, and the black dots correspond to output activations. A specific *spray pattern* is a collection of output activations, and is visualized by a line connecting the black dots.

### 4.1 Normal Scenario

During normal, non-erroneous behavior, the robot controller sends commands to the IPS according to the constraint given in (3), and the IPS generates output activations according to (4). These two constraints correspond to Figure 3a.

**Input sequence:**

$$\begin{aligned} &\forall i \in 1 \ldots {}^{*}SeqLen, \\ &t_i - t_{i-1} \geq {}^{*}MinBrushSep, \\ &t_i > t_{i-1}, \quad t_i \geq 0, \\ &B_i \neq B_{i-1}, \quad B_i \in 0 \ldots {}^{*}BTabSize \end{aligned} \tag{3}$$

**Each single actuator output:**

$$\begin{aligned} &\forall j \in 1 \ldots {}^{*}C, \forall i \in 1 \ldots {}^{*}SeqLen \\ &t_{j,i} - t_{j,i-1} \geq {}^{*}MinTrigSep, \\ &t_{j,i} > t_{j,i-1}, \quad t_{j,i} \geq 0 \end{aligned} \tag{4}$$

Where ${}^{*}C$ refers to the number of actuator output in the setup.

### 4.2 The Overlap Scenario

Figure 3b shows two overlapping event activations. This overlap can be generated using the constraint in (5). This scenario is used to test how well the IPS is able to handle that an activation time for one actuator output gets out of order with respect to time. For real robot applications, there are many possible sources for this particular problem scenario. The most common one being that a user increase the robot's speed by changing the speed parameter in a `PaintL` instruction. For the IPS, this behavior should be reported as an error message to the user, and resolve any conflicts with the actuator output.

$$\begin{aligned} t_{c,e} - t_{c,e+1} &\geq {}^{*}MinOverlapTime, \\ t_{c,e+1} - t_{c,e-1} &\geq {}^{*}MinTrigSep, \\ t_{c,e+2} - t_{c,e} &\geq {}^{*}MinOverlapTime \end{aligned} \tag{5}$$

Above, $t_{c,e}$ represents the activation time for a specific actuator output $c$ and $e$ indicates where in the sequence the overlap should be enforced.

### 4.3 The Illegal Value Scenario

The illegal value constraint is important for checking that the IPS is able to shut down safely in case of an error. By forcing one of the output channels to fail, the internal structure of the IPS should initiate a controlled shutdown. This shutdown procedure must be performed in a special sequence, taking care to avoid pressure build up in hoses, which could otherwise lead to rapturing hoses. This constraint is visualized in Figure 3c, and specified in Equation (6).

$$P_{c,e} = {}^{*}IllegalVal \tag{6}$$

Where $P_{c,e}$ is the output value of actuator output $c$ for the $e'th$ event.

## 5 Implementation

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

### 5.1 Continuous Integration

CI [1] is a software engineering practice aimed at uncovering software errors at an early stage of development, to avoid problems during integration testing. One of the key ideas in CI is to build, integrate, and test the software frequently. For developers working in a CI environment, this means that even small source code changes should be submitted to the source-control system server frequently, rather than submitting large sets of changes occasionally. Typically, a CI infrastructure includes tools for source control system, automated build servers, and testing engines.

### 5.2 Testing in a CI Environment

Our goal in this work has been to develop an automated testing framework for the IPS as an integrated part of a CI environment. Compared to traditional software testing, a test running in a CI environment will have some additional requirements. In particular, Fowler [1] points out that the total *round-trip time* is crucial for a successful CI deployment. Here, round-trip time refers to the time it takes from a developer submits a change to the source repository, until feedback from the build and test processes can be collected. Thus, to keep the round-trip time to a minimum, we have identified a few areas that we have given special attention to in our design.

**Test complexity**: Testing based on CI will never replace all other kinds of testing. In CI, a less accurate but faster test will always be preferred over a slow but

accurate test. In practice, a test must satisfy the *good enough* criteria, frequently used in industry.
**Solving time**: Many constraint-based optimization problems are time consuming to solve, especially if an optimal solution is sought. For our purposes, an optimal solution is unnecessary. Simply finding a good solution after running the solver for a limited period is considered good enough for our purpose.
**Execution time**: The most important factor for controlling the execution time for a generated test sequence is to control the length of the test sequence. By balancing the length of the test sequence between a reasonable, realistic enough but still small enough, the execution time of the test is kept under control. Clearly, it will require some experience, to determine what is *good enough*.

### 5.3 Model Implementation

We choose to use *Constraint Programming* (CP) over finite domains to solve the model. CP is a well-known paradigm introduced twenty years ago to solve combinatorial problems in an efficient and flexible way [6]. Typically, a CP model is composed of a set of variables $V$, a set of domains $D$, and a set of constraints $C$. The goal of constraint resolution is to find a solution, i.e., an assignment of variables to values that belong to the domains and that satisfy all the constraints. Finding solutions is the role of the underlying constraint solver, which applies several filtering techniques to prune the search space formed by all the possible combinations. In practice, the constraint models that are developed to solve concrete and realistic testing problems usually contain complex control conditions (conditionals, disjunctions, recursions) and integrate dedicated and optimized search procedures [7].

We have chosen to implement the constraint model using the finite domains library in Sicstus Prolog [3], and have thus written the actual model in Prolog. To integrate the model with ABB's existing testing framework, we have also built a front-end layer in Python. This front-end layer can be used by test engineers with no prior knowledge of CP or Prolog, and 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 Figure 4.

While referring to Figure 1, we now assume that the number of actuator outputs is a constant input parameter ${}^*C$, instead of 3. The variables for our problem can be divided into three groups: the input sequence $((\underline{B_1}, \underline{t_1}), \ldots, (\underline{B_N}, \underline{t_N}))$, where $N = {}^*SeqLen$ (the length of the test sequence). The configuration variables: $(\underline{PreTime}, \underline{PostTime}, \underline{D_1^+}, \underline{D_1^-}, \underline{K_1}, \ldots, \underline{D_{*C}^+}, \underline{D_{*C}^-}, \underline{K_{*C}})$, and the variables in the brush table $(\underline{L_{1,1}}, \ldots, \underline{L_{*BTabSize,*C}})$ where ${}^*BTabSize$ is the size of the brush table. By selecting one of the previously mentioned scenarios, a solution to the CP model is constructed by an instantiation of these variables. Thus, the expected output on each actuator output along with its activation time, represents the *test oracle*.

### 5.4 Test Setup

An overview of our test setup is shown in Figure 4. With reference to the numbers in Figure 4 a test sequence execution is typically triggered by a build server upon a successful build of the IPS software, (1). Building the software is scheduled to run every night, but can also be triggered manually by a developer. The first task the test server does is to upgrade all connected embedded controllers with the newly built software, (2). Then the IPS is configured with test specifications retrieved from the source control repository, (3). This is followed by executing a set of basic *smoke tests*, before the constraint model is launched for test cases generation. By feeding the constraint model with data retrieved from the new configuration and properties from the IPS (4), we ensure that the generated tests are synchronized with the current software and configuration. Further details on just-in-time test generation (JITTG) is discussed in Section 5.5. Finally, the actual test is executed by applying the generated test sequence, and comparing the actuator outputs with the model generated oracle, (5).

In the production test facility at ABB, each generated test sequence is executed on a total of 11 different configurations. The different configurations include execution on different hardware and software generations of the IPS, and execution on both VxWorks and Linux as the base operating system for the IPS. The test framework is written in Python and supports parallel execution of tests when there are no shared resources. Thus, the time to run the test sequence on many different configuration is very low, compared to running them in a sequence, one at a time.

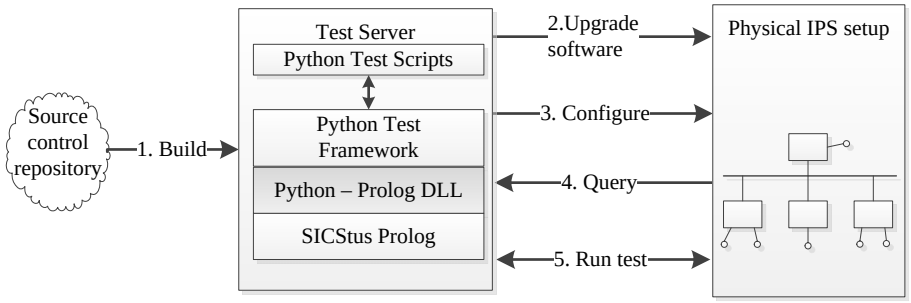



Fig. 4: Integration of CP-model with physical setup.

### 5.5 Just-in-Time Test Generation

As shown earlier in this paper, there are many parameters in the model that must be specified before the model can be solved. Some of these parameters come from configuration files used to configure the IPS, and some of the parameters can be extracted by querying a newly built IPS. Common for both sets of parameters is that the resulting model will be different if the parameters have changed. This

means that the model is closely linked to what is fetched from the source control system. Because of this, we decided to generate and solve the model at testing time, as opposite to solving the model once, and adding the resulting model to source control. There are several advantages to using JITTG, but there are also some drawbacks. The most important advantage of using JITTG is that there is a lower probability of falsely reporting an error due to mismatch between the generated model and the running system. The main drawback is that the time needed for solving the model becomes an important factor. 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 a crucial. The models that we have solved so far have had a solving time of less than a few minutes.

## 6 Evaluation

This section presents the errors that were found after having used the new model in ABB Robotics production test facility for about four months. We also compare our new model-based testing approach with some of the other testing practices.

### 6.1 New Errors Found

This section describes the errors found immediately after we introduced the new model. We found three previously unknown errors in the IPS. These errors have been in the IPS for some time, and were only found through the use of the new model. Two of the errors was directly related to the behavior of the IPS, while the third was related to how a PC tool present online diagnostics from a live, running system. It is important to emphasize that tests executed in a CI framework is primarily designed to detect errors caused by source code changes committed to the source control repository. Consequently, when a test framework like this is first introduced, it will not necessarily find many new errors in a mature, well-designed system. During normal operation, the test framework should only rarely find any errors.

**PreTime/PostTime Calculation Error:** If an actuator output was assigned a value 0, while at the same time another actuator output was set up to slave the timing of the first actuator output, using the logic specified in Equation (1), this could lead to an erroneous calculation of the timing of the slaved actuator output. This error was found by running the *normal behavior* scenario as described in Section 4.1.

**Event Queuing Blocks Shutdown:** If a shutdown of the IPS occurred at the same time as a queue of actuator output events was scheduled for execution, the shutdown could be delayed. This could happen because the shutdown message was treated with the same priority as a normal output message, when placed in the scheduling queue.

**Shift in Clock:** There exist several online monitoring tools like RobView and Robot Studio. These tools can be used for online monitoring of the process data from the IPS. To be able to present very accurate timing on the presented

signals, the clock on the PC running the tool, and the clock on the IPS is synchronized through a proprietary light weight version of the IEEE 1588 clock synchronization protocol. Our test framework found an error in this synchronization protocol, which could cause the reported time of a signal change to deviate by as much as 10 ms compared to the actual time of the change. The error could even cause changes in the causal ordering of events.

Table 1: Historical data on old bugs that were reintroduced.

| Bug#[1] | Time in system[2] | Time to solve[3] | Time to validate[4] | Detected by model |
|---|---|---|---|---|
| 44432 | 5-10 years | 1-2 hours | 1 day | Yes |
| 44835 | 5-10 years | 2-4 days | 1 day | Yes |
| 27675 | 6-12 months | 1-2 months | 1-2 weeks | Yes |
| 28859 | 6-12 months | 2-3 months | 2-3 weeks | Yes |
| 28638 | 4-6 months | 1-2 weeks | 2-3 weeks | Yes |

[1] Refer to bug-number in ABB's bug tracking system.
[2] How long the bug was present in the IPS before it was discovered.
[3] How long it took from the bug was discovered until the bug was fixed.
[4] How long it took to validate that the bug had actually been fixed. For many bugs, this involved testing time spent at customer facilities.

### 6.2 Detection of Old Errors

To further validate the robustness of the model, a collection of old, previously detected errors were reintroduced into source control repository with the intention to verify that the model was able to detect the errors. The selected errors were chosen by searching in ABB's bug tracking system, by interviewing ABB's test engineers, and through discussions with the main architect of the IPS. Most of the errors were originally discovered at customer sites during staging of 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 put into a large configuration with many components.

The chosen errors are summarized in Table 1. The table shows historical data on how long it original 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. Finally, the table shows if our new model was able to detect the re-injected bug. Note that some of these numbers cannot be accurately specified; they represent reasonable estimates. Especially errors related to *how* long a bug has been in the system is difficult to estimate. However, by interviewing the main architect of the IPS and the lead test engineer, we feel that the numbers presented has high confidence.

### 6.3 Comparison of Test Methods

While our new model-based test strategy cannot entirely replace the current testing methods, it represents an excellent supplement to identify bugs at a much earlier stage in the development process. Nonetheless, we can still compare the different methods quantitatively. Table 2 gives the results of our comparison. As we can see from this table, our new test strategy gives a huge improvement in number of activations that can be tested within a reasonable timeframe, which is not possible with existing testing methods. If we also include the 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.

Table 2: Comparison of test methods.

| | Activation w/oscilloscope | Paint on paper | Constraint-based test |
|---|---|---|---|
| Setup time[1] | 1-2 hours[5] | 3-4 hours[5] | 1-2 min[6] |
| Activations per test[2] | 1 | 5-10 | >100 |
| Repetition time[3] | 5 sec | 10 min | < 1 sec |
| Interpretation time[4] | < 1 min[5] | 2-4 min[5] | < 1 sec[6] |
| Synch. with mechanical robot | Yes | Yes | No |
| Run stand alone after initial setup | Yes | No | Yes |

[1] Time to upgrade software, configure IPS, and loading test.
[2] The number of physical outputs that are verified with respect to time in one test.
[3] Time needed to repeat two identical tests.
[4] Time needed to inspect and interpret the result.
[5] Manual task performed by a test engineer.
[6] Automated task performed by a computer.

## 7 Conclusion

This section concludes the paper by presenting some lessons learned from our experience at ABB Robotics of introducing CP in our CI process. We also outline some ideas for further work.

### 7.1 Lessons Learned

Based on experience from about one year of live production in ABB Robotics software development environment, we report on lessons learned. The lessons learned presented here are based on experience gathered through development

and deployment of the test framework, and on discussion with test engineers. We summarize lessons learned in the following:

**Higher confidence when changing critical parts:** Based on feedback from developers, there are now less 'fear' of applying changes in critical parts of the code. Previously such changes involved significant planning efforts and had to be synchronized with 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 result. If the change has caused unwanted side effects, the change is rolled back to keep the build 'green'.

**Simple front-end, complex back-end:** By using Python [8] 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 are able to utilize personnel with minimal computer science background. It has been recognized by both [9] and [10] that CP has a high threshold for usage. By limiting the training to introduction of the famous classical problems like 'SEND+MORE=MONEY' and the N-Queens problem [11], the test engineers have received enough training to use the constraint solver from Python without major problems.

**Less focus on legacy, manual tests:** A positive side effect of introducing model-based testing is that the focus in the organization has shifted from lots of manual testing towards more automatic testing. Even for products beyond the scope of this paper, the introduction of a fully automatic test suite has inspired other departments to focus more on automatic testing.

**Putting everything in source control repository:** In our work, we never perform any installation on any build server. After a build server is installed with its CI software, absolutely everything is extracted from the source control repository. This is also a clear recommendation in CI [1]. By being 100 % strict to this philosophy, it is possible to utilize big farms of build servers without any prior installation of special build tools. This is also the case for the new CP based tool presented in this paper. We consider that the effort taken to do a 100 % integration of the tools is quite demanding, but in the long run very efficient.

**Keeping tests in sync with source code and hardware:** The combination of adding 'everything' to the source control repository and JITTG is that we experience less problems with tests generating false errors due to a mismatch. We still have other test suites where we do not have this tight integration, and thus these tests may occasionally produce false errors.

### 7.2 Further work

The very promising results we got from use of the model inside the CI environment, has inspired us to also make use of the model outside the CI environment. This will enable us to use the model to generate tests where there is a focus on tests with longer duration. When using the model inside the CI environment, the focus is always on short duration tests, where each test never exceeds 3 minutes.

### 7.3 Conclusions

This paper introduced a new testing strategy for validating timing aspects of complex distributed control systems for robotics systems. A constraint-based mathematical model was introduced to automatically generate test cases through constraint solving and continuous integration techniques. The model, designed through a collaboration with SIMULA, has been implemented and a pilot installation has been set up at ABB Robotics. Using the model, both previously discovered bugs, as well as new bugs were found in a period that has been dramatically reduced with respect to other current testing practices.

**Acknowledgements** This work is funded by the Norwegian Research Council under the frame of the Industrial PhD program, the Certus SFI grant and ABB Robotics.

## References

1. Fowler, M., Foemmel, M.: Continuous integration (2006) [Online; accessed 13-August-2013].
2. Mossige, M., Gotlieb, A., Meling, H.: Poster: Test generation for robotized paint systems using constraint programming in a continuous integration environment. In: Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on. (2013) 489–490
3. Carlsson, M., Ottosson, G., Carlson, B.: An open-ended finite domain constraint solver. In: Proceedings of the9th International Symposium on Programming Languages: Implementations, Logics, and Programs: Including a Special Trach on Declarative Programming Languages in Education. PLILP '97, London, UK, UK, Springer-Verlag (1997) 191–206
4. Di Alesio, S., Nejati, S., Briand, L., Gotlieb, A.: Stress testing of task deadlines: A constraint programming approach. In: Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on, IEEE (2013) 158–167
5. K. Balck, O.G., Pearson, J.: Model-based protocol log generation for testing a telecommunication test harness using clp. In: Proceedings of DATE 2014, the 3rd International Conference on Design, Automation, & Test in Europe. IEEE Computer Society. (2014)
6. Van Hentenryck, P.: Constraint Satisfaction in Logic Programming. MIT Press (1989)
7. Gotlieb, A.: TCAS Software Verification using Constraint Programming. The Knowledge Engineering Review **27**(3) (Sep. 2012) 343–360
8. Rossum, G.: Python reference manual. Technical report, Amsterdam, The Netherlands, The Netherlands (1995)
9. Francis, K., Brand, S., Stuckey, P.J.: Optimisation modelling for software developers. In: Principles and Practice of Constraint Programming, Springer (2012) 274–289
10. de la Banda, M.G., Stuckey, P.J., Van Hentenryck, P., Wallace, M.: The future of optimization technology. Constraints (2013) 1–13
11. Marriott, K., Stuckey, P.J.: Programming with constraints: an introduction. MIT press (1998)