Using CP in Automatic Test Generation for ABB Robotics' Paint Control System

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. Designing industrial robot systems for welding, painting, and assembly, is challenging because they are required to perform with high precision, speed, and endurance. ABB Robotics has specialized in building highly reliable and safe robotized paint systems based on an *integrated process control system*. However, current validation practices are primarily limited to manually designed test scenarios. A tricky part of this validation concerns testing the timing aspects of the control system, which is particularly challenging for paint robots that need to coordinate paint activation with the robot motion control.

To overcome these challenges, we have developed and deployed a cost-effective, automated test generation technique based on Constraint Programming, aimed at validating the timing behavior of the process control system. We designed a constraint optimization model in SICStus Prolog, using arithmetic and logic constraints including use of global constraints. This model has been integrated into a fully automated continuous integration environment, allowing the model to be solved on demand prior to test execution, which allows us to obtain the most optimal and diverse set of test scenarios for the present system configuration.

After three months of daily operational use of the constraint model in our testing process, we have collected data on its performance and bug finding capabilities. We report on these aspects, along with our experiences and the improvements gained by the new testing process.

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 continues to grow, developing and validating software for CIRs is becoming increasingly difficult. For robots performing process-intensive tasks such as

B. O'Sullivan (Ed.): CP 2014, LNCS 8656, pp. 25–41, 2014.

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 more reliable products 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.

In this paper, we report on our work to use 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. Using CP to generate model-based test scenario is not a completely new idea [4,5], but, according to our knowledge, this is 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 Robotic's current testing practices of the IPS and the rationale behind our validation choices ; Section 4 presents the CP model with its decision variables, test objectives and optimization principle ; Section 5 explains how the model and its solving process are implemented and included in the CI process ; Finally, Section 6 discusses some lessons learnt and summarizes the impact of using CP in ABB Robotics's industrial context.

Notation. Throughout the paper a constant in the CP model is prefixed with a *, as in $^*SeqLen$. This is typically a value set by a validation engineer or queried from the system under test prior to launching the model.

2 Robotized Painting

This section briefly introduces robotized painting, and highlights some of the challenges faced when testing such systems. 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.

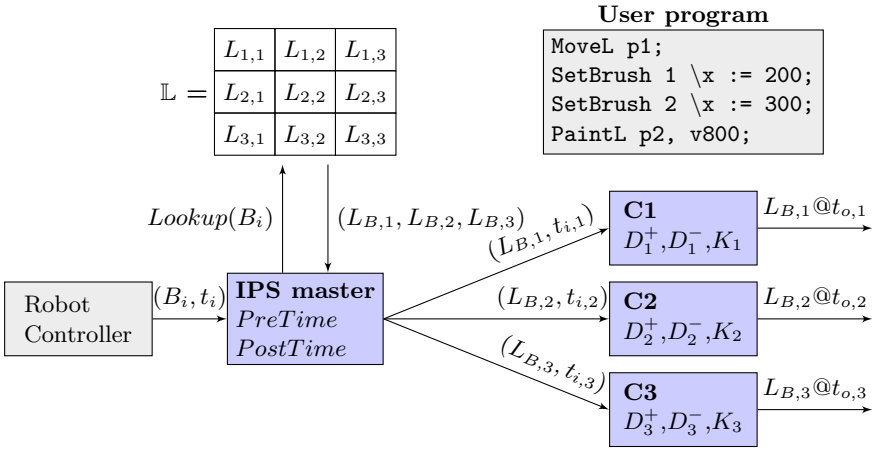


Fig. 1. Logical overview of a robot controller and the IPS

The program listing of Figure 1 shows an example 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` and activates the painting process. The `v800` argument of `PaintL` gives the speed of the movement (i.e., 800 mm/s).

Assuming 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 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 ≈ 50 ms for spray pattern 1, and at ≈ 175 ms for spray pattern 2. These messages simply convert 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.

Figure 1 shows three different actuator outputs (**C1, C2, C3**). The value of each actuator output for a given spray pattern is resolved by using a brush table ($\mathbb{L}$). 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 now passes these values to each actuator output along with its activation time, which may be different from the original time received from the robot controller. Possible modifications can be formalized as follows:

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

What equation (1) shows is that the activation time of each actuator output may be adjusted by a constant factor ($PreTime$, $PostTime$), 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**.

Activation of Actuator Outputs: Referring to Figure 1, we now present how messages are processed when sent from the IPS master to a single actuator output. Let us assume that message (L, t_i) is sent, and 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 one is to adjust the time with a *constant* factor: D^+ for positive change,

and D^- for negative change. The second one is using a *linear* timing function to adjust the change of the physical value.

Equation (2) combines these strategies into a single compensation function, where ${}^*Min$ (resp. ${}^*Max$) is the physical minimum (resp. maximum) value possibly handled[1] by some actuator output.

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

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 physical process it is responsible of.

3 Testing the IPS

Having a distributed control system such as the IPS mounted on a physical robot makes its validation unnecessarily complex, and current testing practices involve a considerable amount of manual work, including setup and collecting observations. If while developing a new version of the IPS software, test scenarios are only run when approaching the release date, then development costs can grow substantially, as correcting software defects late in the development process may require developers to dig into the early stages of development. Even worse, if a software failure is observed during operation (i.e., by the customer), costs become even higher since corrections may need to take place at the customer site.

3.1 Continuous Integration

CI is a software engineering practice aimed at uncovering software defects at the earliest stage of development, by regularly building the system and executing tests automatically [1].

A good engineering practice requires developers to submit only small source code changes frequently, instead of large sets of changes occasionally. Together with this practice, CI has been shown to be a very efficient way of uncovering defects when developers are geographically distributed or large teams are involved. Typically a CI infrastructure includes tools for source control repository, automated build servers, and testing engines.

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

3.2 Testing in a CI Environment

We have developed an automated testing framework for the IPS as an integrated part of ABB's CI environment, where we have used CP to generate both the configuration for the IPS, the test sequence, the brush table and the output of each actuator output and finally execute the test as part of a CI cycle.

Compared to traditional software testing, running a test scenario in a CI environment has additional requirements. In particular, as pointed out by Fowler [1], mastering the total *round-trip time* is crucial for a successful CI deployment. Here, round-trip time refers to the time it takes for a developer to submit a change to the source control repository and get feedback from the build and test processes. Thus in order to keep the round-trip time as small as possible, we have identified a few areas where special care must be taken:

- **Test complexity:** 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 so-called *good enough criterion*, frequently used in industry [6].
- **Solving time:** Constraint-based optimization is most often a time-consuming task, especially if a global optimum solution is sought [7]. Thus, when used in CI, it becomes imperative that a time-contracted optimization procedure be used. In other words, it is important to have precise control over the time needed to compute the optima, by sacrificing the solution quality.
- **Execution time:** We observe that test execution time is dependent on the length of the test sequence, i.e., the number of test scenarios. This must be accounted for, together with the time needed to generate the test sequence.

In essence, balancing between the length of a test sequence (its execution time) and the time needed to generate the test sequence (its solving time) is a way to find the appropriate trade-off to fully integrate CP into a CI process.

4 CP Model of the IPS

We now present our CP model for the IPS. We emphasize that test models, as proposed in model-based testing [8], are usually limited in their scope. They are not intended to reflect the full behavior of the system they represent. In our case, we confine ourselves to modeling the timing aspects of the IPS in order to build an efficient CP model for generating test scenarios.

4.1 Decision Variables and Domains

While still referring to Figure 1, we now assume that the number of actuator outputs is a constant input parameter C, instead of 3. The decision variables for our problem can be divided into three distinct groups: the variables of the input sequence $\mathbb{I}$, the configuration variables $\mathbb{C}$, and the variables of the brush table $\mathbb{L}$. In principle, a solution of the CP model is formed by an instantiation of these variables, in addition to the so-called test oracle $\mathbb{O}$, which is the expected output

computed by the system formed by each actuator output and its corresponding time.

Formally, the test input sequence $\mathbb{I}$ corresponds to $((B_1, t_1), \ldots, (B_N, t_N))$, where $N = {}^*SeqLen$ and each $B_i \in [0, {}^*BTabSize]$ and each $t_i \in [0, {}^*MaxTime]$. The configuration $\mathbb{C}$ contains parameter variables for each actuator output and for the IPS master:

$\mathbb{C} = [PreTime, PostTime, D_1^+, D_1^-, K_1, \ldots, D_{*C}^+, D_{*C}^-, K_{*C}]$

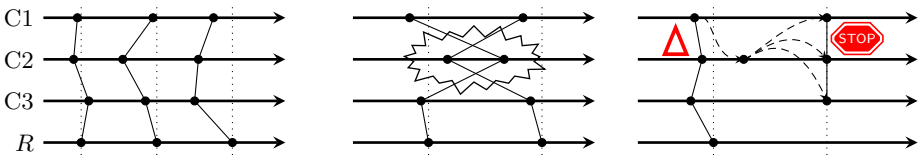
The domain of the variables in $\mathbb{C}$ is given by configurable constants to the CP model. In the brush table $\mathbb{L}$, the number of columns corresponds to the number of actuator outputs, i.e., ${}^*C$, and the number of rows is a constant ${}^*BTabSize$. The domain of each variable in $\mathbb{L}$ is extracted from ${}^*Min$ and ${}^*Max$ for the corresponding actuator output. The test oracle $\mathbb{O}$ corresponds to the physical output of each actuator output with its corresponding time. Each actuator output has output and time corresponding to a single input: $(B_i, t_i) \mapsto \big((L_{1,i}, t_{1,i}), \ldots, (L_{C,i}, t_{C,i})\big)$ for $i \in [1, N]$.

4.2 Test Scenarios

We have identified several distinct test scenarios, and we present three of them here, as shown in Figure 2. Scenarios *overlap* and *kill brush* represent failure conditions, where the IPS is forced into an error state. When generating such scenarios it is our interest to check whether the IPS can respond correctly (i.e., shutdown, error messages, etc.). On the contrary, the scenario *normal* represents acceptable behavior and they are targeted to check whether the IPS behaves as expected. Whenever the CP model is solved, a scenario is given as a test objective to the solver, and the solving process intends to find an assignment of variables that can drive the execution of the IPS in the corresponding scenario status. The *overlap* scenario is used throughout the paper, as it is the hardest to find and therefore, it corresponds to the most difficult objective to solve for the CP model. Let us explain it in more details. As explained in Section 2.1, the IPS can queue up a sequence of actuator output changes. However, a sequence spray pattern number sent to the IPS can cause one or several of the actuator outputs to come out of order with respect to time. This can be due to changes over time *between* spray patterns, or due to usage of $PreTime/PostTime$ configuration or else due to different configuration of the actuator output. In principle, the IPS must handle these issues by sending an appropriate error message to the control system.

4.3 Avoiding Trivial and Enforcing Diversity

An additional objective in test sequence generation for the IPS is to introduce diversity in the test input sequence $\big((B_1, t_1), \ldots, (B_i, t_i)\big)$, in the values of the brush table ($\mathbb{L}$) and in the configuration parameters for each actuator outputs (D^+, D^-, K). By diversity, we mean variations in the test scenarios so that the chances to discover an error-prone scenario are greater.



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

Fig. 2. Test scenarios considered as test objectives. Horizontal axis represent time and black dots correspond to output activation. A specific *spray pattern* is a collection of output activations, and is visualized by a line connecting the black dots.

As solving the CP model is a deterministic process, introducing diversity is a way to cope with the possible generation of useless scenarios. Let us consider the setup in Figure 1 where configuration $\mathbb{L} = \begin{smallmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{smallmatrix}$, $D_j^+ = 0$, $D_j^- = 0$ and $K_j = 0$, $j \in 1..3$ is given, we see that no matter what the input sequence $(\ldots,(B_i,t_i),\ldots,)$ is, the actuator output output is always $(0,t_i)$. This is of course a solution, but it has no practical interest, as it does not correspond to a possible behavior of the IPS. Using randomization in the CP model, in order to introduce diversity, has clearly been discarded. In fact, one of our initial requirements is to maintain a reproducible process. When testing the IPS, it is important to document failure cases and to help debug the system with the generated test scenarios.

4.3.1 Variation in $\mathbb{I}$ Values

Let $\mathbb{I} = ((B_1,t_1),\ldots,(B_N,t_N))$ be an input sequence to the IPS. Significant variation on values for t_i is not interesting, as the only requirement is that time must increase monotonically with a minimum step ($^*MinBrushSep$). More interesting is the variation in B_i. Obviously, enforcing a change in two successive brush element selections is important, i.e., $\forall i, B_i \neq B_{i+1}$, but this is not sufficient to guarantee that all indexes in the $\mathbb{L}$ are tested. Diversity on sequence $B = B_1,\ldots,B_N$ could be implemented by using global constraint `nvalue( LenLookupTab, B )`, enforcing that every index is present at least once. But, it will not be sufficient for our testing purposes.

Let us define the notion of *diversity entropy* (DE): given a sequence of integers, DE is the product of the number of occurrences of each value in the sequence. For example $DE([0,1,0,1,0,1,0,1,2,3]) = 4 \cdot 4 \cdot 1 \cdot 1 = 16$, while $DE([0,1,2,1,2,3,1,3,2,3]) = 1 \cdot 3 \cdot 3 \cdot 3 = 27$. With this example, we see that the first solution, respecting both previously mentioned constraints, has a diversity entropy lower than the second solution. We therefore come up with another solution in which we use the `global_cardinality` constraint. By specifying the minimum number of times an index of $\mathbb{L}$ must appear in the input sequence,

we increase the diversity entropy of the solution. For example, given ${}^*SeqLen = 10$, ${}^*BTabSize = 4$, variation in the input sequence is enforced by using

```
global_cardinality( [B1,...,B10], [1-N1,2-N2, 3-N3, 4-N4] )
N1 #>= Ob, N2 #>= Ob, N3 #>= Ob N4 #>= Ob
```

where `Ob` is a given constant input parameter of the CP model. This implementation is flexible enough to consider solutions with a satisfactory DE.

4.3.2 Variation in $\mathbb{L}$ Values

Variation in $\mathbb{L}$ is equally important. When the validation engineers create these tables manually, they try to enforce that each actuator outputs ${}^*Min$ and ${}^*Max$ is part of the $\mathbb{L}$, and that the whole operating area of the actuator output is used. If each entry in $\mathbb{L}$ is regarded as coordinates in an Euclidian space, $\mathbb{R}^C$, an approach could be to maximize the distance between each point, i.e., each entry in $\mathbb{L}$. However, we observed that this approach is too costly to compute in practice, and we prefer a more light-weight approach. For each row in $\mathbb{L}$, $(R_1, \ldots, R_{{}^*BTabSize})$, we exploit the global constraints $minimum({}^*Min_j, R_j)$ and $maximum({}^*Max_j, R_j)$ to enforce usage of extremal values. In addition, the `all_min_dist` [9] constraint is used to make sure the values are spread out.

To introduce variation *between* the entries in $\mathbb{L}$, additional constraints are used to enforce at least one transition where all except one value is changed. For example, from Figure 1, if two entries is $[L_{1,i}, L_{2,i}, L_{3,i}]$ and $[L_{1,j}, L_{2,j}, L_{3,j}]$ then there should exist i and j such that $L_{1,i} < L_{1,j} \wedge L_{2,i} \geq L_{2,j} \wedge L_{3,i} \geq L_{3,j}$, and so on other entries. This is clearly far away from maximizing the Euclidian distance between each entry, but this approach turns out to perform fairly well together with the scenarios presented earlier. Of course, there is room for improvement here in further work.

4.3.3 Variation in $\mathbb{C}$ Values

The generated configuration for a specific test scenario includes both the values for each actuator output (D^+, D^-, K) and the value for IPS master ($PreTime$, $PostTime$). In many setups, validation engineers select these values manually without questioning the error-proneness of a given configuration with respect to another. By adding simple constraints for each actuator output, such that $D^+ \neq D^- \wedge D^- \neq 0 \wedge D^+ \neq 0$, we offer an opportunity for the CP model to introduce diversity in the configuration values as well. By using global constraint `all_different` $(D_1^+, \ldots, D_C^+)$,etc, we also enforce diversity between actuator output values. For the $PreTime$ and $PostTime$ values, a similar strategy is employed: $PreTime \neq PostTime \wedge PreTime \neq 0 \wedge PostTime \neq 0$.

It is worth noticing that these variation strategies have served well the *good enough principle*, as introducing diversity is important but not at the cost of losing efficiency.

4.4 Search and Optimization

We now briefly present the optimization function and the search heuristics used in our model. In our framework, finding optimal solutions that respect the set

of above-mentioned constraints is the most interesting. Optimal solution here means a sequence of timed events $\mathbb{I} = ((B_1, t_1), \ldots, (B_N, t_N))$ of the *smallest execution time*, i.e., where t_N is minimized. By doing this, we increase the capabilities of the CI process to execute more tests during a limited period. Of course, reaching exactly the global minimum over t_N is interesting from an intellectual perspective, but not really necessary in our industrial setting.

As mentioned in Section 3.2, managing the time needed to generate and execute test sequences when running tests in a CI environment is of crucial importance. Considering a test sequence $\mathbb{I} = ((B_1, t_1), \ldots, (B_N, t_N))$, and the fact that each spray pattern is sent approximately 200 ms before execution, as explained in Section 2.1, we see that the execution time of the test can be roughly estimated to be t_N. This means that the total time used is roughly $t_s + t_N$, where t_s correspond to the solving time of the model.

Knowing that the constrained optimization model tends to minimize t_N, the goal is therefore to control the time needed to find an optimal solution. CP offers means to control the time taken to optimize by using a *branch-and- bound* procedure. That is, we can give a contract of time to this procedure, and it returns the current feasible solution after the contract of time has passed. We found this option very useful to compromise between the time spent on search and solving, and the time spent on execution of the test.

4.5 Search Heuristics

When searching for solutions, many heuristics can be employed or programmed in CP. Observing the absence of evident *structure* in our CP model, we have considered variable orderings as the first element to examine systematically. In order to extract useful information, we considered 72 distinct static variable orderings depending on rearrangements of the decision variables $(\mathbb{I}, \mathbb{L}, \mathbb{C})$. In addition to this systematic exploration, we took as a reference two well-known dynamic variable orderings, namely *first-fail* and *first-fail constraint* [3]. We also tested *up* and *down* which dictates the direction the domain is searched (ascending or descending). This analysis and the experiments revealed two points:

1. Even if first-fail and its variation are efficient for timed sequences containing few events, they quickly become unusable for larger sequences. This can be easily explained by the necessary computation of comparison between domain sizes during the search, which becomes intractable as soon as the number of variables grows.
2. Looking at search heuristics with static variable orderings, we can group the result into three groups: H_1, H_2 and H_3.
   $H_1 = (\mathbb{C}, \mathbb{L}, \mathbb{I}')$, $(\mathbb{L}, \mathbb{C}, \mathbb{I}')$ $(\mathbb{C}, \mathbb{L}^{\mathbb{T}}, \mathbb{I}')$,$(\mathbb{L}^{\mathbb{T}}, \mathbb{C}, \mathbb{I}')$, $H_2 = (\mathbb{C}, \mathbb{B}, \mathbb{L}, \mathbb{T})$, $(\mathbb{B}, \mathbb{C}, \mathbb{L}, \mathbb{T})$, $(\mathbb{B}, \mathbb{L}, \mathbb{C}, \mathbb{T})$, $(\mathbb{C}, \mathbb{B}, \mathbb{L}^{\mathbb{T}}, \mathbb{T})$, $(\mathbb{B}, \mathbb{C}, \mathbb{L}^{\mathbb{T}}, \mathbb{T})$, $(\mathbb{B}, \mathbb{L}^{\mathbb{T}}, \mathbb{C}, \mathbb{T})$, H_3 = The other 62 tested combinations, where $\mathbb{I}' = (t_1, B_1, t_2, B_2, \ldots)$, $\mathbb{B} = (B_1, B_2, \ldots)$, $\mathbb{T} = (t_1, t_2, \ldots)$ and $\mathbb{L}^{\mathbb{T}} = transp(\mathbb{L})$. H_1 is the only heuristics able to produce a solution within an acceptable timeframe for small values of ${}^*BTabSize$ combined with large values of ${}^*SeqLen$, e.g. ${}^*BTabSize = 10, {}^*SeqLen = 200$.

For configurations of large values of ${}^*SeqLen$ combined with large values of ${}^*BTabSize$, e.g. ${}^*BTabSize = 40, {}^*SeqLen = 600$, H_2 is the only heuristic able to generate a solution within reasonable time. The result for H_3 is either no solution at all, or only solution for small ${}^*BTabSize$ and small ${}^*SeqLen$. Understanding precisely why H_1 and H_2 performs so well is part of our planned further work.

5 Implementation and Exploitation

This section details our implementation of the CP model [10] with SICStus Prolog and its `clpfd` library [3], and its exploitation in the CI process at ABB Robotics. It also gives some insights on the rationale behind the selection of CP instead of other possible techniques.

5.1 Selection of CP and the CP Solver

The mathematical model of the IPS could have been implemented with other techniques than CP, including SAT- or SMT-solving [11], local search techniques for test data generation [12], or Mixed Integer Programming (MIP) [13]. We briefly review the reasons why these other techniques have been discarded[2]:

1. The selected technique had to be flexible enough to accommodate the many alternatives in the dynamic configuration of the IPS. CP offers a higher degree of flexibility to handle disjunctive constraint systems, by authorizing the usage of *backtracking*, *reification*, or *constructive disjunction* [14] ;
2. Time-constrained optimization was essential in our industrial context in order to accommodate with the CI process. SAT- and SMT-solving are very efficient to handle boolean and theory-based satisfiability problems [11], but they are not tuned to solve optimization problems (i.e., to minimize a cost function in a given contract of time). Even if extensions exist to handle optimization problems, classical off-the-shelf SMT-solvers do not provide implementations of these extensions. On the contrary, CP integrates time-aware optimization methods on discrete combinatorial problems ;
3. As the model is used to predict the expected outputs of the IPS, using exact methods was mandatory. Despite the efficiency of local search techniques for test data generation [12], the absence of guarantee on the satisfiability of the constraints (e.g., no possible detection of unsatisfiability or no guarantee on the determination of satisfiability for complex constraint sets) was sufficient to discard these techniques ;
4. Input formats of the constraint solver had to be easily tunable to accommodate the high-level tuning of IPS parametrization. SAT- and SMT-solvers takes specific formats as inputs (e.g., SMTLIB formats) while CP-solvers are usually hosted by a programming language (e.g., Prolog, Java or C++) which includes high-level programming features such as predicate/method invocation, recursivity, inheritance, and so on ;

[2] Note that no general claim is made, just specific claims to illuminate our choice of CP in the case of validating the IPS.

5. The availability of global constraints to implement diversity in test sequences was a strong advantage, even if, to be honest, we discovered it after our choice was made.

We found that SICStus 4.2.3 in combination with `clpfd` responded well to our industrial requirements and decided to use it as back-end, and Python 2.7 as front-end.

5.2 Overall Implementation

The complete system contains around 2k lines of Prolog code, 300 lines of C code (an interface DLL between Python and SICStus), and finally around 3k lines of Python code. A schematic overview of the implementation and how it is executed can be found in Figure 3.

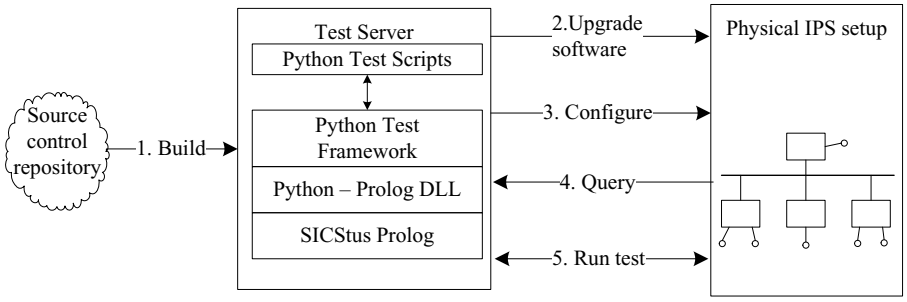


Fig. 3. Integration between the test server and IPS

The modeling part of the project has started early in 2013 ; at the beginning, just by using the user interface of SICStus. In April 2013, a first running model was available on a desktop for testing IPS, running over a single embedded board. In May, the model was integrated into the source control repository and the first automatic test running in a full CI environment was executed. From May to October 2013, the system was further extended to also cover testing over complete distributed systems (i.e., several embedded boards) of the IPS. Today, the model is used in the CI process and solved daily. It generates test sequences for 11 different physical embedded IPS boards. For testing on the full-distributed setting, we currently run the model on one single physical setup, but we run 10 different configurations on this setup. To summarize, the number of measurable activations of physical actuator outputs shows that around 20.000 distinct test scenarios are executed during each individual CI cycle. It means that these test scenarios are executed at least once every 24 hours.

5.3 Execution of the Model

Test execution is typically triggered by a build server upon a successful build of the IPS software. These steps are illustrated in Figure 3 and explained below.

1. **Build:** Building the software is scheduled to run every night, or a developer can trigger a build manually.
2. **Upgrade:** Upgrade all connected embedded controllers with the newly built software. A failure in this step aborts the complete cycle.
3. **Configure:** Configuration fetched from the source control repository is loaded onto the IPS. The configuration describes the interconnections of embedded boards and the properties of this specific paint-setup.
4. **Query and Solve Model:** Data retrieved from the IPS is fed into the CP model. This enables us to keep the generated test in sync with changes in the newly built software or changes in the configuration.
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*, $\mathbb{O}$.

5.4 Using the Flexibility of CP

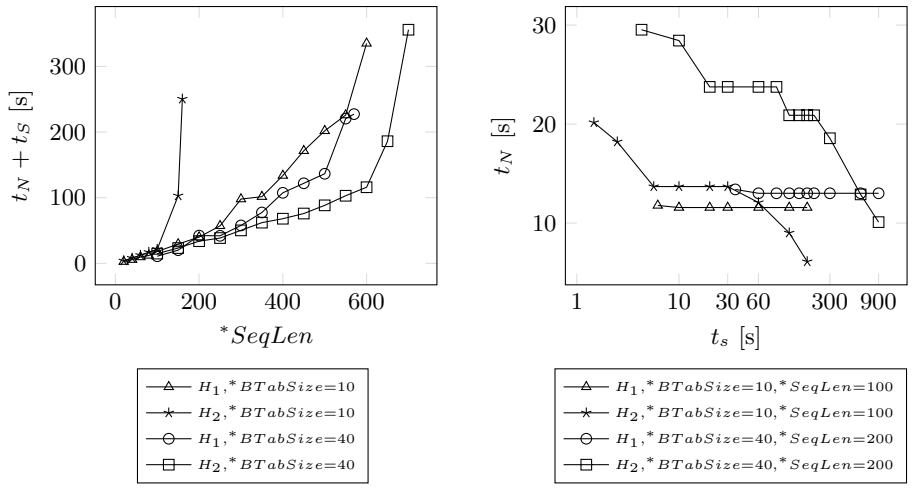
As described in the previous sections, we have designed the model to be flexible enough and to be able to generate realistic test sequences. In particular, introducing diversity by applying global constraints between variables has been a key factor for satisfying our industrial requirements. However, the CP model can also handle specific parameter values, directly given by the validation engineers not having a strong knowledge of CP. This is simply implemented by guarding the posting of each constraint with some groundness conditions. For example, using `(var(X), var(Y)) -> X \#= Y ; true` to guard the posting of `X \#= Y`. Thanks to the Prolog commodity, our Python front-end can give value to *any* variable in the model and avoid posting spurious constraints that would slow down the solving process, or prevent a solution.

5.5 Performance of Model

Recalling that H_1 and H_2 represent two different groups of variable orderings with similar performance, Figure 4a compares the total time for a test execution $(t_s + t_N)$ for H_1 and H_2 with two different sizes of ${}^*BTabSize$.

When used only to find a solution to the constraints (i.e., without optimization), H_1 gives better results for ${}^*BTabSize = 10$, while H_2 performs better for ${}^*BTabSize = 40$. This experiment revealed 1) that H_1 usually produces a relative small value for t_N, by using more time than H_2 and 2) that H_2 usually produces a larger value for t_N but faster than H_1.

We also compared H_1 and H_2 when minimizing the overall time of a test sequence, i.e. $minimize(t_N)$. Figure 4b shows that H_2 provides the first solution faster than H_1, and that the quality of solution is better when more time is



(a) Total time ($t_s + t_n$) (b) Optimization, $minimize(t_N)$

allocated to the search for optimality. For H_1, Figure 4b shows that there is no gain in minimizing t_N.

For the setup $^*SeqLen = 100, ^*BTabSize = 10$ we see that the solver must run for ≈ 60 s before H_2 gives a smaller t_N than for H_1. For the setup $^*SeqLen = 200, ^*BTabSize = 40$ the solver must run for ≈ 600 s before a break-even occurs.

From the results on Figure 4a and Figure 4b, no strong conclusion can be drawn when it comes to select between H_1 and H_2. If a test sequence is generated for multiple uses, i.e. reusing the same test sequence multiple times, then using H_2 is beneficial at the price of allocating more time to the optimization procedure. On the contrary, if a single usage is targeted, as is in the CI process, then using H_1 should be preferred by considering than the total time $t_s + t_N$ is the actual target of our test generation and execution procedure. Consequently, at ABB Robotics, we decided to keep the choice between these two heuristics as an option in our CP model. From a practical point of view, it permits the validation engineers to tune the test generation process according to their needs.

6 Lessons Learned and Conclusions

This section concludes the paper by presenting some lessons learned at ABB Robotics from our experience with introducing CP in our CI process.

6.1 CP for Validation Engineers

As previously stated, validation of robotized painting involves a fair amount manual, labour-intensive work. Therefore, replacing parts of this validation process with automation is necessary, and is perceived by validation engineers as a means to strengthen the process. However, it also comes with some drawbacks.

Two factors must be distinguished: (1) **The automation through the CI process** including automatic building of software, software upgrade, test execution and results reporting, and (2) **Test generation through use of CP**, which permits validation engineers to focus on validating other parts of the CIRs.

Point (1) does not have any drawbacks except the effort required to set up the CI process. From an industrial perspective, point (2) is the most critical, especially because (a) validation engineers are not yet sufficiently trained in CP, to change the model without help ; (b) validation engineers are usually reluctant to trust any tool that produces results, that are very difficult to compute by hand or with an easily understandable process. It is also recognized [15,16] as a concern that many optimization problems require expert knowledge. In order to reduce the risks, we decided to build a Python front-end to our CP model, so that some details can be hidden from the validation engineers. We also organized basic training in CP with simple and understandable examples in order to facilitate the adoption. Of course, we do not claim that these actions form a recipe for adopting CP in general, but we observe that it worked well in the context of ABB Robotics IPS validation.

6.2 Actual Defects Found with the CP Model

After the model was put into production at ABB Robotics, it immediately detected two new unknown defects related to timing aspects of IPS. These defects were however classified as non-critical, as they correspond to very unlikely scenarios. Digging into the causes of these defects, we saw that they had been present in the IPS for several years without any significant consequences and that they had been spotted by the CP model through enforcing diversity in the selection of test sequences. These defects were corrected and the test sequences used for spotting them were introduced into our non-regression test suite.

For validating the CP model, we also reintroduced five old, historical, defects into the source control repository. These defects were known by the validation engineers to be extremely hard to find. After a round of experiments, the CP model produced test sequences that spotted all the five defects. This was considered as a strong justification for the continued use of the CP model in production.

6.3 Return on Investment with the Use of CP

Computing the ROI for the use of CP for ABB Robotics' IPS validation is not easy. Possibly, one can measure the number of defects found with and without the CP model during the validation of a new IPS release. It is also possible to compare the human effort required in both cases. However, another important factor is the increased confidence of the engineers to the validation process, which is a factor that is very difficult to measure. After the introduction of the CP model in production, we observed a much higher confidence among the engineers to the testing framework and their appetite to perform necessary code re-factoring is now higher. They are more willing to make critical, but needed,

changes in the software and they rely on the test framework to detect undesired side-effects. If a side-effect is discovered, they can simply roll back the change.

In the long term, we expect to see the benefits of using CP being recognized as a way to increase the general quality of the testing process, since necessary re-factoring will be performed before the technical depth grows beyond control.

6.4 Further Work

In the previous section, we mentioned at least two main points to dig into, in order to get a better understanding of the benefits of the CP model in ABB Robotics' IPS validation. Firstly, as introducing diversity in the selection of test sequences is crucial in our application, more dedicated global constraints could be built to capture the needs of validation engineers. In particular, constraining the variables of the brush table to take balanced values is highly desirable. Secondly, a deep understanding of the reasons why our heuristics H_1 and H_2 perform significantly better than other variable ordering choices would help us improving the constraint model by refining constraint posting.

Acknowledgements. We thank the anonymous reviewers for their comments. This work has been realized with the partial support of the Research Council of Norway, as part of the Industrial PhD of Morten Mossige, ABB Robotics and Certus SFI - Project 8.

References

1. Fowler, M., Foemmel, M.: Continuous integration (2006) (accessed August 13, 2013)
2. Mossige, M., Gotlieb, A., Meling, H.: Poster: Test generation for robotized paint systems using constraint programming in a continuous integration environment. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST), pp. 489–490 (2013)
3. Carlsson, M., Ottosson, G., Carlson, B.: An open-ended finite domain constraint solver. In: Glaser, H., Hartel, P., Kuchen, H. (eds.) PLILP 1997. LNCS, vol. 1292, pp. 191–206. Springer, Heidelberg (1997)
4. Di Alesio, S., Nejati, S., Briand, L., Gotlieb, A.: Stress testing of task deadlines: A constraint programming approach. In: 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE), pp. 158–167. IEEE (2013)
5. Balck, K., Grinchtein, O., Pearson, J.: Model-based protocol log generation for testing a telecommunication test harness using CLP. In: Design, Automation and Test in Europe Conference and Exhibition (DATE), pp. 1–4 (2014)
6. Stolberg, S.: Enabling agile testing through continuous integration. In: Agile Conference, AGILE 2009, pp. 369–374. IEEE (2009)
7. Marriott, K., Stuckey, P.J.: Programming with constraints: an introduction. MIT Press (1998)
8. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann Publishers Inc., San Francisco (2007)

9. Régin, J.C.: The global minimum distance constraint. Technical report, Technical report, ILOG (1997)
10. Mossige, M.: Prolog Model of ABB's Paint Control System for test case generation (2014), http://www.ux.uis.no/~mortenm/ips/trigdev_bt.pl
11. de Moura, L., Bjørner, N.S.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
12. McMinn, P.: Search-based software test data generation: A survey. Software Testing, Verification and Reliability 14, 105–156 (2004)
13. IBM, ILOG Labs, I.: IBM CPLEX: High-performance software for mathematical programming and optimization (2006), http://www.ilog.com/products/cplex/
14. Rossi, F., Beek, P.V., Walsh, T.: Handbook of Constraint Programming (Foundations of Artificial Intelligence). Elsevier Science Inc., New York (2006)
15. de la Banda, M.G., Stuckey, P.J., Van Hentenryck, P., Wallace, M.: The future of optimization technology. Constraints, 1–13 (2013)
16. Francis, K., Brand, S., Stuckey, P.: Optimisation modelling for software developers. In: Milano, M. (ed.) CP 2012. LNCS, vol. 7514, pp. 274–289. Springer, Heidelberg (2012)