Testing Robotics Software using Constraint Programming in a Continuous Integration Process

by

Morten Mossige

A dissertation submitted in partial satisfaction of the requirements for the degree PHILOSOPHIAE DOCTOR (PhD)

University of Stavanger

Faculty of Science and Technology
Department of Electrical Engineering and Computer Science
2015
Abstract

Testing complex integrated robots (CIRs) requires testing several interacting control systems. This task 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.

Current practices for validating CIRs involve manual test case design and execution. To reduce testing costs and improve quality assurance, one trend has been to automate the generation of test cases and execute the test case automatically as part of a continuous integration process.

This thesis makes two main contributions. First, we present a methodology for the fully automated testing of CIR control systems. Our approach 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. We call the methodology CATS, which is short for Constraint-based Automatic Testing of IPS, where IPS is an abbreviation for ABB Robotics’ integrated paint control system.

Second, we present TC-Sched, a cost-effective method for automatic test case execution scheduling on multiple machines with constraints on accessible resources, such as measurement devices or network equipment. TC-Sched is also based on a constraint-based model and is designed to be integrated with and executed as part of continuous integration process.

The combinations of CATS and TC-Sched represent an efficient method for quickly validating the critical software components of CIRs. We first use CATS to automatically generate test cases. We then use TC-Sched to optimally schedule the test cases. We show that, when operating in a continuous integration process, there is a trade-off between the time spent solving the constraint model and the time spent executing the result of the model.

To evaluate our approach, we integrated CATS within ABB Robotics’ continuous integration process. A full lab for the automatic testing of actual embedded control systems was built. The first version of the model was introduced two years ago and has been extended several times. For TC-Sched, the method was tested on several real industrial test suites, in addition to a large set of randomly generated test suites. The results are promising and ABB Robotics has decided to implement TC-Sched in a full-scale setting at its research facilities at Bryne.
In conclusion, the research presented in this thesis shows that solving constraint programming models as part of a continuous integration process can generate realistic test cases and efficient test execution schedules. The research also shows that the solving process can be performed so that the trade-off between solving time and execution time is optimal.
Acknowledgments

First and foremost, I would like to thank my supervisors Arnaud Gotlieb and Hein Meling. Their insightful guidance and support was invaluable. Although they are from two completely different scientific domains, together they made up a powerful team. In particular, their scientific and practical advice helped me develop my analytical skills and improve my scientific presentation skills. I learned from them how to conduct research and I am grateful for their high work standards, kindness, friendliness, and sincerity.

This thesis would not have been possible without the support I received from ABB, where I have worked for the last 18 years. Since the very first day I raised the idea of pursuing an industrial Ph.D., I received incredible support. However, there is a big difference between supporting an idea and actually committing to the obligations involved. ABB Robotics not only supported me financially but also shielded me from daily operations, enabling me to focus on the research.

In this regard, I would first like to thank Jan Christian Kerlefsen. He supported the idea from day one and really fought hard for this project when it came to either financial matters or internal politics. I also want to thank Svein Seglem, who was the manager at ABB Robotics Bryne when the idea was sown. He signed the first contracts and committed to the project from ABB’s side. I must also thank my colleagues at ABB Robotics Bryne, especially Nina, Øysten, and Tore, who were of great help and good partners when discussing the intersection between research and industry. Finally, I want to thank ABB Robotics, Bryne, in general. It has to be one of the best places to work. If you have great ideas, you will get the opportunity to execute them at ABB Robotics Bryne. There is really no limit to what you can achieve when working at ABB Robotics, Bryne.

I would also like to thank Simula Research Laboratory, with special thanks to the members of Certus. Being part of that environment has given me the opportunity to gain insight into different industrial domains and their ways of attacking software engineering and software testing problems. I was also introduced to other industrial partners and was given insight into their handling of software testing problems. The cooperation I enjoyed with Cisco was especially valuable.
I would also like to thank my colleagues at the University of Stavanger. My work at the University of Stavanger one to two days a week provided an academic counterweight to my usual industrial environment.

During the last part of my research, I was introduced to Mats Carlson, one of the pioneers of constraint programming. This encounter led to further cooperation around my research, for which I am truly grateful. It has been a veritable pleasure to work together with such a leader in the field.

Last but not least, I would like to thank my family and friends.

Morten Mossige, August 2015
Contents

List of Papers .......................................................... ix

Summary .............................................................. 1

1 Introduction ......................................................... 1
2 Background ......................................................... 6
3 Main Contributions ................................................ 17
4 Research Method ................................................... 23
5 Summary of Results ............................................... 27
6 Further Work and Lessons Learned .......................... 31
7 Conclusion .......................................................... 35

Paper 1: Testing Robot Controllers Using Constraint Pro-
gramming and Continuous Integration ..................................... 43

1 Introduction ......................................................... 44
2 Background and Related Work ...................................... 47
3 ABB’s Process Control System ...................................... 50
4 Legacy Test Practices ............................................... 54
5 Modeling the IPS .................................................... 57
6 Test Scenarios with Constraints .................................... 61
7 Implementation ....................................................... 65
8 Empirical Evaluation ............................................... 71
9 Conclusions .......................................................... 87
Appendix ................................................................... 88

Paper 2: Using Constraint Programming to Test Paint
Control System for Robots .................................................. 95

1 Introduction ......................................................... 96
2 Robotized Painting .................................................... 98
3 Testing the IPS ....................................................... 103
4 CP Model of the IPS ................................................ 105
List of Papers

The following papers are included in this thesis:

• **Paper 1**

  Testing Robot Controllers Using Constraint Programming and Continuous Integration.
  M. Mossige, A. Gotlieb, H. Meling
  Published in the journal of Information and Software Technology, 2015

• **Paper 2**

  Using Constraint Programming to Test Paint Control System for Robots
  M. Mossige, A. Gotlieb, H. Meling
  Submitted to the Constraints journal.

• **Paper 3**

  Optimal Test Execution Scheduling on Multiple Machines with Resource Constraints
  M. Mossige, A. Gotlieb, H. Meling, M. Carlson
  Submitted to ASE 2015.

I was the main contributor for the idea, implementation, integration, and application in all the above-mentioned papers, but my supervisors were involved throughout all phases of the work.
For **paper 1**, I was the main contributor in the design and implementation of the constraint models. I also carried out the full integration of the constraint models into ABB’s continuous integration environment and performed the experiments.

For **paper 2**, I was the main contributor in the design of the constraint model, its full integration into ABB Robotics’ continuous integration environment, and the design and performance of the experiments.

For **paper 3**, I was the main contributor in the design of the constraint model. I was also the main contributor in the design and performance of the experiments and the generation of the random test suites.

**Papers not included in the thesis.** In addition to the papers included in this thesis, I published three others. These publications were not included because they overlap with the journal papers included, which are extensions of the conference papers listed below. **Paper 1**, which is included, is an extension and combination of **paper 4** and **paper 5**, and **paper 2** also included, is an extension of **paper 6**.

**Paper 6** won the **Best Application Paper** award at the 20th International Conference on Principles and Practice of Constraint Programming (CP) in 2014. We were subsequently invited to write an extended version of the paper, which was recently submitted to Springer’s journal Constraints.


Summary

1 Introduction

This section starts by presenting background information for the research performed as part of this thesis including an introduction to robot control systems. It also presents the industrial settings under which this research was performed and argues for the direction the research has taken during the work.

This thesis concerns the field of software engineering under the topic of software validation and testing. Its main topic focuses on challenges related to software development for robot control systems. The research performed in this thesis focuses on the development and solving of constraint programming (CP) models as a tool for improving the quality of software testing for robot control systems as part of a continuous integration (CI) process.

1.1 Challenges in the Development of Robot Control Systems

In the early days of industrial robots, individual robots existed in a standalone setting; that is, they were not connected together in larger groups of robots or to an overall control system. Furthermore, the robot control system’s internal design was based on a single standalone single-core computer. However, as industrial robots evolved, connecting each individual robot to external systems quickly became a requirement. Such connections were, in the beginning, very primitive. An example is the connection from a robot to an external control system by means of setting or reading a digital signal, where each digital signal has a specific meaning. An input signal could mean start paint job, while another output signal could mean paint job finished.
Today, customers of industrial robots require even more features. They need to read the robot’s current status and affect its behavior. A general requirement is the ability to analyze the internals of the robot and to control the robot in a more finely granulated manner than ever before. This is one of the reasons why robot manufacturers have experienced a shift in robot control system design technology, from a single-core computer setup controlling all aspects of a robot to today’s robot controllers, which are networks of computers, where each computer controls separate parts of the robot: one computer controlling the man-machine interface, one controlling the robot’s physical input and output, one to execute the user program and run the interpolator, and so forth. This has led to what we define today as complex industrial robots (CIRs).

Separation of the robot’s control system into multiple interconnected computers has led to large improvements in the flexibility of control systems, including reductions in cabling and the ability to bring the computers closer to the physical process. However, this development has also led to great challenges in testing such systems. Unlike the early days of having a single application binary for a robot, we now have more than 30 applications for the most complex settings. During testing, we need to make sure that the complete system works as expected while still maintaining the flexibility to modify only parts of the control system. To achieve this goal, an automated solution is needed to support the development process of CIRs.

This thesis addresses two specific tasks in which we propose a fully automated solution to support the development of CIRs. The first task relates to modeling a robot’s paint control system. With such a model, we can extract automatic test cases in which timed event sequences related to the synchronization of the robot’s physical motion path with the activation of a spray painting process. The second task relates to the automatic scheduling of test cases on parallel execution on multiple machines.

1.2 Industrial Setting

This thesis was written as part of the industrial Ph.D. program offered by the Research Council of Norway. The research was funded by ABB Robotics and the Research Council of Norway. The main part of the research was performed at ABB Robotics, Bryne, Norway. The remainder of the research was performed at the University of Stavanger, Norway, and at Simula Research Laboratory, Oslo, Norway. The thesis and research
project is mainly focused on applied industrial research. The following summarizes the areas prioritized during the research:

- The performance of high quality industry-driven research, mainly applicable to the automatic testing of ABB Robotics’ paint robots.

- The performance of realistic experiments and the development of tools integrated in ABB’s current software development processes and infrastructure.

- A strong focus on completing the research project on time and on budget.

*The performance of realistic experiments and the development of tools integrated in ABB’s current software development processes and infrastructure involved a significant amount of effort beyond what is reflected in the included papers. This work can be summarized as follows:*

![Figure 1: Software test lab built as part of the research.](image)
1. INTRODUCTION

SUMMARY

• Construction of a full-scale automatic software test lab, as shown in Figure 1.

• Restructuring of the current build tools used at ABB Robotics to enable model-based testing [54].

• Extensive refactoring of ABB’s in-house testing framework, which is based on Python [46].

• A preference for high-quality tools integrated into ABB’s production and testing rather than experimental prototype tools, such that the tools and models developed during the research were integrated into ABB’s current testing platform as much as possible and built to production-quality standards.

1.3 Collaborations

Through cooperation with my supervisors, Hein Meling (University of Stavanger) and Arnaud Gotlieb (Simula Research Laboratory), new doors were opened. Through Hein Meling’s connections, we established contact with Lyse and Altibox, among the largest energy and Internet providers, respectively, in Norway. Further, Arnaud Gotlieb introduced ABB Robotics to the Certus consortium. The Certus consortium is a group of Norwegian companies together with Simula Research Laboratory created for long-term perspectives and a strong industrial profile and whose main focus is to lead in software verification and validation research. ABB Robotics became a full-fledged member of the Certus consortium in 2013. Through Certus, we established valuable connections and collaborative relations with other companies that also had a strong focus on software testing and validation. Our cooperation with Cisco especially added a new dimension to the research in this thesis. Cisco provided us with background data and served as a valuable partner with whom to discuss solutions when we needed a counterweight to solutions provided by the research community.

1.4 Contributions

This thesis focuses on the generation and scheduling of test cases as part of a CI cycle by using CP. In particular, the contributions can be summarized as follows.
SUMMARY 1. INTRODUCTION

(i) We modeled the timing aspects of ABB Robotics’ integrated paint control system (IPS) as a CP model based on a finite domain. Our aim was to capture the timing of the process related to the activation of physical actuator outputs along a robot’s moving path. The model can reflect both the correct behavior of the IPS as well as its behavior in various error scenarios. The model is highly disjunctive and highly configurable. This means that the model has many solutions, where some of the solutions are better with respect to test case execution time. The model underwent extensive testing to find the best search heuristics to solve the CP model. We find two different heuristics, each with its own strengths and weaknesses: Heuristic 1 has a fast generation time while the test case’s execution time is longer. Heuristic 2 is the opposite: Its generation time is longer and the execution time is shorter compared with heuristic 1 (paper 2).

(ii) Based on the CP model of the IPS, we built a system for the automatic test case generation of the IPS. Given a test objective, it is capable of generating the configuration parameters for the IPS, the actual timed event test sequences, and the Oracle describing the expected behavior. The model was fully integrated into ABB Robotics’ CI process. Test cases were generated and executed as part of the CI cycle. This was the first time such complex solving processes had been both solved and executed as part of a CI process. The model was shown capable of both detecting completely new errors and detecting old, reintroduced errors (paper 2).

(iii) We developed a methodology for test case execution scheduling based on scheduling techniques from the CP community. The goal of the scheduling model was to generate a schedule that executes all test cases in a single test suite. Each test case can be executed on one or several machines and the scheduler selects which of them to use for execution. The scheduler also handles resource constraints between test cases, which can prevent two test cases from being executed simultaneously, even if executed on two distinct machines. The scheduling model is designed for integration into a CI cycle, where the schedule is calculated at execution time. This setup makes it possible to add or remove both test cases and execution machines dynamically without interfering with the test case execution (paper 3).
1.5 Thesis Structure

This thesis is a collection of papers and is organized into two parts, as follows.

- **Summary:** This part presents the research conducted in relation to this thesis and introduces the papers included. Section 2 presents background information on the main concepts. Section 3 explains the core ideas, including how the contributions fit together within a broader setting. Section 4 describes the research methods and Section 5 summarizes the main results of each paper. Finally, Section 6 discusses future directions for this research before Section 7 concludes the thesis.

- **Papers:** The second part of this thesis presents the research papers included that have been published or submitted.

2 Background

Section 2.1 starts by stating some of the common problems in developing large-scale software systems designed to run on multiple hardware platforms and multiple operating systems, such as robot control systems. Section 2.2 presents the basics of CI, including how testing in a CI process is typically performed and some of its challenges. Section 2.3 discusses CI in general and provides a short introduction to its use in solving scheduling problems. Finally, Section 2.4 provides an introduction to industrial robots and highlights some challenges related to testing industrial robot systems.

2.1 Challenges in Multi-Platform/Multi-OS Systems

In today’s industry, software is a valuable asset. When a software component is written, its subsequent reuse is usually desirable, since the cost of rewriting it is very high. An example of such a long-living software product is ABB’s robot controller and the Linux operating system. Although both have been through long phases of development and restructuring, they still retain some of their initial software components.

Maintaining such long-living software components can be challenging, especially when a component is designed to be executable on many different hardware platforms and operating systems. For example, ABB Robotics’
IPS has software components that are compiled on more than six different hardware architectures (e.g., x86, MPC5200, P1010, MC68223, C5400, C5500, F2812) and more than four different operating systems (e.g., Windows XP, Windows CE, Linux, VxWorks). This means that, when a software component is changed because of either a bug fix or an extension, extensive testing is required to verify that the software component conforms to the requirements of all the hardware architecture-operating system combinations.

A number of tools can help the developer to perform such verification and testing, such as static code analyzers, automatic code inspection tools, and model checkers. However, these are usually insufficient. To obtain better confidence about changing a component, the de facto industry standard is to fully integrate it into the end product and execute tests on actual hardware. With many combinations of hardware and operating systems, this task quickly becomes infeasible if carried out manually. When such tasks are done manually, the time since the change is tracked in the source code and the next time the component is tested can be long enough to prevent the developer from remembering it clearly. It will also be very difficult to track which change caused which error, since integration and testing are potentially conducted over several changes by several developers.

However, solutions to this problem exist. By automating all the task involved in building the software, performing the integrations, deploying it to actual hardware, and executing tests, it is possible to drastically reduce the time from a component’s change until it is verified on actual hardware. The following sections discuss some of the techniques used and the challenges of using them on industrial robots.

2.2 Continuous Integration

Continuous integration [16] is a software engineering practice aimed at uncovering software errors at an early stage of software development, to avoid problems during integration testing. A typical CI infrastructure includes source control repository tools, automated builds, build servers, and test servers. A build server is a machine that fetches source code from the source control repository and carried out the building, testing, integration, and so forth. All steps are carried out completely automatically and are typically triggered by a source code commit or a timer. A typical example of such a process is shown in Figure 2. Fitzgerald and Stol
describe CI 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 a common understanding that the time from a CI cycle being triggered to the developer receiving feedback should be as short as possible [13, 13]. Furthermore, one of the key ideas behind CI is to build, integrate, and test the software as frequently as possible and keep the cycle as short as possible. Developers working under CI are encouraged to submit small source code changes to the source code repository instead of infrequent, large chunks.

![Diagram](image-url)

**Figure 2:** Continuous Integration as currently performed in many industrial settings.
2.2.1 Testing as Part of Continuous Integration

In its simplest form, testing as part of CI executes a static collection of test suites containing unit tests \cite{55} and the unit test is executed on the build/test machine rather than on the end target hardware platform. This stage is usually very quick and will reveal a large group of potential errors in the software component under test. However, since the unit test is not being executed on the actual end platform, errors related to the target hardware architecture are not detected. An extension would therefore be to execute each unit test on the actual hardware platform. This setup would, of course, potentially detect more errors but would require a more expensive and complicated infrastructure where the actual end hardware platforms must be installed.

The next step, in terms of better testing as part of CI, is to execute integration tests on the actual end hardware platform in addition to running the unit tests as already described. Integration tests are typically performed on the end product, into which the full end product software has been integrated. Such integration tests are typically a set of static, fixed test cases and the test suite of integration tests is typically kept static over a longer time period. A typical integration test for an industrial robot system would be to reboot the robot’s control system and make sure that the mechanical arm can perform basic movements. This will, of course, require placing the robot in a safe area to avoid human injury.

Both of the above-presented testing techniques executed as part of CI use one or more test suites of fixed test cases. This means that exactly the same test suite is executed for every test campaign. In many contexts, this is sufficient; however, a more advanced approach is needed when the test suites are large and the time allocated for test execution is limited.

The approach of having a known, fixed set of test suites executed in a CI process has major drawbacks, which can be summarized as follows.

- There is no strong link between a change in a software component and the corresponding test case. For example, if a feature is removed from a software component, the corresponding test case will also need to be updated. This can lead to false errors due to a mismatch between the behavior of the test case and that of the software component. The task of detecting this deviation and potentially modifying the test case is a manual one.
2. BACKGROUND

SUMMARY

- Changing a test suite to accommodate a new test case prioritization or new test suite optimization, for example, is typically a manual job, performed outside of the CI environment. The result of the change (the new test suite or the new schedule) then needs to be manually transferred into the CI environment.

- If the time allocated for testing in the CI cycle is less than the time it takes to run the tests in the test suite, a manual selection needs to be undertaken that could leave out important test cases.

Several approaches could, in principle, be integrated and performed as part of the CI cycle, such as test case generation, test suite minimization, and prioritization \cite{7, 12, 21, 6, 31}. There are, however, no reported cases of such complex processes or their equivalent having been integrated and executed as part of the CI process. The only example we have found involves the inclusion of system execution modeling tools as part of CI in testing distributed real-time systems Hill et al. \cite{22}. This approach, however, does not work on the end product itself but, rather, on a simulated model of the system.

In the work presented in this thesis, we incorporate systematic automated test case generation in CI in addition to presenting a solution for automated test case scheduling. This is a first step toward greater automation in the software validation of complex software control systems such as robot control systems.

2.2.2 Scheduling Test Cases as Part of Continuous Integration

In many settings, a test suite can be executed in several ways, for example, by order of test case execution or by the machine to which each test case is assigned for execution. If test case execution machines exist, several test cases can even be executed in parallel. However, it is common for constraints on test cases and target machines to require an execution schedule to be solved to minimize the test suite execution time.

The scheduling of test cases, even outside of CI, has not, to our knowledge, been mentioned in the literature. To our knowledge it seems that the default method of scheduling test cases in a CI environment is to set up a static schedule based on one or more static test suites. However, we know from our industrial collaboration that test cases need to be scheduled and that simple methods are being used in industry.
Our cooperation with Cisco, Norway, provided great insight into their simple yet powerful scheduling approach, compared to static scheduling. Their approach is based on a first come, first served principle, where the scheduler dispatches test cases for execution in order of appearance. However, test cases requiring external resources such as a special network interface are postponed until the resources are available. This method of scheduling test cases, however, does not calculate an optimal schedule with respect to the available time allocated for test execution.

In short, the scheduling challenge in CI is to select the best test case to execute at a given time on a given target execution machine so that the total time in the CI cycle is minimized. However, the time allocated to solve the scheduling problem needs be taken into account to minimize the overall test execution time.

2.3 Constraint Programming

This section first provides a general introduction to CP. Further, it presents a more special direction of CP, that is, CP over a finite domain. Finally, it discusses the use of CP for test case generation and scheduling.

CP is a well-known programming paradigm that was introduced 20 years ago to solve combinatorial problems in an efficient and flexible manner. The programming paradigm consists of a way to describe relations between variables in the form of constraints. 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, that is, an assignment of values to variables that belong to the required domains and 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 involve complex control conditions (conditionals, disjunctions, recursions) and integrate dedicated and optimized search procedures.

The use of domains in CP typically distinguishes one CP problem from another. The domain could be a continuous one, typically with real numbers. Domains of lists, Boolean, trees or sequences are other domains that can be used to express a CP problem.

Many problems that can be expressed as a CP problem can also be expressed through other techniques. It is important to emphasize that...
other techniques such as satisfiability (SAT) or satisfiability modulo theory (SMT) techniques \cite{9}, search-based techniques \cite{36}, and mixed integer programming \cite{23} are often very comparable in terms of performance and the possibility of expressing the problem.

### 2.3.1 Constraint Programming over a Finite Domain

CP over a finite domain is an important subclass of constraint domains where the values of the variable $V$ are restricted to a finite set. An example of such a finite domain is $D = \{\text{red, blue, black}\}$ or $D = \{2, 5, 8\}$. A bijective mapping between a finite set of symbols and a finite set of integers is trivial for finite domains and, in the above example, we could therefore easily say that $D = \{\text{red} = 1, \text{blue} = 2, \text{black} = 3\}$ can switch completely to using integers. The constraints, $C$, used in finite domain CP are typically simple relations such as $=, >, <$ or more advanced constraints such as \texttt{ALL_DIFFERENT} \cite{27}, which constrains a set of variables to be different from each other.

Use of CP over a finite integer domain is particularly useful for modeling discrete optimization and verification problems such as scheduling, planning, packing, and timetabling.

The following small, well-known example highlights some of the most basic properties of CP. The \texttt{SEND+MORE=MONEY} \cite{59} problem is based on the crypto-arithmetic problem

$$\begin{array}{cccc}
    S & E & N & D \\
+ & M & O & R & E \\
\hline
    M & O & N & E & Y
\end{array}$$

where each letter represents a different integer from zero to nine. This problem can be modeled as a constraint problem, as follows:

**Listing 1:** The \texttt{SEND+MORE=MONEY} implemented in SICStus Prolog

```prolog
:- use_module(library(clpfd)).

smn([S,E,N,D,M,O,R,Y]) :-
  domain([S,E,N,D,M,O,R,Y],0,9),
  all_different([S,E,N,D,M,O,R,Y]),
  S #\=0, M #\=0,
  1000*S + 100*E + 10*N + D
```

12
The program starts by setting the possible domain for all of the variables and then stating all the constraints. Finally, the labeling statement will start the search among the variables (S,E,N,D,M,O,R,Y) and try to assign them values from the domain (zero to nine) so that all the constraints are satisfied.

By launching the above constraint model in a constraint solver, we obtain the following answer:

Listing 2: Solution to the SEND+MORE=MONEY problem from Listing 1. This is the only solution; that is, no other assignments of the variables exist that satisfy the constraints.

| ?- smn([S,E,N,D,M,O,R,Y]).  
| S = 9,  
| E = 5,  
| N = 6,  
| D = 7,  
| M = 1,  
| O = 0,  
| R = 8,  
| Y = 2 |

Although this example is very basic, it illustrates the general approach to solving a CP problem: 1) Assign a domain to all unknown variables, 2) post the constraints, and 3) initiate the search for a solution.

### 2.3.2 Solving Constraint Programming Problems over a Finite Domain

A CP model over a finite domain will, in many practical situations, have more than one distinct solution. The example shown in Listing 13 has only one solution. However, it is not uncommon for the solution space to be very large and it is often a challenge to find one distinct solution in the solution space that optimizes a given objective function. An objective function can be related to finding the solution with the lowest cost, the solution with the highest profit, or the solution with the shortest schedule. Problems in which a CP model searches among a large set of possible solutions typically has one of the following outcomes.
2. BACKGROUND SUMMARY

- **No solution with proof**: The problem provably has no solution. The interaction between the constraints and the search algorithm is such that the problem cannot be satisfied.

- **No solution without proof**: The search algorithm was unable to find a solution within a given contract of time. This means that a solution could exist, but the time allocated to the search was insufficient to draw any conclusions.

- **Quasi-optimal solution**: At the end of the time contract, a solution is returned, meaning that the search calculated an estimate for the optimum but was interrupted while trying to prove the solution’s optimality. There could be a better solution in the solution space that has not yet been found.

- **Optimal solution with proof**: Before the end of the time contract, the search algorithm returns an optimal solution. This means that the search found a guaranteed global optimum, that is, the search explored the entire search space.

As we can see from the above, a search therefore typically operates with a timeout. This element is essential for many problems in which there is a trade-off between the time searching for the best solution and the result of the solution itself, which can be the execution time of a schedule or of a test case.

2.3.3 Constraint Programming for Test Case Generation

In this section, we describe some of the main tasks that need to be executed when developing a test case or a test suite. Further, we discuss how CP can be used with the intention of automating this task.

When developing a test suite for testing a software system, we can divide the development into two parts: the generation of the test cases, often referred to as the input data, and the generation of the expected result, which is also known as the output data or the oracle. There are, however, many different approaches in the literature. In this thesis, we use CP for both test case generation and for generation of the expected result.

One of the trends in software engineering has been to use CP and constraint solvers to calculate the test inputs and the oracle. The
The general idea behind constraint-based testing is to use constraint-solving and constraint optimization techniques to generate test cases and the oracle for the program under test. The software program or a model of the system to be tested can be expressed as constraints with corresponding domains. Then, by providing a set of test objectives, the CP model extracts a test case and oracle to satisfy the test objective. A test objective could generate input data to a unit test such that all possible paths through a program are executed or generate a set of inputs such that all possible assertions in the program are tested.

The idea of using CP for automatic test case generation dates back to the early 1990s, to the work of Marre [33] and Dick and Faivre [11]. Use of CP for test case generation has been shown to be both versatile and flexible. Examples include CP being used to automatically generate unit tests by conducting static source code analyses [60, 19]. More recently, Di Alesio et al. [10] used CP to generate stress tests for real-time applications. They constructed a complex declarative constraint model implementing the behavior of a priority-based real-time operating system with the goal of finding worst-case scenarios with respect to deadline misses. Further, K. Balck and Pearson [25] used CP to generate replay scenarios for a public warning system service as part of the 4G Long Term Evolution (LTE) standard.

### 2.3.4 Constraint Programming for Scheduling

When given a test suite containing multiple test cases and more than one machine is available to execute each test case, we can decrease the time it takes to execute all test cases by running them in parallel. However, constraints must often be considered when setting up such a parallel test case execution. These constraints can, for example, relate to the machine each of the test cases is to be executed on or the order in which they are executed. When such execution constraints need to be accounted for, the execution problem can turn into a complex scheduling problem.

The field of scheduling is very broad. Scheduling can relate to how a kernel in an operating system decides which task to execute and when. It can also relate to finding the optimal way of carrying out a building project, solving a rostering problem, or planning a sports tournament. There are many tools and techniques for solving and modeling scheduling problems from different communities, such as operations research [26], and genetic algorithms [8]. However, many scheduling problems, even if
quite diverse, turn out to share similarities, which makes CP a possible candidate for a tool for both modeling and solving schedules.

Over the years, several variants of scheduling problems have been addressed by both the CP and operations research communities. Lombardi and Milano [30] point out that neither CP nor operations research techniques can claim to have the best approach; they both perform equally well. However, CP has been shown to work extremely well for some scheduling problems [41, 49], and is also attractive because of its declarative nature and closeness to the mathematical model.

2.4 Testing Robot Systems

We define a CIR (paper 1) as a classical industrial robot with an additional control system attached that performs an industrial or physical process. This control system is typically responsible for controlling such physical processes as painting, gluing, and welding. This means that we have one control system responsible for moving the mechanical arm of the robot and an additional control system responsible for controlling the physical process.

The interconnection between the motion control system and the process control system typically varies. Some process control systems have a very tight interconnection with the motion control system through interfaces such as dual-port RAM, while others have a looser interconnection, such as industrial field buses or industrial Ethernet. Testing software for CIRs is a complex task, mainly because of this separation between control systems. The separation introduces delays and latency, which is one of the main reasons why testing becomes complex.

To facilitate process-intensive tasks such as painting, gluing, or sealing, it is convenient to decouple the process control system from the motion control system even further. However, this decoupling introduces additional challenges with respect to testing such robot systems. In particular, a key requirement for robotized painting is the ability to precisely activate the process equipment along a robot’s programmed path. However, many of the physical processes involved in robotized painting are relatively slow compared to moving the mechanical robot. Consequently, advanced computationally based techniques have been established to take advantage of the knowledge of the slower physical processes to compensate for these latencies. The validation of such systems 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 often requires access to the physical layer to activate many of the robot’s features. Much of the testing is based on running a full-scale system with a moving robot and measuring outputs with instruments such as an oscilloscope. This results in a lengthy period between a change in the software and the time of testing and reduces the possibility of automating testing. In addition, many of the tests produced for one configuration cannot be easily 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 design process, often close to the release date, leading to increased validation costs.

3 Main Contributions

In this section, we introduce the two main contributions of this thesis. The first contribution is a methodology for automatic test case generation for ABB Robotics’ IPS. The methodology, which we have named Constraint-based Automatic Testing of IPS (CATS), is designed to be executed as part of a CI cycle. We present more details of CATS in Section 3.2.

The second contribution relates to test case execution scheduling inside a CI cycle. We have named our methodology TC-Sched and will present it in more detail in Section 3.3.

3.1 Relation between CATS and TC-Sched

Both CATS and TC-Sched are designed to reduce the overall testing time, $T_t$, in a CI cycle. This section presents an overview of how the two methodologies are related to each other and how they are integrated in a full CI cycle. It is well known [16, 42, 43] that an important success criterion for CI, is the round-trip time of a CI cycle, or the time it takes to return test results to the developer. Recall from Figure 2 the different phases in a CI cycle. Figure 3 shows a complete CI cycle laid out in time, where each phase in the CI cycle will contribute to the overall time, as follows:
3. MAIN CONTRIBUTIONS

**Developer commit.** When a developer is finished with a small task involving code change, he will submit the changes to the source control repository. The time it takes to do the commit can in most cases be ignored.

**Software building.** Building the software occupies a considerable slice of $T_{ci}$. This task include fetching source code from the source control repository, compiling it, and packing it into installation packages. To optimize the build time, several techniques are used, such as distributing the builds\[44\], building software in parallel\[2\], caching builds\[53\], using pre-compilation\[61\], backtracking on the last successful build\[52\], and using a distributed workflow\[50\]. Google has even designed a new language, Go\[18\], with the express goal of fast build times.

**Software deployment.** The time it takes to deploy the software can be neglected for some products (e.g., copy a single file). For other products, it can take considerable time (e.g., reinstalling a complete operating system). The time will often depend on the physical characteristics of the target platform. An example from ABB Robotics shows that installation times can vary from more than 30 minutes when the upgrade is performed over serial lines to less than five seconds when done over Ethernet. This makes it difficult to generalize the time it takes to deploy software.

**Testing.** The testing phase often involves running a static set of test cases. In more advances settings, this will also include running multiple tests on multiple embedded computers in parallel.

**Developer feedback.** Providing feedback to the developer is usually just a matter of sending an email with the results of the build and test. This time is short and can be ignored in most cases.

The time from when a change is committed to the source control repository until the developer is notified of possible errors is denoted $T_{ci}$. The time spent in the testing phase is denoted $T_i$, where $T_i = T_s + T_e$. The variable $T_s$ is the time spent in generating either a test case or a test case execution schedule and $T_e$ is the time to actually execute the test case or schedule.

This thesis focuses on the testing phase. Both our contributions, CATS and TC-Sched, aim to reduce the test execution time $T_i$. Figure[1]
Figure 3: A complete CI cycle. The total cycle time is denoted $T_{ci}$. The testing time is denoted $T_t = T_s + T_e$, where $T_s$ is the time spent to generate or schedule a test suite and $T_e$ is the time it takes to execute the generated test suite or schedule.

shows how CATS and TC-Sched relate to each other and to the complete CI cycle.

Figure 4: Overview of how CATS and TC-Sched interact in a complete system.

In a CI cycle, CATS automatically generates a test suite to test ABB Robotics’ IPS. By describing the IPS as a CP model, CATS is able to automatically generate test cases. These test cases can be scheduled in combination with other test suites for execution using TC-Sched. TC-Sched is also based on a CP model intended to be solved as part of the testing phase.

The goal of both CATS and TC-Sched is to find a solution based on constraints given as inputs to the model, in addition to solutions such
that the trade-off between $T_s$ and $T_e$ minimizes $T_t$.

### 3.2 CATS

In this section, we present CATS and provide the background for its use. For CIRs, the single most important function it needs to master is to move the mechanical robot arm along a programmed path and perform physical operations synchronized to the movement along the path. The physical operations that are synchronized to the path typically classify the type of robot. These operations can be simple, such as opening/closing a gripper for handling robots, taking a picture for an inspection robot, or turning on a spray paint pattern for a paint robot. All of these examples involve an external physical process that needs to be synchronized with the movement of the robot arm. Another common feature is that the time it takes to operate the physical process is significant and needs to be accounted for. Examples of such process delays are the time it takes to open a solenoid valve, accelerating a motor to the correct speed, and filling a hose with air.

As already mentioned in Section 2.4, it is common to separate the motion control system and the process control system for advanced processes such as painting and gluing. This is done for the various reasons already addressed in Section 1.1. One of these reasons is to move the control system physically closer to the physical process. Another reason is to be able to use one process control on different types of motion control systems, even running the process control system as a standalone module without the use of any robot.

This separation of the motion control system and the process control system has many obvious benefits. However, there are also some drawbacks. One drawback is that testing becomes more complex, mainly due to the physical distance between the two control systems but also because they need to be time-synchronized (e.g., IEEE 1588 [28]), complicating matters even further.

ABB Robotics’ IPS is such a separated process control system. The IPS is a distributed paint control system designed mainly for installation on ABB Robotics’ robots, but it can be installed on other types of robots and also includes a standalone version. The IPS is highly configurable and several different embedded controllers can be tied together over a distributed industrial network to form it. One of the key features of
the IPS is the synchronization of many physical outputs (valves, pumps, airflow, air pressure, etc.) which together form the spray pattern with the robot’s motion. Due to the IPS’s distributed nature, the different parts of the IPS can be placed close to the physical processes on the robot. This improves the accuracy of the control of the process and reduces the robot’s paint consumption. As mentioned in Section 2.4, this distributed nature makes testing challenging. Until now, there was no systematic, automatic
3. MAIN CONTRIBUTIONS

We developed CATS to overcome the problems raised above. The core of CATS consists of a CP model of the IPS that captures the relation between the control systems moving the robot and those activating the physical processes along its path. CATS can generate configuration parameters for the IPS, test cases, and a test oracle \cite{54} for an IPS behaving in a normal, non-erroneous fashion. CATS can also drive the IPS into error states to verify that it is able to operate in and recover from such situations. A high level overview of CATS is shown if Figure 5.

CATS is currently installed in ABB Robotics’ software development infrastructure. It has been running for more than two years, solving the CP model several times a day. It has shown that it can detect both completely new errors in addition to old, reintroduced errors.

3.3 TC-Sched

It is well recognized \cite{43} that the time from when a change is committed to the source control repository until the developer receives feedback on potential errors is crucial. It is important to minimize this time while still maintaining a high quality of testing. On the one hand, minimizing the scope of the testing is desired to shorten the feedback time to the developer; on the other hand, extending the scope is desired to raise the quality of testing. To address this problem, we developed TC-Sched, the second main contribution of this thesis. The main idea of TC-Sched is to schedule the execution of a test suite in parallel on a set of machines available for executing test cases. This method is particularly useful for applications that are executable on many different platforms. For such applications, it is common for test cases to be reused between the different platforms. In this context, a platform can be a hardware platform or an operating system or have another feature that differentiates the machines executing test cases.

TC-Sched is a cost-effective method for test execution scheduling on multiple machines with constraints on accessible resources, such as measurement devices or network equipment. TC-Sched takes as input a
test suite, a set of machines available for test cases execution, and a set of shared resources and produces an execution schedule. The schedule guarantees that each test will be executed once and minimizes the round-trip time, that is, the time it takes to solve the schedule and execute it. Figure 6 shows a high-level schematic of TC-Sched. As with CATS, TC-Sched is based on a CP model over a finite integer domain and is designed to run as part of a CI cycle.

**Figure 6:** TC-Sched overview. TC-Sched takes as input a test suite, where each test case can require exclusive access to a resource (e.g., an instrument) and a set of machines on which the test cases can be executed. TC-Sched then generates an execution schedule that satisfies the given constraints.

### 4 Research Method

In this section, we describe the research methods used in the entire research project. From a high-level perspective, the research methodology is divided into four parts:

(i) Industrial context and problem identification (Section 4.1).

(ii) Background study and literature review (Section 4.2).

(iii) Modeling the problem (Section 4.3).

(iv) Implementation and evaluation (Section 4.4).
4. RESEARCH METHOD

4.1 Industrial Context and Problem Context

Having worked for ABB Robotics for more than 15 years, I am well aware of the actual challenges and problems ABB Robotics has from an engineering point of view. To get a broader, more scientific view of these challenges, we started by identifying two problems and tried to map them to similar problems observed by other industrial partners and the research community. The two problems we identified are related to the automatic testing of the behavior of timed events (CATS) for robots and test case execution scheduling (TC-Sched) and their performance as part of a CI cycle.

4.2 Background Study and Literature Review

For the first problem identified, the automatic testing of timing-related events for robots, there was no industrial partner with whom to discuss the issues. Although industrial robots are common nowadays, the challenge of synchronizing an external process control system to a motion control system was not addressed by the research community at the time. Furthermore, besides ABB Robotics, only two companies in the world manufacture high-end robots with advanced process control systems such as those involved in painting. In addition, these two companies have no tradition of publishing their research in this specific field.

However, we have had discussions with companies in other industrial domains, such as the testing of synchronization in smart grid technology [35, 29] and the synchronization of events in distributed programmable logic controller (PLC) systems. These discussions showed that we were all facing similar problems with respect to automatic testing. However, the partners we discussed our issues with mainly had experience in manual testing and less experience with automatic testing. None had any experience in testing within a CI cycle, as we required.

Regarding the TC-Sched problem, we held valuable discussions with Cisco, Oslo, a branch that develops video conferencing equipment. In addition to seeing that we face some of the same challenges, we also shared valuable experiences related to test case execution scheduling. Cisco already had a simple system for scheduling test cases. Although Cisco uses a minimalistic and simplified scheduling approach, we still gained valuable experience from these discussions. Cisco’s approach is based on a simple first come, first served scheduler. Cisco also provided industrial test suites to use in our own empirical evaluation of TC-Sched [paper 3].
On the more theoretical side, we also reviewed the related literature. Testing of timed event system has received significant attention (e.g., \cite{45, 51, 62, 35}). However, we found no references to systems where the test case generation was integrated as a dynamic part of a CI cycle. We experienced the same for the scheduling of test cases. However general work on scheduling is widely referenced in both the CP community (\cite{4, 10, 47, 48}) and, in general, the software testing community (\cite{24, 6}).

The review helped us understand the current state of CI-based testing and the use of CP for test case generation and scheduling.

### 4.3 Modeling Problem

For both CATS and TC-Sched, we began by formulating mathematical models. The mathematical models were derived from knowledge of the IPS and prior test execution scheduling discussions we had held internally and with industrial partners. Initially, the mathematical relation was documented in internal technical reports, which formed a foundation for discussion between researchers and developers. These technical reports were later used as input for the published papers. The mathematical models were then transferred to experimental CP models. These experimental CP models were used in smaller experiments, especially experiments related to finding optimal search heuristics and the speed of the solving process. Ultimately, the experimental models were rewritten to industrial-quality standards and adapted to the current build and test infrastructure at ABB Robotics. After the first version was integrated in a full-scale setting, we continued to make improvements to the smaller experimental models, which were later transferred to the full-scale models.

### 4.4 Implementation and Evaluation

CATS has been fully integrated into the ABB Robotics build and test environment. To perform this task, the current build and test tools had to be studied in detail. This involved several modifications of the current system, which can be divided into two major tasks:

- Comparing the measurements of actuator outputs and their corresponding times with the oracle produced by CATS. This task might seem trivial, but it turned out to be quite complicated, since the IPS does not store the time and value of an actuator output if no changes
are made. We evaluated several possible solutions to this problem. One of them was based on an extension of the CP model to capture more of the IPS’s actual behavior. This approach was discarded mainly because the complexity of the CP model would increase to a level that would overly complicate the model’s maintenance. Instead, we decided to add more logic and knowledge to the software part responsible for comparing the oracle with the measured actuator outputs. We consider this solution to be more in line with recommended practices [54], which emphasize a simple and minimalistic model.

- The clock on the computer that is responsible for executing CATS and running the derived test cases [paper 1], needs to be synchronized with the clocks on the embedded controllers running the IPS. This requirement forced us to perform a major rebuild of the interfaces to the IPS and develop a clock synchronization algorithm inspired by IEEE 1588 [28]. However, this extension was shown to be applicable to other ABB Robotics products and we were able to reuse this synchronization in contexts not related to testing. We already used this rebuild in one of our other products, RobView [1]. The work done in cooperation with this thesis is therefore also applicable outside software testing.

CATS went through extensive evaluation during its entire development. The goal of this evaluation was to ensure that the proposed methodology actually contributed to detecting bugs faster and earlier in the development cycle. The evaluation was mainly divided into the following parts:

- Evaluation of the model itself, including performance of the heuristics [paper 1, paper 2].
- Evaluation of the quality of the generated test cases compared to manually generated test cases [paper 2].
- Evaluation of CATS when deployed in a CI cycle, including its ability to find old, reintroduced bugs [paper 1].

TC-Sched has not been integrated to the same extent as CATS, which has prevented us from carrying out a full-scale evaluation in a CI environment. However, the empirical evaluation conducted [paper 3] on both artificially
SUMMARY  5. SUMMARY OF RESULTS

generated test suites and real-life industrial test suites shows that the proposed methodology corresponds to the expected results. Therefore, ABB Robotics has decided to build and deploy a full-scale implementation of TC-Sched in continuation of this thesis.

5 Summary of Results

The main research results of our work are elaborated in the three papers included in this thesis. In this section, we summarize the key results obtained for each paper.

5.1 Paper 1


This paper is a journal extension of the following papers:


This paper reports on CATS from a software engineering perspective, focusing mainly on the software testing. The CP model is mentioned and explained but given less attention. We introduce the domain of robotized painting, with a special focus on how testing is currently performed and its challenges. We also present the mathematical model of the IPS on which CATS is based. We introduce the concept of just-in-time test generation (JITTTG) and discuss experience gained by the deployment and use of JITTTG.

Paper 1 answers the following research questions:
5. SUMMARY OF RESULTS

- **RQ1: How efficient are the search heuristics of CATS?** We perform a detailed analysis of experimental results for a simplified version of the constraint model used in CATS. We determine the most appropriate parameterization of the search process. We conclude there are two different, complementary search heuristics that are the most efficient.

- **RQ2: What is the model scalability of CATS?** Through a controlled experiment on a full-scale model, we determine the scalability of CATS. We tested a large set of different combinations (length of test sequence and number of different spray patterns). We found that, for all combinations, CATS provided a first solution within 10 seconds. Most of the combinations generated a first solution in less than three seconds. We also performed an experiment on the same setup that determined the optimal solving time when minimizing the total execution time. We found that there is no benefit to run the solving process longer than 12 seconds for the configurations we tested. This result assumes that a test case is only used once. If a test case can be used multiple times, there is greater benefit in investing more in the solving phase.

- **RQ3: Can CATS be adapted in an industrial environment?**

  This research question addresses how CATS can be integrated within an industrial environment. We divided the question into the following sub-questions:

  - **Can the model detect new errors?** CATS found three previously unknown errors in the IPS when it was deployed. Two of them were directly related to the IPS’s behavior, while the last was related to how a PC tool presents online diagnostics for a live system.

  - **Is the model able to detect old errors that were reintroduced into the IPS?** 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 them. The results show that CATS was able to find them all.

  - **Does the proposed JITTG framework behave as expected?** We discuss our core experience gathered through the deployment of CATS. We point out several key factors and
SUMMARY

5. SUMMARY OF RESULTS

lessons learned that we consider important to the success of the CATS deployment in a full industrial setting at ABB Robotics.

5.2 **Paper 2**

“Using Constraint Programming to Test Paint Control System for Robots”. Mossige, Gotlieb, and Meling. Invited journal paper submitted to the journal *Constraints*. This paper is a journal extension of the following paper [39]:


**Paper 2** builds on the results found in **paper 1**. We shift the focus from software engineering/software testing to CP. We present the mathematical relations in the IPS in a way more suited to CP, which includes explaining a real painting case from a CP point of view, with real numbers. Further, we provide a deeper explanation of boundary and start-up conditions and present the more fundamental constraints.

The paper focuses on the complexity of the paint process, including how to obtain an overview of the timing of different processes distributed on different embedded controllers, through the introduction of a graphical, easily understood method of representing the painting process.

In **paper 1**, we present a set of search heuristics, but why the different search heuristics behave as they do remains an open question. This issue is addressed in **paper 2**. It explains the difference between the two search heuristics, including why one of them behaves as it does.

The paper also present how to introduce diversity in the solutions with the help of global constraints. That method avoids useless test cases, establishes diversity in the configuration for the IPS, and at the same time generates test cases for a given test scenario.

Finally, we perform a detailed evaluation in which we analyze how CATS generates test sequences compared to a group of highly skilled paint domain experts. The results show that, even in very small instances, humans are not able to generate the test cases with the given constraints.
within a reasonable time. We had to give the human group more relaxed constraints to even generate the test cases. This shows that CATS is not only capable of generating test cases in a CI environment but also capable of generating test cases that cannot be generated manually.

5.3 Paper 3


In paper 1 and paper 2, we introduced test case generation as part of a CI cycle and showed how a carefully controlled solving process of the CP model can lead to more efficient test cases. Paper 3 maintains the same basic focus, that is, to improve the efficiency of the test phase in a CI cycle. We introduce TC-Sched, which addresses test case scheduling as part of a CI cycle, where we assume the availability of multiple machines on which to execute the test cases and where some of the test cases require exclusive access to an external resource such as an instrument or an actuator.

Paper 3 provides a formal mathematical definition of the optimal test case scheduling problem (OTS) and describes the context of testing in a CI cycle. Further, it presents a solution based on CP and the use of the Cumulatives [4] constraint.

In Paper 3, we answer the following research questions:

- RQ1: How does TC-Sched compare with simpler scheduling methods in terms of schedule execution times?

  The proposed solution is compared with two naive algorithms (random and greedy), where we show that the first solution found by TC-Sched is comparable or better than the solutions provided by the two naive algorithms.

- RQ2: For TC-Sched, will an increased investment in the solving time reduce the overall time of a CI cycle?

  By continuing the solving process after the first solution is found, it is possible to find better candidates. This will, of course, require an extra investment in the time allocated to the search, which contributes negatively to the time spent in the test phase in a CI cycle. By executing TC-Sched on 840 randomly generated test suites,
we conclude that the total execution time, $T_t$, can be minimized by adding extra time to the solving phase, $T_s$. We found that the optimal amount of time to allocate to solving depends on the size of the test suite.

- **RQ3:** Can TC-Sched scale up to industrial optimal test case scheduling problems?

To further validate the capabilities of TC-Sched in handling real-world test suites, we performed an additional controlled experiment: We tested TC-Sched on two real-world industrial test suites, one test suite for ABB Robotics’ IPS [39] and one for a video conferencing system [32]. The tests revealed that TC-Sched was able to find the optimal execution time, $T_t$, by running the solving process for less than one second (IPS) and less than 33 seconds (video conferencing system).

### 6 Further Work and Lessons Learned

The research presented in this thesis has several promising directions for further work. We divide this future work into that related to CATS, presented in Section 6.1, and work related to TC-Sched, presented in Section 6.2. Finally, in Section 6.4 we present some lessons learned during the research.

#### 6.1 Further Work on CATS

CATS has already been in daily use since almost two years and has proved its ability to find errors. Besides adding small extensions to the model, we do not foresee much future work on CATS in its current setting. However, one possible spin-off would be to take CATS out of the CI environment and use it as a standalone tool. In its current usage, the duration of a generated test case is usually constrained by the available time in the CI cycle. We do, however, see the need for generating longer test cases for long-term testing outside of the CI environment.

#### 6.2 Further Work on TC-Sched

Our work on test case execution scheduling has yielded very promising results [paper 3]. However, several tasks still remain to be done to realize
6. FURTHER WORK AND LESSONS LEARNED

SUMMARY

the full potential of our findings. First, TC-Sched needs to be able to handle more realistic industrial requirements. One of those requirements relates to test cases requiring more than one execution machine. This is a well-known requirement \cite{57, 58} in several real-world industrial testing suites, such as the testing of video conference equipment, where one test case can use two or more video telephone terminals to test call functionality.

Another very promising direction for future work relates to scheduling of test cases where the time window allocated for testing is constrained. When scheduling test cases for a time-constrained window, there will not be enough time to execute all the test cases in the test suite, even if an optimal schedule is found. This scenario will require that only a subset of the available test cases in the test suite is scheduled for execution. A possible approach to this selection involves the use of priorities for each test case, in addition to consideration of the duration of the last test case execution. This method will result in a setting where high-priority test cases are executed often and lower-priority test cases are executed less often. This is, however, not a new problem. \cite{12} and \cite{21} address both test suite reduction and prioritization. However, their work does not take place within a CI process. We believe that the previously mentioned examples are good candidates for extension into a CI process.

A full-scale implementation of TC-Sched is highly desirable. ABB Robotics has already decided to implement a full deployment of TC-Sched in their CI environment as a spin-off of this thesis. However, this will require considerable refactoring of the current testing infrastructure. First, we will have to develop an industrial-quality test dispatcher responsible for the actual execution of the schedule. Further, a better system for the collection of historical data is required. Based on experience with the full-scale implementation of CATS, the extensions required on the infrastructure are considerable. Without going into too much detail, the extensions can be summarized as follows:

- Formalization of the definition and execution of a test case. What kind of metadata need to be stored in a test case?
- A system for storing and restoring the historical execution times of a test case.
- Solution of the synchronization problem that can arise if a test case takes more or less time to execute while holding one or more global resources.
We also see that the current search heuristics used for TC-Sched can be targeted for further work. We think that there might be other options that provide a better trade-off when the time allocated to solving the schedule is limited. We will probably favor search heuristics that converge quickly in the beginning of the search but which may lack capabilities to prove optimality. Proving optimality is mainly interesting from a theoretical point of view, but less from an industrial point of view.

### 6.3 New Testing in Continuous Integration

Both CATS and TC-Sched show that the solving of CP models for test case generation or scheduling inside a CI cycle is both possible and effective. This opens up replication of the methodology for application to other parts of the robot system. CATS has only focused on the timing and synchronization of actuator outputs related to a robot’s moving path. Other components, however, could be modeled through either CP models or other modeling techniques, with the intention of applying JITTG (paper 1) in a CI environment. One component for which we are currently considering developing a model is the error reporting module in the IPS. We are aware that customers of robot systems have a desire for more precise error messages while requesting that “unimportant” errors be suppressed. This leads to a complex IPS error module. A model of the error system will make it much easier to validate extensions made to the system as part of new customer demands, as well as extensions to the error reporting module system.

### 6.4 Lessons Learned

The research performed as part of this thesis has taught us several lessons, based on experience gathered through the development and deployment of the test framework and discussions with test engineers at ABB Robotics. We summarize these lessons as follows:

- **Greater confidence when changing critical parts:** Based on feedback from developers, there is now less apprehension about applying changes to critical parts of the code. Previously, such changes involved significant planning efforts and had to be synchronized with the test engineers responsible for executing tests manually. With the new testing framework as part of the CI cycle 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 caused unwanted side effects, it is rolled back to keep the build “green.”

- **Simple frontend, complex backend:** By using Python [46] as the frontend interface to the constraint solver and keeping the test engineer’s interface as simple as possible, we can utilize personnel with a minimal computer science background. [17] and [3] recognize that CP has a high threshold for usage. By limiting the training to the introduction of famous classical problems such as the SEND+MORE=MONEY problem [34], test engineers were given enough training to use the constraint solver from Python without major problems.

- **Less focus on manual, legacy tests:** A positive side effect of introducing model-based testing is that the organization’s focus shifted from mainly manual testing toward more automatic testing. Even for products beyond the scope of this thesis, the introduction of a fully automatic test system as part of a CI cycle has inspired other areas of ABB Robotics’ internal organization.

- **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 maintain other test suites that do not have such tight integration and thus their tests may occasionally produce false errors.

- **Return on Investment with CATS** Computing the return on investment for ABB Robotics for the work performed during this thesis is not easy. One could potentially measure the number of defects found with and without the CATS model during the validation of a new IPS release or compare the human effort required in both cases. However, none of these measurements have been carried out yet. In the long term, we expect CATS to be recognized as a way to increase the general quality of the testing process, since necessary refactoring will be performed before the technical depth grows out of control.
7 Conclusion

This thesis examines the solving of CP models as part of CI, where the motivation is to increase the efficiency of testing during the CI cycle. The work is divided into two distinct problem formulations: 1) automatic test case generation for ABB Robotics’ IPS and 2) automatic test case execution scheduling.

For the test case generation of ABB Robotics’ IPS, we developed a CP model describing the behavior of the timing involved for the activation of actuators synchronized to the path a robot arm is moving along. This model can generate both IPS test sequences and configurations, where the IPS is in a normal, non-erroneous operating state. The model can also generate test sequences with corresponding configurations in which the IPS is forced into error states and illegal behavior. The complete model has been integrated into ABB Robotic’s CI environment, where it is currently used as part of daily automatic testing.

For large instances of test cases generated by the CP model, we also performed experiments that investigated how the trade-off between solving the model and executing the test cases in a CI environment. We show there is little to gain from running the solving process for more than around 10 seconds. Furthermore, we conducted an experiment that compared test cases generated by a group of highly skilled engineers and test cases generated by our model. The most astonishing result was not that the model was able to generate better (in terms of execution time) test cases than the manual ones, but that our model was able to generate test cases that a group of highly skilled engineers could not, even with more relaxed constraints and tools to help generate the test cases. This result shows that not only can our model help generate test cases, but also it can generate test cases that cannot be generated manually.

Finally, we show that we are not only able to generate efficient test cases as part of CI, but we are also able to schedule the test cases as part of CI. In paper 3, we develop a CP model for test case execution scheduling. We performed comparable experiments as for the CP model of the IPS, where we found an optimal trade-off between solving the schedule and executing it. The results show that, for the test suites we investigated, it is possible to set a specific time-out that will generate the optimal result. This time, however, depends on the size of the test suite and the number of machines available to execute the schedule.
To summarize our findings, we show that both test case generation and test case execution scheduling modeled based on CP and solved as part of a CI cycle are very efficient and definitively an area where more research can be performed based on the same ideas.

References


Notes in Computer Science. **Best Application Paper Award.** Springer, 2014, pp. 25–41. ISBN: 978-3-319-10427-0. DOI: [10.1007/978-3-319-10428-7_6](http://dx.doi.org/10.1007/978-3-319-10428-7_6) URL: [http://dx.doi.org/10.1007/978-3-319-10428-7_6](http://dx.doi.org/10.1007/978-3-319-10428-7_6).


REFERENCES


M. Mossige$^{1,3}$, A. Gotlieb$^{2}$, H. Meling$^{3}$

1 ABB Robotics, Nordlysvegen 7, 4349 Bryne, Norway

2 Certus Centre for Software Verification and Validation, Simula Research Laboratory, P. O. Box 134, N-1325 Lysaker, Norway

3 Department of Electrical Engineering and Computer Science, University of Stavanger, 4036 Stavanger, Norway
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 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 [29]. 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:

(i) 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.

(ii) A full-scale implementation of the model is presented with constraint programming tools [27]. 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.

(iii) 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 including 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 [15] 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, and test servers. 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. Fitzgerald and Stol [14] 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 [12, 30]. 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 [11, 7, 25, 3, 19] 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. [20] report on the inclusion of system execution modeling tools to test distributed real-time systems as part of continuous integration. However, to the best of our 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.

2.2 Model-based Testing 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.

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, Liu, and Lee [40] reports on the use of discrete-event systems (DES) [32, 6] 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 [24, 22] 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) [31, 35], whose modeling is known to be challenging [23]. Broy et al. [4] 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.
MBT. MBT [37] 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 [37], 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 [2].

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.

Constraint-based testing. Use of constraint programming for automatic test case generation has been around for a long time e.g. Gotlieb, Botella, and Rueher [17], Marre and Blanc [26], Di Alesio et al. [9]. Gotlieb et al. [18] developed a constraint programming model for automatic test case generation for C programs. Similarly, Marre and Blanc developed GATeL [26], 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. [10] 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
ABB’s Process Control System

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 Figure 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 Figure 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
3. ABB’s Process Control System

pump; a large change will take longer to apply than a small change, due to the acceleration of the motor.

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 [36]. 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 Figure 2. As we can see, the robot controller communicates with an embedded controller, denoted the
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

**Figure 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 [24]. 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.

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.
4. Legacy Test Practices

Figure 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.

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 [13].

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 Figure 3. 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.

1The video at [http://youtu.be/oq524vu05n8](http://youtu.be/oq524vu05n8) also shows painting on paper.
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.

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 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.

[2]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]
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. Modeling the IPS

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 Figure 4, each channel is shown as an output of the model.

5.2 Mathematical Model

Abstractly, the IPS can be modeled as shown in Figure 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 \(i\)-th spray pattern \(B_i\) and its application time \(t_i\). This sequence

\[
P_j(i) = L[B_i(j)] \quad \forall j \in 1...5
\]

\[
t_{c_i} = f_{c_i}(P_{j,i}, P_{j,i-1})
\]

\[
t_{a_i} = f_{a_i}(P_{0,i}, P_{1,i-1})
\]

\[
t_{j,i} = t_i - t_{a_i} - t_{c_i}
\]
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 $(B_i, t_i) \mapsto (P_{j,i}, t_{j,i}), \forall j$.

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 Figure 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 $L$ that is consulted to perform the transformation $B_i \mapsto P_{j,i}$.

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 Figure 4 contains configuration parameters describing how to generate the IPS test model. These parameters typically include the length of the test sequence, the type of test scenario, and so on.

### 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:

$$P_{j,i} = L[B_i][j], \quad \forall j \in 1 \ldots 5$$  \hspace{1cm} (1)

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

$$P_{0,i} = 1 \quad \text{if } P_{1,i} > 0$$  \hspace{1cm} (2)

$$P_{0,i} = 0 \quad \text{if } P_{1,i} \leq 0$$

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

By referring to Figure 4, 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. Equation (3) shows how this calculation is performed:

$$\forall j \in 0...5, \forall i \in 1...N$$
$$t_{j,i} = t_i - t_{a_{j,i}} - t_{c_{j,i}}$$

$$= t_i - f_{a_{j}}(P_{j,i}, P_{j,i-1}) - f_{c_{j}}(P_{j,i}, P_{j,i-1})$$

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 Equations (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:

$$f_{a_{j}}(u, v) = \begin{cases} 
PreTime & \text{if } u = 0 \land v \neq 0 \\
PostTime & \text{if } u \neq 0 \land v = 0 \\
0 & \text{otherwise}
\end{cases} \quad \forall j \in 2...4$$

where $PreTime$ and $PostTime$ are considered constant configuration parameters (see also Table 4 on page 89).
5.6 Timing on Isolated Channels

Each channel has its own set of parameters that can be used to adjust its timing characteristics. This timing is calculated using the channel compensation function $f_c$, shown in Equation (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:

$$f_{cj}(u, v) = \begin{cases} 
D_j^- \cdot \left( \frac{v-u}{\text{Max}_j - \text{Min}_j} \right)^K_j & \text{if } u < v \\
D_j^+ \cdot \left( \frac{u-v}{\text{Max}_j - \text{Min}_j} \right)^K_j & \text{if } u > v \\
0 & \text{otherwise}
\end{cases} \quad (5)$$

where $K_j \in \{0, 1\}$ is used to enable or disable the linear delay component. $\text{Max}_j$ and $\text{Min}_j$ give the maximal and minimal value the physical actuator output can take. The terms $D_j^+$ and $D_j^-$ are considered constant configuration parameters (see also Table 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 Figure 5 and discussed in detail in the following sections.
6. Test Scenarios with Constraints

(a) Normal scenario.

(b) Overlap scenario.

(c) Shutdown scenario.

Figure 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.

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 Figure 5(a):

\[
\forall i \in 1 \ldots N,
\begin{align*}
t_i - t_{i-1} & \geq \text{MinBrushSep}, \\
t_i & > t_{i-1}, \quad t_i \geq 0, \\
B_i & \neq B_{i-1}, \quad B_i \in 0 \ldots \lambda
\end{align*}
\tag{6}
\]

\[
\forall j \in 0 \ldots 5, \forall i \in 1 \ldots N
\begin{align*}
t_{j,i} - t_{j,i-1} & \geq \text{MinTrigSep}, \\
t_{j,i} & > t_{j,i-1}, \quad t_{j,i} \geq 0
\end{align*}
\tag{7}
\]

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

\[
t_e - t_{c+\text{BurstLen}} \leq \text{BurstTime}
\tag{8}
\]

\[
t_{c+\text{BurstLen}} - t_{c,e} \leq \text{BurstTime}
\tag{9}
\]

BurstLen, BurstTime, c and e are configurable input parameters in the model (see Table 4).

### 6.2 Overlap Scenario

Overlapping events are probably one of the most interesting scenarios that can be generated, as shown in Figure 5(b). This scenario is best
explained with a simple example. Assume that one actuator 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 $P_{t2} = 1$ and $t_{t2} = 15$. Assuming that the current time (gettimeofday()) is less than 10, it is easy to see that $t_{t2} = t_{t2} - D^+ = 15 - 10 = 5$. 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:

\[\begin{align*}
t_{c,e} - t_{c,e+1} & \geq \text{MinOverlapTime}, \\
t_{c,e+1} - t_{c,e-1} & \geq \text{MinTrigSep}, \\
t_{c,e+2} - t_{c,e} & \geq \text{MinOverlapTime}
\end{align*}\]

where $t_{c,e}$ represents the activation time for a specific user configurable channel $c$ and user configurable event $e$. Note that $\text{MinOverlapTime}$ and $\text{MinTrigSep}$ are considered positive constants given as input when a test sequence is generated (see also Table 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 Figure 5(c) and its constraint is specified as

\[ P_{c,e} = \text{IllegalVal} \]  

where \(\text{IllegalVal}\) is a configurable input parameter in the model (see Table 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 \(t_S + \delta \leq t_i\). This means that the IPS receives each pair \((B_i, t_i)\) around \(t_i\) before the activation time. In practice, the value of \(t_S\) 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 Figure 6 and explained below.
7. Implementation

(i) **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.

(ii) **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.

(iii) **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.

(iv) **Query and Solve Model:** A set of basic smoke tests 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 generated 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.

(v) **Run Test:** Finally, the actual test is executed by applying the generated test sequence and comparing the actuator outputs with the model generated oracle. Figure 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 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 Just In Time Test Generation

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, Pretschner, and Legeard \[38\] 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) \[27\].
Figure 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.
CP is a well-known paradigm introduced 25 years ago to solve combinatorial problems in an efficient and flexible way. 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 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.

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.

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

(i) 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, 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.

(ii) 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, 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 optimisation problems within an industrial process [33].

(iii) 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 [28], 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 [39]. 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, backtracking or inheritance, and so on.

We implemented our mathematical model using the finite domain constraint solving library of SICStus Prolog, called clpfd [5]. 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. We used a compiled version of the model. 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 Figure 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 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:

\[
\{\text{SeqLen}, \lambda, \text{Channels}, \text{MinTrigSep}, \text{MinBrushSep}\} = \{7, 3, 3, 3, 1\} \text{ for Exp1a. For Exp1b and Exp1c we use respectively } \{10, 5, 5, 3, 1\} \text{ and } \{20, 10, 5, 1, 1\}. \text{ Since experiment Exp1b and Exp1c are just slight variations of Exp1a, we present only the final results for these, while for Exp1a we also present detailed setup and execution results.} 
\]
Experiment Exp1a uses a minimal configuration with three channels, $j \in [1, 3]$, as illustrated in Figure 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 $\lambda = 3$ rows and, since there are three channels, L becomes a $3 \times 3$ matrix, as shown in Figure 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:

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

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 L 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 $\text{MinTrigSep} = 3$, $\text{MinBrushSep} = 1$, and $\text{MinOverlapTime} = 1$. 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 Figure 8. Figure 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 $C_2$ between events 5 and 6, as shown in Figure 5(b) 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 seconds 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:
We now define the following two sequences of variables from $\mathbb{L}$ in vector form:

$$L = \text{vec}(\mathbb{L}) = (L_{1,1}, L_{1,2}, L_{1,3}, L_{2,1}, L_{2,2}, L_{2,3}, L_{3,1}, L_{3,2}, L_{3,3}, \ldots)$$

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

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

\[ G_1 = \{C, L, B_i, t_i\} \quad G_3 = \{C, L, t_i, B_i\} \quad G_5 = \{C, L, \overline{t_i}, \overline{B_i}\} \]
\[ G_2 = \{C, L^T, B_i, t_i\} \quad G_4 = \{C, L^T, t_i, B_i\} \quad G_6 = \{C, L^T, \overline{B_i}, \overline{t_i}\} \]  (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 \texttt{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:

(i) **(Optimal):** An optimal solution is found and an optimality certificate is obtained within the contract of time, that is, optimality is proven.
Table 1: Summary of results for experiment 1, where we classify the search heuristics into four categories. The timeout was set to 180 seconds for all experiments. A more detailed graphical presentation of Exp1a is given in Figure 10.

<table>
<thead>
<tr>
<th>Search direction</th>
<th>Exp1a</th>
<th>Exp1b</th>
<th>Exp1c</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>up</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 Optimal</td>
<td>24</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>2 Optimal, timeout</td>
<td>6</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>3 Sub-optimal</td>
<td>0</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>4 No solution</td>
<td>42</td>
<td>59</td>
<td>68</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>72</td>
<td>72</td>
<td>72</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>down</strong></th>
<th>Exp1a</th>
<th>Exp1b</th>
<th>Exp1c</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Optimal</td>
<td>20</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2 Optimal, timeout</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>3 Sub-optimal</td>
<td>6</td>
<td>18</td>
<td>2</td>
</tr>
<tr>
<td>4 No solution</td>
<td>46</td>
<td>52</td>
<td>68</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>72</td>
<td>72</td>
<td>72</td>
</tr>
</tbody>
</table>

(ii) **(Optimal, timeout):** An optimal solution is found but no certificate is provided, that is, optimality is not proven.

(iii) **(Sub-optimal):** A sub-optimal solution is found but the search timed out. This means that optimality is neither reached nor proven.

(iv) **(No solution):** No solution is found within the contract of time.

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.

Figure 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 to 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 Figure 10, among the 24 successful heuristics, the time needed to find and prove optimality ranges from 0.7 seconds to 149.4 seconds. Generally speaking, the two graphs in Figure 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.

We found that only two variable selection search heuristics perform acceptably for the three sub-experiments, namely, \((L, B, C, \overline{t})\)
8. Empirical Evaluation

and \((L, \overline{B}, C, \overline{T})\), 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 \((\overline{B}, C, \overline{T})\) 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 \((L^T, \overline{B}, C, \overline{T})\) 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 \(\lambda = (10, 15, 20)\) and \(\text{SeqLen} = (50, 100, 150, 200, 250, 300)\), which yields 18 different configurations. Each configuration was systematically executed using all timeout values in the range \([2, 30]\) seconds, in addition to 60 seconds, 120 seconds, 180 seconds, and 600 seconds. 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

Figure 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 \(\lambda = 10\) in combination with large values of \(\text{SeqLen}\) within the time contract of 600 seconds. For this reason, only three results are reported for \(\lambda = 10\), namely, those where \(\text{SeqLen} \leq 150\). For \(\lambda = 15\) and \(\lambda = 20\), we got results for all combinations of \(\text{SeqLen}\). Note also that all executions, except for \(\lambda = 10, \text{SeqLen} = 150\), provided a sub-optimal solution within
Figure 11: How well the model minimizes the duration of the test sequence, $t_N$, if more time is added to the solving process, $t_s$. 
10 seconds. In fact, most of the executions generated a first solution in less than three seconds.

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 trade-off 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:

$$E = \frac{\text{SeqLen}}{t_N + t_s}$$ \hspace{1cm} (14)

In Figure 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 seconds 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 $\lambda = 10$ and $\text{SeqLen} = 50$. For this case, the first value found is $t_N = 9.99$ seconds and, by running the model an additional 30 seconds, we obtain a solution that executes in only $t_N = 3.05$ seconds, 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.

### 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. Empirical Evaluation

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

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
### 8. Empirical Evaluation

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

<table>
<thead>
<tr>
<th>Bug#</th>
<th>Time in system</th>
<th>Time to solve</th>
<th>Time to validate</th>
</tr>
</thead>
<tbody>
<tr>
<td>44432</td>
<td>5–10 years</td>
<td>1–2 hours</td>
<td>1 day</td>
</tr>
<tr>
<td>44835</td>
<td>5–10 years</td>
<td>2–4 days</td>
<td>1 day</td>
</tr>
<tr>
<td>27675</td>
<td>6–12 months</td>
<td>1–2 months</td>
<td>1–2 weeks</td>
</tr>
<tr>
<td>28859</td>
<td>6–12 months</td>
<td>2–3 months</td>
<td>2–3 weeks</td>
</tr>
<tr>
<td>28638</td>
<td>4–6 months</td>
<td>1–2 weeks</td>
<td>2–3 weeks</td>
</tr>
</tbody>
</table>

- **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.

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.
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 CM2, 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., CM1 and CM2) 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, 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
Table 3: Comparison of constraint-based testing versus current test methods.

<table>
<thead>
<tr>
<th></th>
<th>Activation w/oscilloscope</th>
<th>Paint on paper</th>
<th>Constraint-based test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup time(^a)</td>
<td>1-2 hours(^e)</td>
<td>3-4 hours(^e)</td>
<td>1-2 min(^f)</td>
</tr>
<tr>
<td>Activations per test(^b)</td>
<td>1</td>
<td>5-10</td>
<td>&gt;100</td>
</tr>
<tr>
<td>Repetition time(^c)</td>
<td>5 sec</td>
<td>10 min</td>
<td>&lt;1 sec (^d)</td>
</tr>
<tr>
<td>Interpretation time(^d)</td>
<td>&lt;1 min(^e)</td>
<td>2-4 min(^e)</td>
<td>&lt;1 sec(^f)</td>
</tr>
<tr>
<td>Synchronized with mechanical robot</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Can run standalone after initial setup</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

\(^a\) 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.

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 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 [34] 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, Brand, and Stuckey [16] and Banda et al. [1] 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 [27], 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. After a build server is installed with its continuous integration software, absolutely everything is extracted from the source control repository, as recommended in [15]. 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 seconds, 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

In Table 4 we summarize the notation used in the mathematical model for the IPS.
Table 4: Notation for the parameters in the production model.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Test control parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>The size of the input sequence.</td>
</tr>
<tr>
<td>$i$</td>
<td>The $i$-th sequence, $i \in [1, N]$.</td>
</tr>
<tr>
<td>$j$</td>
<td>Channel number $j \in [1, 5]$.</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>The number of different spray patterns in the model, or entries in the lookup table $L$.</td>
</tr>
<tr>
<td>$e$</td>
<td>A subscript $e$ specifies at which sequence $i$ a scenario should start.</td>
</tr>
<tr>
<td>$c$</td>
<td>A subscript $c$ specifies on which channel $j$ a scenario should appear.</td>
</tr>
<tr>
<td>$MinBrushSep$</td>
<td>The minimum time between two spray pattern changes, $t_i - t_{i-1} \geq MinBrushSep$.</td>
</tr>
<tr>
<td>$MinTrigSep$</td>
<td>The minimum time between two actuator output changes for some channel $j$.</td>
</tr>
<tr>
<td>$MinOverlapTime$</td>
<td>The minimum time an overlap should be in the overlap scenario.</td>
</tr>
<tr>
<td>$BurstTime$</td>
<td>The minimum time a burst of changes should last in the burst scenario.</td>
</tr>
<tr>
<td>$BurstLen$</td>
<td>The number of changes to use in the burst scenario.</td>
</tr>
<tr>
<td>$ IllegalVal $</td>
<td>Value to use for the shutdown scenario.</td>
</tr>
<tr>
<td>$B_i$</td>
<td>The value $i$-th spray pattern in the test sequence.</td>
</tr>
<tr>
<td>$t_i$</td>
<td>The time of the $i$-th spray pattern in the test sequence.</td>
</tr>
<tr>
<td>$Max_j$</td>
<td>The maximum value channel $j$ can have.</td>
</tr>
<tr>
<td>$Min_j$</td>
<td>The minimum value channel $j$ can have.</td>
</tr>
<tr>
<td>$D^+_j$</td>
<td>Parameter used to calculate timing for increasing value of the output on channel $j$.</td>
</tr>
<tr>
<td>$D^-_j$</td>
<td>Parameter used to calculate timing for decreasing value of the output on channel $j$.</td>
</tr>
<tr>
<td>$K_j$</td>
<td>Boolean value deciding whether or not a channel should use linear delay calculations.</td>
</tr>
<tr>
<td>$P_{j,i}$</td>
<td>The activation value for the $i$-th output on channel $j$.</td>
</tr>
<tr>
<td>$t_{j,i}$</td>
<td>The activation time for the $i$-th output on channel $j$.</td>
</tr>
</tbody>
</table>
References


Paper 2:
Using Constraint Programming to Test Paint Control System for Robots

M. Mossige\textsuperscript{1,3}, A. Gotlieb\textsuperscript{2}, H. Meling\textsuperscript{3}

\textsuperscript{1} ABB Robotics, Nordlysvegen 7, 4349 Bryne, Norway
\textsuperscript{2} Certus Centre for Software Verification and Validation, Simula Research Laboratory, P. O. Box 134, N-1325 Lysaker, Norway
\textsuperscript{3} Department of Electrical Engineering and Computer Science, University of Stavanger, 4036 Stavanger, Norway
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. In this context, testing the control system’s timing behavior is particularly challenging, since paint activation must be synchronized with the robot’s 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 process control system’s timing behavior. We designed a constraint optimization model in SICStus Prolog, using arithmetic and logic constraints, in addition to global constraints. The model has been integrated into ABB’s testing process, based on a fully automated continuous integration environment. This allows the model to be solved on demand prior to test execution, which permits us to obtain the most optimal and diverse set of test scenarios for the present system configuration.

The model has been part of ABB’s testing process for 18 months of daily operation and 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, such as Ethernet and dual-port RAM, depending on the delay and latency requirements on their communications. As the complexity of robot control systems continues to grow, developing and validating software for CIRs are becoming increasingly difficult. For robots performing process-intensive tasks such as painting, gluing, or sealing, the problem is even worse, since their dedicated process control systems are loosely coupled with the robot motion control system. A key feature of robotized painting is the ability to precisely activate the process
equipment along a robot’s programmed path. ABB Robotics (ABB), Norway, develops and validates integrated painting control systems (IPS) for CIRs and are constantly improving their testing processes to deliver more reliable products.

Current practices for validating the IPS software involve designing and executing manual test scenarios. However, to reduce the costs of testing and to improve product quality, there is potential in automating the generation of test scenarios and executing them frequently.

In this paper, we report on our use of constraint programming (CP) over finite domains to automatically generate test scenarios in the form of timed event sequences for the IPS and execute them within a continuous integration (CI) process [7]. Building on initial ideas presented by Mossige, Gotlieb, and Meling [21, 20], we developed a constrained optimization model in SICStus Prolog clpfd [3] to help test the IPS under operational conditions.

Due to the IPS’s online configurability, test scenarios must be reproduced on a daily basis. This means that indispensable trade-offs between optimality and efficiency must be found to increase the capabilities of the Continuous Integration process to reveal software defects as early as possible. While using CP to generate model-based test scenarios is not entirely new [3, 14], to our knowledge, this is the first time that a CP model and its solving process have been designed and integrated into a Continuous Integration environment to test complex distributed systems.

In previous work [21, 20], we focused on the software testing aspects of the Continuous Integration process, while in this paper we focus on the CP aspects of the testing process. However, for this paper to be self-contained, we include some content from our previous work pertaining to the background of robotized painting and the IPS. In addition, some of the equations and the notation used in our previous work are changed to better match common CP practices.

The rest of the paper is organized as follows: Section 2 introduces robotized painting along with an example serving as a basis for describing the mathematical relations involved. Section 3 describes ABB’s current IPS testing practices 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 Continuous Integration process. Section 6 presents the output of our model compared to manually
generated test sequences. Section 2 discusses related work on using CP for software validation purposes. Finally, Section 8 discusses some lessons learned, summarizing the impact of using CP in ABB’s industrial context. This latter section also presents ideas for further work.

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 physics of the paint process, that is, controlling the activation and deactivation of several physical processes such as paint pumps, air flows, and air pressure and to synchronize these with the motion of the robot arm. A spray pattern is defined as the combination of the different physical processes, each of which may 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 IPS’s timing behaviors.

2.1 Example of Robotized Painting

We now provide a concrete example of how a robot controller communicates with the IPS to generate a spray pattern along the robot’s path. A schematic overview is shown in Figure 1 where the node marked robot controller is the CPU interpreting a user program and controlling the robot’s servo motors to move it. The example is realistic but simplified to keep the explanations as simple as possible. Figure 1 shows an example user program. The first instruction, MoveL p1, moves the robot to the Cartesian coordinate p1. The next two SetBrush instructions tell the robot to apply spray pattern number 1 when it reaches \( x = 200 \) mm on the \( x \)-plane and to apply spray pattern number 2 when it reaches \( x = 300 \) mm. Both SetBrush instructions tell the IPS to carry out a specific behavior when the physical robot arm is at a given position. The last instruction (PaintL) starts the robot’s movement from the current position \( p1 \) to \( p2 \) and activates the painting process. The \( v800 \) argument to PaintL denotes the speed of the movement, that is, 800 mm/s.
Assume now that the path from $p_1$ to $p_2$ results in a movement in a Cartesian coordinate system from $x = 0$ mm to $x = 500$ mm, that is, a straight line of length 500 mm. The robot controller interprets the user program ahead of the robot’s actual physical movement and, since the robot moves at a constant speed of 800 mm/s here, the robot controller can estimate when it will be at a specific position. If the movement starts at time $t = 0$, the robot controller will compute, based on the robot’s speed, that the two `SetBrush` activations should be triggered at $t_1 = 250$ ms and $t_2 = 375$ ms, where $t_1$ and $t_2$ correspond to $x = 200$ mm and $x = 300$ mm, respectively.

The robot controller now sends the following messages 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. Note that the IPS may receive the second message before the first spray pattern is scheduled for execution, which means that the IPS must handle a queue of scheduled spray patterns.

### 2.2 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 such as paint type or equipment in use. In the IPS, each spray pattern is translated into three to six different physical actuator outputs that must be activated at appropriate times, possibly different from each other.

As exemplified in Figure 1, we may use 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, which is a simple lookup table. In this example, L(B1 = 1) returns \((L_{1,1}, L_{1,2}, L_{1,3})\) while \(L(B_2 = 2)\) results in \((L_{2,1}, L_{2,2}, L_{2,3})\). Given the result of a lookup in \(L\), the IPS master will pass these values to each actuator output along with its activation time \(t_{i}'\), which may be different from the original time \(t_i\) received from the robot controller. This adjustment can be formalized as follows:

\[
t'_{i} = \begin{cases} 
  t_i - \text{PreTime} & \text{if } P_{i-1,1} = 0 \land P_{i,1} \neq 0 \\
  t_i - \text{PostTime} & \text{if } P_{i-1,1} \neq 0 \land P_{i,1} = 0 
\end{cases}
\]  

which shows that the activation time of an actuator output may be adjusted by a constant factor (\(\text{PreTime}, \text{PostTime}\)), depending on changes applied to another actuator’s output. The rationale behind this is that some physical processes can affect the timing of other physical processes. For example, in robotized painting the process of turning on or off the paint fluid affects the timing of the air control process. In our current example from Figure 1, the timing of \(\text{C2}\) is influenced by changes in \(\text{C1}\). This means that the activation time \(t'_i\) sent to \(\text{C2}\) is calculated based on (1), while the activation time sent to \(\text{C1}\) and \(\text{C3}\) is the time \(t_i\) received from the robot controller, that is, the time is forwarded without any adjustments.

### 2.3 Activation of Actuator Outputs

As previously mentioned, many of the physical processes in painting involve slow physical processes, such as starting a pump motor, filling a hose with air, or opening a solenoid valve. To compensate for these physical process delays, the actuator output calculates an adjusted activation time \(t_{i,j}\), that accounts for this time in an effort to overcome the physical process delay. While still referring to Figure 1, we now present how an activation message sent by the IPS master is processed by the receiving actuator output. In the following, we consider the message \((P_{i,j}, t_i)\) sent from the IPS master to actuator output \(j\), whose current actuator output is \(P_{i,j-1}\).
The IPS can adopt two different strategies to compute this compensation time. The first is to adjust the time by a fixed delay, $D^+$ for a positive change and $D^-$ for a negative change. This strategy is typically used for solenoid valves and other actuators that have a fixed process delay. The second strategy uses a linear timing function, where the duration of the compensation is linear with the change in physical output. This strategy is typically used for pumps and air flows. For example, changing the speed of a pump motor from 100 rpm to 300 rpm takes longer than changing it from 100 rpm to 200 rpm.

The following equation combines both the fixed delay and the linear delay strategies into a single compensation function, where the value of $K$ determines whether fixed or linear compensation is to be applied:

$$
t_{i,j} = \begin{cases} 
    t_i - D_j^- & \text{if } P_{i-1,j} < P_{i,j} \land K_j = 0 \\
    t_i - D_j^+ & \text{if } P_{i-1,j} > P_{i,j} \land K_j = 0 \\
    t_i - D_j^- \cdot (P_{i,j} - P_{i-1,j}) \cdot M_j & \text{if } P_{i-1,j} < P_{i,j} \land K_j = 1 \\
    t_i - D_j^+ \cdot (P_{i-1,j} - P_{i,j}) \cdot M_j & \text{if } P_{i-1,j} > P_{i,j} \land K_j = 1 \\
    t_{i-1,j} & \text{otherwise}
\end{cases} \quad (2)
$$

where $M_j = \frac{1}{\text{Max}_j - \text{Min}_j}$ is a constant factor calculated based on the physical minimum and maximum values that can be applied to actuator output $j$. They are determined by the properties of the physical equipment, such as the pumps and valves.

### 2.4 Physical Layout of the IPS

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

### 2.5 Example Summary

To wrap up the example given in Section 2.1, we enrich it with realistic values and visualize both the configuration and the actual output data.
This visualization will be used later to explain error scenarios and describe the results obtained using the IPS’s CP model.

We first specify values for the brush table as follows:

\[
L = \begin{pmatrix}
0 & 0 & 0 \\
100 & 75 & 150 \\
500 & 150 & 175
\end{pmatrix}
\]

This corresponds to a configuration where the robot can use two different spray patterns in addition to the zero spray pattern, which is used to deactivate all the actuators. In this example, \( C_1 \) is the actuator output controlling a paint pump, whose working range is from zero to 500 ml/min. This gives us \( \text{Min}_1 = 0 \) and \( \text{Max}_1 = 500 \). The terms \( C_2 \) and \( C_3 \) are both actuators for air flow control with the limits \( \text{Min}_2 = 0, \text{Max}_2 = 200, \text{Min}_3 = 0, \) and \( \text{Max}_3 = 200 \). For \( C_1 \), we use linear timing compensation \( (K_1 = 1) \), while both \( C_2 \) and \( C_3 \) use a fixed delay \( (K_2 = K_3 = 0) \). In this example, \( C_1 \) affects the timing of \( C_2 \) and thus we set \( \text{PreTime} = 80 \) and \( \text{PostTime} = -50 \). A complete overview of this configuration is shown on the right and bottom portions of Figure 2. The figure also illustrates, on M’s timeline, that the IPS is instructed to change to spray pattern \( B_1 = 1 \) at \( t_1 = 250 \) and then change to spray pattern \( B_2 = 2 \) at \( t_2 = 375 \). The corresponding actuator outputs for the different channels are also shown on separate timelines.

From a tester’s perspective, the challenge in this example is to generate test configurations, decide the order and timing of the different spray patterns, and validate each physical output with the corresponding timing.

In this section, we have provided an overview of how the IPS works together with a robot controller to generate spray patterns along a moving path. We have also described how the IPS can compensate for process delays in an actuator output, with both fixed delay compensation and compensation based on the change of another actuator output. Finally, we have introduced graphical notation to illustrate spray patterns and their corresponding actuator outputs.
3. Testing the IPS

Current IPS testing practices involve considerable manual labor, including setting up physical machinery and collecting observations from manually performed test scenarios. Due to their manual nature, these test scenarios are also costly to perform, even when conducted only once per release cycle. Performing the test scenarios with such a low frequency also results in substantially higher development costs for new versions of the IPS, since correcting software defects late in the development process may require developers to recall source code changes made at early stages of development. Even worse, if a software failure is observed during operation at a customer’s site, the costs are even higher, since corrections may need to take place at the customer’s physical location. In addition to the above concerns, having a distributed control system mounted on a physical robot makes the validation process unnecessarily complex.
3. Testing the IPS Paper 2

3.1 Continuous Integration

The concerns highlighted in the previous section call for a new set of tools to reduce the costs of testing advanced systems, such as IPS. One such tool is Continuous Integration.

Continuous Integration is a software engineering practice aimed at uncovering software defects at the earliest possible stage of development by regularly building the system and executing tests automatically [7]. In this respect, good engineering practice requires developers to submit only small source code changes frequently instead of large sets of changes occasionally. Together with this practice, Continuous Integration has been shown to be very efficient at uncovering defects when developers are geographically distributed or large teams are involved.

3.2 Testing in a Continuous Integration Environment

Compared to traditional software testing, running a test scenario in a Continuous Integration environment has additional requirements. In particular, as pointed out by Fowler and Foemmel [7], controlling the total round-trip time is crucial for successful Continuous Integration deployment. Here, the round-trip time refers to the time it takes for a developer to submit a change to the source control repository and obtain feedback from the build and test processes. Thus, to keep the round-trip time as short as possible, we have identified a few areas where special care must be taken:

- **Test complexity.** In Continuous Integration, a less accurate but fast test is usually preferred over a slow but accurate test. In practice, a test must satisfy the so-called good enough criterion, frequently used in the industry [26].

- **Solving time.** Constraint-based optimization is usually a time-consuming task, especially if a global optimum is sought [17]. Thus, when used in Continuous Integration, a time-contracted optimization procedure is imperative. In other words, it is important to have precise control over the time needed to compute the optimum, potentially sacrificing solution quality.

- **Execution time.** We observe that the test execution time is dependent on the length of the test sequence, that is, the number of test scenarios. This must be accounted for, together with the time needed to generate the test sequence.
In essence, there is a trade-off between the time spent executing a test sequence and the time needed to generate the test sequence (its solving time). Thus, to integrate CP into a Continuous Integration process, we need to strike a balance between the solving and execution times.

4 CP Model of the IPS

We now present the CP model we use to generate IPS test cases. We note that test models, as proposed in model-based testing [27], are usually limited in scope. They are not intended to reflect the full behavior of the system they represent. In this case, we confine ourselves to modeling the IPS’s timing behavior to build an efficient CP model for generating test scenarios. The model will generate configuration and test inputs to the IPS such that the output values with corresponding time values for each actuator output correspond to a given test scenario.

We built two different IPS CP models. One is an industrial strength model, named CM1, which has been integrated into ABB’s CI process. This model includes detailed paint domain knowledge, which makes it hard for a non-expert to understand. Thus, we also developed a simplified model, named CM2, without the paint domain knowledge. This model, which must be considered a pure research model, is also more flexible and can generate test scenarios that are not executable on a real system but demonstrates the principles. Many of the examples in this paper are generated by using CM2. Both CM1 and CM2 can be found in Mossige [19].

4.1 Decision Variables and Domains

For a given test scenario, the CP model will generate a solution comprised of a set of decision variables. The decision variables for the CP model of the IPS is organized into the following four categories:

(i) Variables of the input sequence $I$,

(ii) Configuration variables $C$,

(iii) Variables of the brush table $L$, and

(iv) The expected output of each actuator output with its corresponding time, also known as the oracle $O$. 

105
Formally, the test input sequence $I$ corresponds to $((B_0, t_0), \ldots, (B_N, t_N))$, where $N$ is the length of the test sequence. The domain of $B_i$ and $t_i$ is given as $B_i \in [1, BTabSize]$ and $t_i \in [0, MaxTime]$. The input parameters $N$, $BTabSize$, and $MaxTime$ are typically provided as constants to the model when the solving process is launched.

Given $\alpha$ number of actuators, the configuration variables $C$ contain parameters for the IPS master and for each actuator output, which are used in the timing compensation functions (1) and (2): 

$$C = [PreTime, PostTime, D_1^+, D_1^-, K_1, \ldots, D_\alpha^+, D_\alpha^-, K_\alpha]$$

The domain of the variables in $C$ is given by user-configurable constants to the model at runtime. The typical range used by the IPS is $\pm 500$ ms for $PreTime$, $PostTime$, $D^+$, and $D^-$. The domain of $K_i$ is, of course, $[0, 1]$.

In the brush table $L$, the number of columns corresponds to the number of actuator outputs, $\alpha$, and the number of rows is the user-configurable parameter $BTabSize$. The domain of each row in $L$ is extracted from $Min$ and $Max$ from the corresponding actuator output. During a test scenario, this is done by querying the physical actuator output before the solving process begins.

Finally, for test oracle $O$, a given input $(B_i, t_i)$ produces an output $P_{i,j}$ and a corresponding time $t_{i,j}$ for each actuator output $j$. More precisely, $(B_i, t_i) \rightarrow ((P_{i,1}, t_{i,1}), \ldots, (P_{i,\alpha}, t_{i,\alpha}))$ for $i \in [0, N]$. The domain for $O$ is given by $P_{i,j} \in [Min_j, Max_j]$ and $t_{i,j} \in [0, MaxTime], \forall i, \forall j$.

### 4.2 Additional Constraints

The constraints given in (1) and (2) represent the main calculation of the constraint model, describing the IPS’s timing behavior. However, additional constraints are needed to capture the ordering of events or the relations between the time instances. Specifically, for the input sequence $I$, the time of each input must be higher than the previous input. Specifying the minimal duration between two subsequent time instances is possible through the user-defined parameter $MinBrushSep$:

$$\forall i, \ t_{i+1} - t_i \geq MinBrushSep \land t_i \geq 0$$

(3)
Similar constraints are also specified for the oracle’s variables but, instead, the MinTrigSep configuration parameter is used to specify the separation:

$$\forall i, j, \quad t_{i+1,j} - t_{i,j} \geq \text{MinTrigSep} \land t_{i,j} \geq 0$$  (4)

Finally, we need to consider the IPS’s boundary conditions, that is, the initial state of the IPS prior to test execution and its final state after test execution. Generally, leaving any actuator outputs in a non-zero state after the test has finished is undesirable. Hence, the following constraints are also included in the model:

$$\forall j, \quad B_0 = 0, \quad t_0 = 0, \quad P_{0,j} = 0, \quad t_{0,j} = 0, \quad B_N = 0, \quad P_{N,j} = 0$$  (5)

### 4.3 Test Scenarios

We have identified several distinct test scenarios, three of which are shown in Figure 3. The scenarios overlap and kill brush represent failure conditions, where the IPS is forced into an error state. When generating such scenarios, our objective is to check whether the IPS can respond correctly, for example, by initiating shutdown or generate an error message. The normal scenario represents acceptable behavior and is aimed at validating that 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 aims to find a variable assignment that can drive the execution of the IPS toward the corresponding scenario.

The overlap scenario is used several times throughout the paper, since it is the hardest to find and therefore corresponds to the most difficult objective to solve for the CP model. By referring to (6) and Figure 3(b), the following explains the scenario in more detail. Recall from Section 2.1 that the IPS can queue a sequence of actuator output changes. However, it is possible that an activation message arriving later at the actuator output can result in output activation earlier than an already scheduled message. Formally, this means that a situation is possible where $$t_{i,j} > t_{i+1,j}$$.

There are several ways such an overlap scenario can occur. It can be due to rapid switching between spray patterns, which may be caused by an increase in the robot’s speed, for example, from \(v800\) to \(v1000\). It can also be caused by using a PreTime/PostTime configuration or the use of different configurations of the actuator output (\(D^+, D^-, K\)). The IPS is designed to handle these issues by sending an appropriate error
message to the control system and possibly issue a shutdown. From a
testing perspective, it is important to be able to generate such a scenario
to ensure that the IPS behaves as expected, no matter what caused the
overlap. The constraints for enforcing an overlap on actuator output $j$
between time instance $e$ and $e + 1$ are given by

$$
t_{e,j} - t_{e+1,j} \geq \text{MinOverlapTime} \\
t_{e+1,j} - t_{e-1,j} \geq \text{MinTrigSep} \\
t_{e+2,j} - t_{e,j} \geq \text{MinOverlapTime}
$$

Figure 3: Test scenarios considered as test objectives. The horizontal axis represents
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.

Last, in the *kill* scenario we try to force the IPS into an error state caused
by an actuator output failing. When an actuator output fails, an error message is reported back to the IPS master. Depending on the setup, the IPS can choose different strategies to handle the error state, such as shutting down, issuing a warning, or ignoring the error. In our model, we only make sure that an actuator output is able to enter an illegal state by ensuring that a special value, $IllegalVal$, is produced on a specific actuator output, $t$, at a specific output instance $k$:

$$P_{t,k} = IllegalVal$$ (7)

The model does not cover the actual shutdown behavior, since this is very configuration specific. We found it more appropriate to validate the IPS’s behavior in the part that compares the oracle with the actual actuator outputs.

4.4 Requirements for Test Case Generation

Our first requirement for generating test cases is that they can be reproduced upon every test execution, so as to document failure cases and to help debug the IPS system. The second requirement pertains to diversity in the solution of the CP model. That is, when generating a test case for one of the scenarios, there should be some diversity in the solution, so as to increase the probability of exposing an error-prone scenario. In addition to diversity, we also want to avoid generating useless solutions, for example a solution for which the oracle, $O$, provides actuator outputs that are always zero, irrespective of the other decision variables, $I$, $L$, and $C$. Such a solution has no practical value, since it does not correspond to a possible behavior of the IPS. Furthermore, past experience with debugging the timing behavior of the IPS, has revealed that executions involving multiple concurrent transitions on different actuator outputs, tends to be more error-prone. Hence, it is desirable to specifically target such executions when generating test cases.

A common approach to introduce diversity is to use a pseudo-random generator, however our other requirements makes this difficult. This is since the various decision variables must be tweaked appropriately to avoid generating useless solutions, and to target the types of solutions that are known to be the most error-prone. Thus to satisfy the above requirements, in the following we present several techniques for adding diversity to the relevant decision variables.
4. CP Model of the IPS

4.4.1 Variations in the Input Sequence

When trying to broaden the scope of the solution space, the first thing to consider is to introduce variation in the input sequence to the IPS, denoted \( I = ((B_0, t_0), \ldots, (B_N, t_N)) \). With respect to the solution, the actual time \( t_i \) when a spray pattern is applied is largely irrelevant, as long as the constraint in (3) is satisfied. Thus, we only consider variations in brush selection, \( B_i \).

An obvious rule is to ensure that any pair of consecutive brush selections differ, i.e., \( \forall i, B_i \neq B_{i+1} \). But this does not guarantee that all the brushes in \( L \) will be tested. However, we can improve the diversity by using the global constraint `nvalue`, to require that all elements in \( L \) are used at least once: `nvalue(BTabSize, B)`. Unfortunately, this won’t produce the desired diversity in using different elements from \( L \), as illustrated with the following example: Consider an input sequence with \( B = B_1, \ldots, B_{10} \) and \( L \) of size 4. By using `nvalue(BTabSize, B)` in combination with \( B_i \neq B_{i+1} \), a typical solution may be: \( B = [1, 2, 1, 2, 1, 2, 3, 4] \). As the example shows, elements 1 and 2 are over-represented due to the \( B_i \neq B_{i+1} \) constraint.

In order to avoid such brush sequences, we need a way to measure the diversity, and hence we define the notion of diversity entropy (\( DE \)) as follows: Given a sequence of integers, \( DE \) is the product of the number of occurrences of each value in the sequence. For example, \( DE([0, 1, 0, 1, 0, 1, 2, 3]) = 4 \cdot 4 \cdot 1 \cdot 1 = 16 \), 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 lower diversity entropy than the second solution does.

We therefore propose another approach in which we use the global_cardinality constraint to improve the diversity entropy. By specifying the minimum number of times an index of \( L \) must appear in the input sequence, we increase the solution’s diversity entropy. If we now repeat the previous example but use `global_cardinality`, we obtain the following:

```plaintext
global_cardinality([B1, ..., B10], [1-N1, 2-N2, 3-N3, 4-N4])
N1 #>= Ob, N2 #>= Ob, N3 #>= Ob, N4 #>= Ob
```
A typical solution here would be \( B = [1, 2, 1, 2, 1, 2, 3, 4, 3, 4] \) for \( Ob=2 \). This implementation approach has been shown to be flexible enough to consider solutions with a satisfactory \( DE \). In our actual implementation, the parameter of \( Ob \) has been made available as a configurable parameter that can be set by the user.
4.4.2 Varying the Brush Table Entries

In order to exercise the entire operating area of each of the actuator outputs, it is also important to vary the entries of the brush table. When validation engineers create these tables manually, they try to ensure that the entire operating area of the actuator output is used, including that both Min and Max of each actuator output are part of \( L \). They also try to ensure that multiple transitions on the actuator outputs occur simultaneously. That is, some actuator outputs transitions from high to low, while other actuator outputs transition from low to high.

By considering the entries in \( L \) as points in a hypercube of \( \alpha \) dimensions, \( \mathbb{R}^\alpha \), we would prefer each point in the cube to have as large a Euclidian distance as possible from the other points in the cube, that is,

\[
\text{maximize}(\text{minimum}(|L(i) - L(j)|)) \quad \forall i, j, i \neq j)
\]  

However, we observed that this approach is too costly to compute in practice and we prefer a more light-weight approach. Since each column of \( L \) corresponds to one specific actuator output, we ensure that the maximum and minimum of the actual actuator output are present in that column through the use of the global constraints maximum and minimum. In addition, the all_min_dist \([24]\) constraint is used to ensure that all the values in that specific column are spread out by a user-configurable factor. Together these three constraints ensure that each column in \( L \) exhibits reasonably good variation.

To introduce variation between the row entries in \( L \), additional constraints are used to make sure that at least one transition exists where all except one value is changed. For example, from Figure[7] if two entries are \([L_{1,i}, L_{2,j}, L_{3,i}]\) and \([L_{1,j}, L_{2,j}, L_{3,j}]\), then \( i \) and \( j \) should exist such that \( L_{1,i} < L_{1,j} \land L_{2,i} \geq L_{2,j} \land L_{3,i} \geq L_{3,j} \land \) and so on for other entries. This approach does not maximizing the distance between each entry in a hypercube, but it turns out to perform fairly well together with the scenarios presented earlier. Further improvements on this is left for future work.

4.4.3 Variation in Configuration Values

The configuration generated for a specific test scenario includes 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 $D^+ \neq D^- \land D^+ \neq 0 \land D^- \neq 0$, we offer an opportunity for the CP model to introduce diversity in the configuration values as well. By using the global constraint all_different ($D^+_1, \ldots, D^+_C$), and so on, we also enforce diversity between the actuator output values. For the PreTime and PostTime values, a similar strategy is employed: $PreTime \neq PostTime \land PreTime \neq 0 \land PostTime \neq 0$. It is worth noting that these variation strategies have served the good enough principle well, since introducing diversity is important but not at the cost of losing efficiency.

4.5 Implementation Details

This section briefly reviews some of the implementation details we used for our model. For lower-level details related to implementation, we mainly refer to Mossige [19]. We start by describing how the mapping from $B_i$ to $P_{i,1}, \ldots, P_{i,q}$ is carried out before presenting how the model can choose between the use of fixed delay and linear delay. From our example in Section 2.1, we have:

$$L = \begin{pmatrix} L_{1,1} & L_{1,2} & L_{1,2} \\ L_{2,1} & L_{2,2} & L_{2,2} \\ L_{3,1} & L_{3,2} & L_{3,2} \end{pmatrix}$$

By using of the global constraint element [29], we can now easily express the physical output value for each actuator output, as follows:

- $\text{element}([L_{1,1}, L_{2,1}, L_{3,1}], B_i, P_{i,1})$,
- $\text{element}([L_{1,2}, L_{2,2}, L_{3,2}], B_i, P_{i,2})$,
- $\text{element}([L_{1,3}, L_{2,3}, L_{3,3}], B_i, P_{i,3})$.

The model provides the choice between using linear delay or fixed delay for the actuator outputs, unless specified differently by the user. If we consider actuator output $m$, the activation time $t_{n,m}$ is given by (2). As we can see from (2), there are five possible outcomes: two in which the physical output value increases, two in which it decreases, and finally one outcome in which there is no change. In the latter case, the time value
is kept to the last value. In our implementation, all five outcomes are calculated and the model selects one outcome based on reification.

4.6 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 given constraints is the most interesting. The optimal solution here means a sequence of timed events \( I = ((B_0, t_0), \ldots, (B_N, t_N)) \) with the \textit{shortest execution time}, that is, where \( t_N \) is minimized. This means that we increase the number of tests that can be executed in a limited period. Of course, finding the global minimum of \( t_N \) 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 the test sequence above, we see that the test’s execution time can be roughly estimated to be \( t_N \). This means that the total time used is roughly \( t_s + t_N \), where \( t_s \) corresponds to the model’s solving time.

To illustrate the benefit of minimizing \( t_N \), we provide a small example. Figure 4 shows two different solutions for the same test objective. The test objective is to generate an overlap on \( C2 \) between outputs 4 and 5. The upper part of the figure shows one possible solution without minimizing \( t_N \). In the lower part, we show a minimized solution, which in this case is also an optimal solution where \( t_N \) has reached a global minimum. As shown in this example, the execution time goes from \( t_6 = 13 \) to \( t_6 = 10 \), which means that the use of optimization can provide significant shorter execution times for test sequences.

Knowing that the constrained optimization model tends to minimize \( t_N \), the goal is therefore to manage the time needed to find an optimal solution. CP offers means to control the optimization time taken by using a \textit{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 find this option very useful in finding a compromise between the time spent on solving and the time spent on the test’s execution.
4. CP Model of the IPS

![Diagram of C3, C2, C1, and M with equations and parameters]

**Figure 4:** Two solutions to the overlap scenario from Figure 3(b) generated by using the CM2 model. The configuration and parameters with fixed values are shown at the bottom. On the right of each graph, we list both parameters and variables found by the solver. The upper graph shows a non-optimal solution, while the lower graph shows an optimal solution with respect to $\text{minimize}(t_6)$.

MinBrushSep: 1
MinTrigSep: 2
MinOverlapTime: 1
PreTime: 0
PostTime: 0

$D_1^3 = -3$
$D_1^4 = -3$
$K_3 = 0$
$D_2^3 = -3$
$D_2^4 = 2$
$K_2 = 1$
$D_3^3 = -3$
$D_3^4 = -3$
$K_1 = 0$

$L = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 3 & 3 \\ 2 & 4 & 3 \\ 3 & 1 & 4 \end{pmatrix}$
4.7 Search Heuristics

When searching for solutions, many heuristics can be used in CP. Observing the absence of evident structure in our CP model, we considered variable ordering as the first element to examine systematically. To extract useful information, we considered 72 distinct static variable orderings depending on rearrangements of the decision variables \((I, L, C)\) [20]. In addition to this systematic exploration, we used as a reference two well-known dynamic variable orderings, namely, first-fail and first-fail constraint [3]. We also tested both the ascending and descending search directions of the domain. This analysis and the experiments revealed two points:

(i) While both first-fail approaches are effective for sequences with few events, they fail for long sequences.

(ii) For search heuristics with static variable orderings, we can group the result into three groups \(H_1, H_2, \text{ and } H_3:\)

\[
\begin{align*}
H_1 &= (C, L, I') \ (L, C, I') \ (C, L^T, I') \ (L^T, C, I'), \\
H_2 &= (B, L, C, T) \ (B, L^T, C, T) \ (L, B, C, T) \ (L^T, B, C, T) \\
H_3 &= \text{the other 64 tested combinations}
\end{align*}
\]

where \(I' = (t_1, B_1, t_2, B_2, \ldots), B = (B_1, B_2, \ldots), T = (t_1, t_2, \ldots), \) and \(L^T = \text{transp}(L).\)

\(H_1\) is the only heuristic able to produce a solution within an acceptable timeframe for small brush tables combined with long test sequences, for example, \(B\text{TabSize} = 10, SeqLen = 200.\) For configurations of long test sequences combined with large brush tables, for example, \(B\text{TabSize} = 40, SeqLen = 600, H_2\) is the only heuristic able to generate a solution within a reasonable time. The result for \(H_3\) is either no solution at all or a solution only for very small instances.

4.8 Performance of the Variable Heuristics

We now discuss the model’s performance, including a deeper discussion of the heuristics used and how they behave with and without optimization. Recalling that \(H_1\) and \(H_2\) represent two different groups of variable orderings with similar performance, the difference between \(H_1\) and \(H_2\) regarding total execution time, \(t_s + t_N,\) is shown in Figure 5(a) with two different brush table sizes. This experiment reveals that
(i) \(H_1\) produces a relative small value for \(t_N\) by using more time than \(H_2\) does.

(ii) \(H_2\) produces a larger value for \(t_N\) but is faster than \(H_1\).

We also compare \(H_1\) and \(H_2\) when minimizing the overall time of a test sequence, that is, \(\text{minimize}(t_N)\). Figure 5(b) shows that \(H_2\) provides the first solution faster than \(H_1\) and that the quality of the solution is better when more time is allocated to the optimality search. For \(H_1\), Figure 5(b) shows that there is no gain in minimizing \(t_N\).

For the setup where the test sequence is of length \(N = 100\) and the brush table is of size 10, we see that the solver must run for \(\approx 60\) s before \(H_2\) gives a smaller \(t_N\) than for \(H_1\). For the setup of \(N = 200\) with a brush table of size 40, the solver must run for \(\approx 600\) s before break-even occurs.

From the results of Figure 5(a) and Figure 5(b), no strong conclusion can be drawn when it comes to selecting between \(H_1\) and \(H_2\). If a test sequence is generated for multiple uses, that is, the same test sequence is reused 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 in the Continuous Integration process, then using \(H_1\) should be preferred, considering than the total time \(t_s + t_N\) is the actual target of our test generation and execution procedure. Consequently, at
4. Evaluating Heuristics

We now give a deeper explanation of the difference between $H_1$ and $H_2$. We look at both the solution provided without the use of optimization and solutions provided when using optimization. Recall from Section 4.7 that the end of the sequence of variables is as follows:

$$H_1 = (\ldots, t_{N-2}, B_{N-2}, t_{N-1}, B_{N-1}, t_N, B_N)$$
$$H_2 = (\ldots, t_0, \ldots, t_{N-1}, t_N)$$

The variables related to the brush table and configuration parameters are located at the start of the sequence. We observe that the values selected for both groups of variables will be identical for both $H_1$ and $H_2$ in the non-optimized solution. The values are determined solely based on the diversity constraints, as explained in Section 4.4. The main difference between $H_1$ and $H_2$ is the way the values of $B_i$ and $t_i$ are selected.

4.9.1 $H_1$ Heuristics with No Optimization

For the $H_1$ ordering of the variables, the values of $t_i$ and $B_i$ are assigned by alternating between $t_i$ and $B_i$, that is, $\ldots, t_{N-1}, B_{N-1}, t_N, B_N$. In Figure 6, we gave an example of how the last phase of the search with the use of $H_1$ heuristics will look like. In this example, the brush table and configuration parameters were already assigned values earlier in the search process. This means that the “shape” of the next spray pattern is given just by selecting the next $B_i$. We now assume that the next variable to label is $t_3$. Since $t_2 = 5 \land MinBrushSep = 1$, we will obtain $t_3 = 6$ as the first possible legal value $t_3$ can have. As further shown in Figure 6, the initial domain of $B_3$ is 1, 2, and 3. However, $B_3 = 1$ will be removed from the domain since that would result in $t_{2,2} = 6 \land t_{3,2} = 6$, which breaks the $MinTrigSep = 2$ constraint. $B_3 = 2$ will also be removed from the domain, since it will lead to $t_{2,2} = 6 \land t_{3,2} = 7$, which also breaks the $MinTrigSep = 2$ constraint. This leaves us with $B_3 = 3$. This spray pattern breaks none of the $MinTrigSep$ constraints and the labeling can
continue where $t_4$ is labeled. As we see in this small example, the value of $t_i$ will be assigned the smallest possible value based on the available $B_i$. However, if the selected value for $t_i$ results in an empty domain for $B_i$, backtracking will occur and a larger value of $t_i$ is tried.

![Diagram](image)

**Figure 6:** Labeling by using the $H_1$ heuristics. If $t_3 = 6$, we see that the domain of $B_3$ is reduced from [1,2,3] to [3] and thus $H_1$ selects $B_3 = 3$.

### 4.9.2 $H_2$ Heuristics with No Optimization

For $H_2$, the values of $B_i$ are assigned early in the search process and will be considered fixed in this example. In the example in Figure [4](#), we consider the point in time where $H_2 = (\ldots, B_3 = 1, B_4 = 2, B_5 = 3, \ldots, t_0 = 0, t_1 = 1, t_2 = 5, t_3 = ?, t_4 = ?, t_5 = ?)$. To label $t_3$ is only a matter of selecting a high enough value to make sure that the $MinBrushSep$ and $MinTrigSep$ constraints are maintained. Given that $B_3 = 1$, we see that to not break any of the $MinBrushSep$ or $MinTrigSep$ constraints, the first possible value is $t_3 = 8$.

### 4.9.3 $H_1$ Heuristics with Optimization

We now change to using $H_1$ with optimization ($\text{minimize}(t_N)$), as shown in Figure [5(b)](#). We observe that after the first solution is found, the value of $t_N$ is not reduced much, since more time is used to find a more optimized solution. In the example shown in Figure [5(b)](#), the only difference between the first solution provided and the last (when the search is stopped) is
the ordering of the $B_i$ values with the corresponding $t_i$ values. Since $C$ and $L$ are at the top of the search space, the backtracking will take a long time to reach those values and thus provide a potential better result; this heuristic will not perform well if we require a significantly lower value of $t_N$ at the end of the search compared to the initial value of $t_N$. In the example given in Figure 5(b), the backtracking never reaches far enough back to try other values for $C$ of $L$.

### 4.9.4 $H_2$ Heuristics with Optimization

The $H_2$ heuristic has the $C$ variables assigned just before the $t_i$ values. As explained in Section 4.9.2, when assigning a value to the next $t_i$, the smallest value is selected based on the available domain of $B_i$. By backtracking further back in the search tree, the first possible value that impacts $t_N$ consists of the configuration parameters in $C$. In the example shown in Figure 5(b), the values of actuator configurations ($D^+$, $D^-$ or $K$) is changed in each step of the graph. At the time the search is stopped, the difference between the first solution found and the solution at the point where the search is stopped consists of only changes in $t_i$ and $C$.

![Figure 7: Labeling by using the $H_2$ heuristics. Since $B_3 = 1$ from earlier in the search, the only possible value for $t_3$ that does not violate the $\text{MinBrushSep}$ and $\text{MinTrigSep}$ constraints is $t_3 = 8$.](image)

### 4.9.5 Heuristics $H_1$ and $H_2$

We have described some of the differences between $H_1$ and $H_2$. While $H_1$ provides a rather good initial solution, it will not improve much if more time is allocated to the search. This is explained by the fact that the initial $t_i, B_i$ ordering is quite good. To achieve a better solution requires a change in $C$ and $L$. However, for larger test sequences, those variables
are too far away in the search tree to be reached within a reasonable time. For $H_2$, we also explained why a first solution is reached quite quickly by just giving a value to the $t_i$ variables. When optimizing with the use of $H_2$, the improvements are gained when backtracking reaches the values in $C$.

As we have seen by the use of both $H_1$ and $H_2$, the values of $L$ will never change in the experiments we performed. The values of $L$ are simply given by the constraint for variation as explained in Section 4.4. For practical use at ABB, the use of $H_1$ and $H_2$ provides a good balance and covers more than what is expected or required in an industrial setting.

5 Implementation and Exploitation

This section details our implementation of the CP model \cite{19} with SICStus Prolog and its clpfd library \cite{3} and its exploitation in the Continuous Integration process at ABB. It also provides insights into the rationale behind the selection of CP instead of other possible techniques.

5.1 Selection of CP and the CP Solver

The IPS’s mathematical model could have been implemented with techniques other than CP, including SAT or SMT solving \cite{22}, local search techniques for test data generation \cite{18}, or mixed integer programming (MIP) \cite{13}. We briefly review the reasons why we discarded these other techniques:

(i) The selected technique had to be flexible enough to accommodate the many alternatives in the IPS’s dynamic configuration. CP offers a higher degree of flexibility to handle disjunctive constraint systems by allowing the use of backtracking, reification, or constructive disjunction \cite{25}.

(ii) Time-constrained optimization was essential in our industrial context to accommodate the Continuous Integration process. SAT and SMT solving are very efficient in handling Boolean and theory-based satisfiability problems \cite{22}, but they are not tuned to solve optimization

\footnote{Note that no general claim is made, just specific claims to illuminate our choice of CP in the case of validating the IPS.}
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 in discrete combinatorial problems.

(iii) Since the model is used to predict the IPS’s expected outputs, the use of exact methods was mandatory. Despite the efficiency of local search techniques for test data generation [18], the absence of guarantee on the satisfiability of the constraints (e.g., no possible detection of unsatisfiability or guarantee on the determination of satisfiability for complex constraint sets) was sufficient to discard these techniques.

(iv) The input formats of the constraint solver had to be easily tunable to accommodate the high-level tuning of IPS parametrization. SAT and SMT solvers adopt specific formats as inputs (e.g., SMTLIB formats), while CP solvers are usually hosted by a programming language (e.g., Prolog, Java, or C++) that includes high-level programming features such as predicate/method invocation, recursivity, and inheritance.

(v) The availability of global constraints to implement diversity in the 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 we decided to use it as the back end and Python 2.7 as the front end.

5.2 Overall Implementation

The complete system contains around 2000 lines of Prolog code, 300 lines of C code (an interface DLL between Python and SICStus), and, finally, around 3000 lines of Python code. A schematic overview of the implementation and how it is executed can be found in Figure 8.

The modeling part of the project began in early 2013, at the beginning, just using the user interface of SICStus. In April 2013, the first running model was available on a desktop for the generation of the IPS’s test sequences, running over a single embedded board. In May 2013, the
model was integrated into the source control repository and the first automatic test was run in a full Continuous Integration environment. From May 2013 to October 2013, the system was further extended to cover testing over complete distributed systems (i.e., several embedded boards) of the IPS. Today, the model is used in the Continuous Integration process and solved daily. It generates test sequences for 11 different physically embedded IPS boards. For testing in the fully distributed setting, we currently run the model on a 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 in each individual Continuous Integration cycle. This means that these test scenarios are executed at least once every 24 hours.

5.3 Model Execution

Test execution is typically triggered by a build server upon a successful build of the IPS software. These steps are illustrated in Figure 8 and explained as follows:

(i) **Build.** The software build is scheduled to run every night or a developer can manually trigger a build.

(ii) **Upgrade.** All connected embedded controllers are upgraded with the newly built software. Failure at this step aborts the complete cycle.
5. Implementation and Exploitation

(iii) **Configure.** A configuration fetched from the source control repository is loaded into the IPS. The configuration describes the interconnections of the embedded boards and the properties of the specific paint setup.

(iv) **Query and solve the model.** Data retrieved from the IPS together with a test objective is fed into the CP model. This means reading the Min and Max values from each actuator output in addition to the upper and lower legal limits for all the $D^+$ and $D^-$ parameters. This enables us to keep the generated test in sync with changes in the newly built software, changes in the configuration, and even changes in the physical lab equipment.

(v) **Run the test.** Finally, the actual test is executed by first applying the configuration parameters generated by the model ($C$ and $L$) to the physical setup and then applying the generated test sequence ($I$). The test is completed by comparing the times and physical outputs of each actuator output with the model generated oracle, $O$, and report the results back to the software developers.

5.4 Using the Flexibility of CP

As described in the previous sections, we 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 the variables has been a key factor in satisfying our industrial requirements. However, the CP model can also handle specific parameter values, directly provided by the validation engineers, who do not have a strong knowledge of CP. This is simply implemented by guarding the posting of each constraint with groundness conditions. For example, recall Section 4.4.3 where we described how we introduce variations in $C$. We also described that in some setups validation engineers require the use a very specific set of configuration values. This can easily lead to unwanted errors. As an example, suppose a validation engineer wants to use $D^+ = 0, D^- = 0$ for a specific actuator output. However, the constraints for enforcing variation require $D^+ \neq D^-$. Thanks to the Prolog commodity, our Python front end can provide value to any variable in the model and prevent spurious constraints that would slow down the solving process or prevent a solution. This solution has also been shown to help validation engineers achieve a better understanding of how the model works. Generally, validation
6. Evaluation

This section compares our model with manually generated test sequences. To perform a controlled experiment, a reference group (RG) was constituted, composed of five highly skilled development engineers, field engineers with a deep domain knowledge of IPS and robotized painting. The same task with the same test objectives was given to both the CP model and the RG. Unfortunately, even for small test sequences, it was impossible for the RG to generate test sequences that could be compared with the results provided by the CP model, so we had to reduce the ambition of this controlled experiment, as explained below.

6.1 Setup, Constraints, and Test Objectives

As explained in Section 4.3, the overlap scenario is an important test objective to apply to the IPS. We also explained that an overlap scenario can be caused by several different reasons. In our experiment, we identified three different basic ways the overlap can be introduced. The first two are by fixed and linear delay, respectively, in an actuator output, and the last is when one actuator output influences another actuator output through (1).

Our initial plan was to make a one-to-one comparison between how the CP model generates a test sequence compared to how the RG does the same task with exactly the same input constraints. However, it became quickly apparent that it is impossible for a human to generate the same test sequences as the CP model does within a reasonable amount of time. None of the members in the RG were able to follow all the constraints given. We even gave the RG a tool based on an MS Excel sheet with which they could work by trial and error while watching the result on a graph similar to that shown in Figure 2.

We therefore decided to give the RG lesser requirements with respect to the constraints than those used in the CP model. The following requirements and setup were common to both the RG- and the CP-generated test sequences:
The basis system is a three-channel system based on the setup found in Figure 1.

\( C_2 \) is influenced by \( C_1 \) through (1).

The minimum and maximum values for \( C_1 \), \( C_2 \), and \( C_3 \) are, respectively, zero and four.

The timing separation between each input in the test sequence, \( t_i \), should be at least one; \( MinBrushSep = 1 \).

The timing separation between each actuator output, \( t_{ij} \), should be at least two; \( MinTrigSep = 2 \).

The complete test sequence should be of size seven; \((B_0, t_0), \ldots, (B_6, t_6)\).

There should be an overlap on \( C_2 \) between \( B_3 \) and \( B_4 \), as shown in Figure 3(b), and the size of the overlap should be at least one, \( MinOverlapTime = 1 \), as in (6).

Initially, we also had the following requirements for both the RG- and CP-generated sequences, but we ended up setting the following constraints as absolute to the CP model while merely encouraging the RG to follow them:

- The test sequence should be as short as possible; minimize(\( t_7 \)).
- The whole range of each actuator output should be present in \( L \); that is, each row in \( L \) should contain zero and four.
- None of the entries in \( L \) should be identical; \( L(i) \neq L(j), \forall i, j, i \neq j \).
- All the entries in \( L \) should be used in \( I \).
- The configuration parameters (\( C \)) should have diversity, as described in Section 4.4.3.

With the constraints given, we asked the RG and the CP model to generate three different test sequences with the following objectives:

(i) To generate an overlap whose source is due to a fixed delay in \( C_2 \) and no use of \( PreTime/PostTime \), which means that the following parameters were given a fixed value: \( K_2 = 0, PreTime = 0 \) and \( PostTime = 0 \).
To generate an overlap whose source is due to a linear delay in \( C_2 \) and no use of \( \text{PreTime/post} \), which means that the following parameters were given a fixed value: \( K_2 = 1, \text{PreTime} = 0 \) and \( \text{PostTime} = 0 \).

To generate an overlap whose source is due to the use of \( \text{PreTime/PostTime} \), which means that the following parameters were given a fixed value: \( D^+_2 = 0 \) and \( D^-_2 = 0 \).

### 6.2 Results

For reference, we added the solution provided by the CP model and one typical solution provided by the RG. Figure 9 shows the solution for test objective 1 (overlap caused by a fixed delay). Figure 10 shows the solution for test objective 2 (overlap caused by linear delay). Finally, Figure 11 shows the solution for test objective 3 (overlap caused by \( \text{PreTime/PostTime} \)).

Table 1 summarizes the results given by the CP model with the average of what the RG generated.

The conclusions we can draw from this experiment are as follows:

(i) None of the RG solutions were able to fulfill all the required constraints when given a limited amount of time to set up the test sequence. Even when provided with an Excel sheet that allowed for an approach based on trial and error.

(ii) The diversity of \( L \) in the manual solutions was acceptable. However, it must be noted that since \( L \) is small (3x4) and the span of possible values is small (\( Min =0, Max =4 \)), it is not a difficult task to manually generate \( L \) so that diversity is maintained.

(iii) The largest problem for the RG was to maintain the relevant configurations of the two other actuator outputs (\( C_1 \) and \( C_3 \)). The validation engineers favored setting all the parameters of \( C_1 \) and \( C_3 \) to zero to reduce the complexity of the problem.

(iv) None of the manually generated sequences obtained an optimal result with respect to \( t_N \).
Figure 9: Top: The solution generated by CM2 for test objective 1, overlap caused by fixed delay in C2. Bottom: One of the solutions generated by the RG. As we can see, the manual solution sets all parameters for C1 and C3 to zero. For both examples, PreTime = 0 and PostTime = 0.
6. Evaluation

\[
L = \begin{pmatrix}
0 & 0 & 0 \\
1 & 2 & 3 \\
2 & 1 & 4 \\
4 & 4 & 4 
\end{pmatrix}
\]

\[
t_0 = 0 \\
B_0 = 0 \\
t_1 = 2 \\
B_1 = 1 \\
t_2 = 5 \\
B_2 = 3 \\
t_3 = 7 \\
B_3 = 0 \\
t_4 = 9 \\
B_4 = 3 \\
t_5 = 11 \\
B_5 = 2 \\
t_6 = 13 \\
B_6 = 0
\]

Figure 10: Top: The solution generated by CM2 for test objective 2, overlap caused by linear delay in C2. Bottom: One of the solutions generated by the RG. As we can see, the manual solution sets all the parameters for C1 and C3 to zero. For both examples, PreTime = 0 and PostTime = 0.

128
$D^+_1 = -1$
$D^-_3 = -2$
$K_3 = 1$
$D^+_2 = 0$
$D^-_2 = 0$
$K_2 = 0$
$D^+_1 = -2$
$D^-_1 = -3$
$K_1 = 1$

$D^+ = 0$
$D^- = 0$
$K = 0$

$L = \begin{pmatrix}
0 & 0 & 0 \\
1 & 2 & 3 \\
2 & 1 & 4 \\
4 & 4 & 1 
\end{pmatrix}$

$PreTime : -2$
$PostTime : -3$

$D^+ = 0$
$D^- = 0$
$K = 0$

$L = \begin{pmatrix}
0 & 0 & 0 \\
1 & 0 & 1 \\
3 & 4 & 3 \\
4 & 0 & 4 
\end{pmatrix}$

$PreTime : -2$
$PostTime : -2$

**Figure 11:** Top: The solution generated by CM2 for test objective 3, overlap caused by PreTime/PostTime. Bottom: One of the solutions generated by the RG. As we can see, the manual solution sets all the parameters for C1, C2, and C3 to zero.
## Table 1: CP based versus manually generated test sequences.

<table>
<thead>
<tr>
<th></th>
<th>Fixed delay</th>
<th>Linear delay</th>
<th>Pre/post</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CP Avg. user</td>
<td>CP Avg. user</td>
<td>CP Avg. user</td>
</tr>
<tr>
<td>( t_N )</td>
<td>11(^a) 14.4</td>
<td>11(^a) 15.2</td>
<td>10(^b) 14.4</td>
</tr>
<tr>
<td>Solving time</td>
<td>45s 310s</td>
<td>21s 471s</td>
<td>11s 367s</td>
</tr>
<tr>
<td>( \mathbb{L} ) constraints ok(^b)</td>
<td>100% 60%</td>
<td>100% 80%</td>
<td>100% 80%</td>
</tr>
<tr>
<td>Avg. Euclidian dist. in ( \mathbb{L} ) (^c)</td>
<td>4.1 3.6</td>
<td>4.1 3.9</td>
<td>4.1 3.9</td>
</tr>
<tr>
<td>( \mathbb{C} ) constraints ok(^d)</td>
<td>100% 0%</td>
<td>0% 0%</td>
<td>0% 0%</td>
</tr>
</tbody>
</table>

\(^a\) Proven optimal value.
\(^b\) The number of the solutions able to fulfill the constraints given for \( \mathbb{L} \) as described in Section 6.
\(^c\) How the entries in \( \mathbb{L} \) are separated according to (8).
\(^d\) The number of solutions able to fulfill the constraints given for \( \mathbb{C} \) as described in Section 6.

(v) All the RG solutions were able to use all the entries in \( \mathbb{L} \) and achieved diversity comparable to that of the CP-based solution. However, the sizes of \( \mathbb{L} \) and of the test sequence in our experiment are small and not even close to the numbers used in, for example, Figure 5(b) or the size of the problems used in daily operation at ABB.

## 7 Related Work

If we consider the 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 [6, 4, 15, 2, 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, Schmidt, Porter, and Slaby [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 of our 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
8. Lessons Learned and Further Work

This section concludes the paper by discussing some lessons learned at ABB from our experience with introducing CP in ABB’s standardized Continuous Integration. This section also outlines a couple of ideas for further work.

8.1 CP for Validation Engineers

As previously stated, the validation of robotized painting involves a fair amount of intensive manual work. Therefore, it is necessary to replace
parts of this validation process with automation, which is perceived by validation engineers as a means to strengthen the process. However, this also comes with drawbacks. Two factors must be distinguished: (1) **Automation through the Continuous Integration process**, including the automatic building of software, software upgrades, test execution, and results reporting, and (2) **test generation through the use of CP**, which permits validation engineers to focus on validating other parts of the CIRs.

Point (1) does not have any drawbacks except for the effort required to set up the Continuous Integration 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 and (b) validation engineers are usually reluctant to trust any tool that produces results which are very difficult to compute by hand or with an easily understandable process. Another concern is that many optimization problems require expert knowledge. To reduce the risks, we decided to build a Python front end for our CP model to hide some of the details from the validation engineers. We also organized basic training in CP with simple and understandable examples to facilitate its adoption. Of course, we do not claim that these actions are a recipe for general CP adoption, but we observe that it worked well in the context of ABB's IPS validation.

### 8.2 Actual Defects Found with the CP Model

After the model was put into production at ABB, it immediately detected two new unknown defects related to the IPS's timing aspects. These defects were, however, classified as non-critical, since 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 by 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.

To validate 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 five
defects. This was considered strong justification for the continued use of
the CP model in production.

8.3 Return on Investment with the Use of CP

Computing the return on investment of the use of CP for ABB’s IPS
validation is not easy. One could possibly 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 in the validation process, a factor that is very difficult to measure.
After the introduction of the CP model in production, we observed much
higher confidence among the engineers in the testing framework, which
has increased their appetite to perform necessary code re-factoring. They
are more willing to make critical but necessary 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 the benefits of using CP to be 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.

8.4 Further Work

Gaining a better understanding of the benefits of CP in IPS validation is
crucial in depicting our experience as a success story. For that, we have
to pursue our analysis of the experimental results obtained with different
search heuristics. We also have to multiply the industrial case studies
within ABB to ensure the CP model’s generality.

Introducing diversity in the selection of test sequences is important
in our application to generate appropriate tests. This can be achieved
by using more dedicated global constraints. By creating a global con-
straint that would balance the solutions over the brush table, we can
capture the needs of validation engineers. In our context, balancing would
mean enforcing the absence of systematic repetition and constraining the
number of different values taken on by the variables (in the spirit of the
\texttt{global_cardinality} constraint).

An extension of this work consists of scheduling the test case exe-
cution over the available time in a CI cycle. We feel that this scheduling
problem requires more than a simple application of classical scheduling algorithms and deserves a complete study and the creation of tuned solutions. This is the more prospective extension of our work.

Acknowledgments

We thank the anonymous reviewers of the former conference version of this paper for their fruitful comments and suggestions. This work was realized with partial support of the Research Council of Norway, as part of the industrial PhD of Morten Mossige; ABB Robotics; and Project 8 of the research-based Innovation Centre Certus, dedicated to software validation and verification.

References


References


Paper 3:
Optimal Test Execution Scheduling on Multiple Machines with Resource Constraints

M. Mossige$^{1,3}$, A. Gotlieb$^2$, H. Meling$^3$, M. Carlson$^4$

$^1$ ABB Robotics, Nordlysvegen 7, 4349 Bryne, Norway
$^2$ Certus Centre for Software Verification and Validation, Simula Research Laboratory, P. O. Box 134, N-1325 Lysaker, Norway
$^3$ Department of Electrical Engineering and Computer Science, University of Stavanger, 4036 Stavanger, Norway
$^4$ SICS SE-751 83 Uppsala, Sweden
1. Introduction

Continuous integration (CI) aims to uncover defects in the earlier stages of software development by frequently building, integrating, and testing software-intensive systems. The process may include running integration tests involving real system components. In the last decade, CI has been recognized as an effective and efficient process to improve software quality [9, 15], while keeping verification costs at an acceptable level [19].

As pointed out by Fowler [10], compared to more traditional testing methods, running a test case in CI requires tight control over the round-trip time, that is, the time from when a source code change is committed until the success or failure of the build and test processes is reported back to the developer. By keeping the round-trip time short, CI can serve as an efficient and effective tool for software quality control, especially for distributed development teams.

Abstract:

When testing large-scale systems with hundreds of test cases in a continuous integration environment, it is crucial to minimize the round-trip time, that is, the time from when a source code change is committed until the test results are reported back to the developer. To this end, scheduling as many test case executions as possible in the minimum amount of time is essential to increasing the effectiveness of continuous integration in the software development process.

This paper introduces TC-Sched, a cost-effective method for test execution scheduling on multiple machines with constraints on accessible resources, such as measurement devices or network equipment. The method uses as input a test suite, a set of execution units, and a set of shared resources and produces an execution schedule. The schedule guarantees that each test will be executed once and minimizes the round-trip time, that is, the time it takes to solve the schedule and execute it.

TC-Sched has undergone extensive experimental evaluation on both randomly generated test suites and industry test suites. Our results provide evidence that TC-Sched conducts efficient and effective test execution scheduling and is suitable for deployment in a continuous integration process. To the authors’ knowledge, this is the first time that a fully automated solution to the test execution scheduling problem has been successfully addressed.
Admittedly, the easiest way to minimize the round-trip time is simply to execute as many tests as possible in the shortest amount of time, that is, by running them in parallel on multiple machines. However, in a scenario with hundreds of tests, targeting different machine architectures and operating systems, and that relies on global resources such as costly measurement instruments or network devices with limited availability, limits are placed on the set of tests that can be executed in parallel. Thus, computing a schedule for test case execution that minimizes the round-trip time under these restrictions is a challenging optimization problem. Since test cases have different execution times and use distinct global resources that are locked during test case execution, finding an optimal schedule is almost impossible by hand. The basic approach would require an exhaustive examination of all the possible schedules to select an optimal one, but this approach is, of course, infeasible, since the number of test cases, machines, and resources would quickly exceed a few units.

Automatic test case execution scheduling is crucial in CI, but it has also applications in the context of embedded systems design and validation. When multiple architectures and operating systems are considered, and when the system is tested with real system components with limited resources, the precise scheduling of the execution of test cases becomes important for ensuring product quality. An increasing number of organizations are facing this scheduling challenge due to platform multiplication and the necessity of shortening the time to market of innovative products.

Formally, consider a set of test cases $\mathcal{T} = \{t_1, \ldots, t_n\}$ with corresponding execution durations, $d_1, \ldots, d_n$. These durations can be slightly over-estimated to account for small variations between the different machines available for test case execution. A set $\mathcal{G}$ of global resources $\{g_1, \ldots, g_o\}$, and a set of machines $\mathcal{M} = \{m_1, \ldots, m_m\}$. We use $n$ to denote the number of test cases and $m$ to denote the number of machines available to run test cases. We also have a function $f : \mathcal{T} \rightarrow \mathcal{M}$ which assigns to each test case one or several machines which can execute the test case. We define the optimal test case scheduling (OTS) as the optimization problem of finding an execution ordering and assignment of all test cases to machines, such that each test case is executed once, no global resource is used by two test cases at the same time, and the overall test execution time, $T_t$, is minimized. We define $T_t$ as the time used to solve the schedule ($T_s$) plus the time used to execute the schedule ($C^*$), $T_t = T_s + C^*$.

Such an assignment and ordering (i.e., a solution to the problem) can be described either by a time- discretized table containing a line per
Table 1: Test suite for example.

<table>
<thead>
<tr>
<th>Test</th>
<th>Duration</th>
<th>Executable on</th>
<th>Use of global resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>2</td>
<td>$m_1, m_2, m_3$</td>
<td>-</td>
</tr>
<tr>
<td>$t_2$</td>
<td>4</td>
<td>$m_1, m_2, m_3$</td>
<td>$g_1$</td>
</tr>
<tr>
<td>$t_3$</td>
<td>3</td>
<td>$m_1, m_2, m_3$</td>
<td>$g_1$</td>
</tr>
<tr>
<td>$t_4$</td>
<td>4</td>
<td>$m_1, m_2, m_3$</td>
<td>$g_1$</td>
</tr>
<tr>
<td>$t_5$</td>
<td>3</td>
<td>$m_1, m_2, m_3$</td>
<td>-</td>
</tr>
<tr>
<td>$t_6$</td>
<td>2</td>
<td>$m_1, m_2, m_3$</td>
<td>-</td>
</tr>
<tr>
<td>$t_7$</td>
<td>1</td>
<td>$m_1$</td>
<td>-</td>
</tr>
<tr>
<td>$t_8$</td>
<td>2</td>
<td>$m_2$</td>
<td>-</td>
</tr>
<tr>
<td>$t_9$</td>
<td>3</td>
<td>$m_3$</td>
<td>-</td>
</tr>
<tr>
<td>$t_{10}$</td>
<td>5</td>
<td>$m_1, m_3$</td>
<td>$g_2$</td>
</tr>
</tbody>
</table>

machine or a starting time for each test case and its assignment to a given machine.

Example

By referring to the test suite in Table 1 we present a small scheduling example. Let $T$ be the set $\{t_1, \ldots, t_{10}\}$, $G$ be the global resources $\{g_1, g_2\}$, and $M$ be $\{m_1, m_2, m_3\}$. The possible machines on which each test case in $T$ can run is given in Table 1. This table can be extracted from the analysis of test scripts which usually specify the set of machines capable of execution this test case, in addition to other information related to the test case.

As stated in Table 1 by sharing the same resource $g_1$, test cases $t_2, t_3, t_4$ cannot be executed at the same time, even if their execution is scheduled on different machines. Furthermore, since $t_7$ can only be executed on $m_1$, $t_8$ on $m_2$, $t_9$ on $m_3$, and $t_{10}$ on $m_1$ or $m_3$, we have to solve a complex scheduling problem. One possible optimal schedule in this example is given in Figure 1 where the time needed to execute the test campaign is $C^* = 11$. For this small problem the solving time, $T_s$, can be assumed to be very short, so the total execution time will be $T_t \approx C^*$.

An objective of this paper is to introduce a method for solving this problem efficiently where the efficiency is demonstrated by an extensive
1.1 Existing solutions and related work

To the best of our knowledge, the problem mentioned above has not yet been systematically addressed and existing solutions include much manual work. In industrial settings, validation engineers manually design the scheduling of test case execution by allocating test case execution to certain machines at given times or following a given order. In practice, they manage the constraints as an aggregate and try to find the best compromise in terms of the time needed to schedule precisely the execution of test cases and the benefit of finding a better approximation of the minimum execution time. Keeping this process manual in a CI environment is paradoxical, since every activity should, in principle, be automated [15]. Furthermore, this is a time-consuming and error-prone activity: It is not uncommon for tests to fail during nightly builds because a critical resource is unavailable due to parallel access attempts.

Finally, it is common that a varying number of machines are available for running tests. This means that machines can be added or removed from the machine pool on daily basis.

Nevertheless, naive automated solutions can be proposed to address the test scheduling problem. For each test case to execute, a machine can be selected at random or based on its current minimum timespan. We are not aware of any tools implementing these methods in a CI process, but we have compared the method proposed in this paper with these two naive methods in our experimental evaluation.
Scheduling problems have been studied in other contexts for decades, and an extensive body of research exists. The scheduling domain is divided into several areas ranging from CPU scheduling in operating systems to scheduling of workforces in a construction project. The scheduling problem described above, belongs to a scheduling category named resource-constrained project scheduling (RCPS). Brucker and Knust have formulated RCPS as follows: Given a time horizon \([0, H]\), \(n\) operations, and \(r\) renewable resources, let a constant amount of \(c_k\) units of resource \(k\) be available at any time \(t = 0, \ldots, H\). Let operation \(i\) be processed in \(d_i\) time units. During this time period a constant amount of \(r_{i,k}\) units of resource \(k\) is occupied. Furthermore, consider that precedence constraints given by a set \(A\) of relations \(i \rightarrow j\), meaning that operation \(j\) cannot start before operation \(i\) has completed.

Over the years, RCPS has been addressed by both the constraint programming community (CP) and the operation research (OR) community. Lombardi and Milano point out that neither CP nor OR can claim to be the best approach, they both perform equally well. However, the clear trend in both CP and OR is to solve such problems with hybrid approaches where techniques are integrated, like, for instance, the work by Schut et al. RCPS is considered to be a generalization of machine scheduling problems where job shop scheduling (JSS) is one of the best known. JSS is the special case of RCPS where each operation uses exactly one resource.

1.2 Flexible Job Shop Scheduling

Flexible Job Shop Scheduling (FJSS) is an generalization and extension of the JSS. JSS is defined as a set \(\mathcal{J} = \{J_1, \ldots, J_n\}\) of \(n\) jobs to be scheduled. Each job \(J_i\) consists of a fixed sequence of operations \(\mathcal{O}_i = \{O_{i,1}, \ldots, O_{i,h_i}\}\) where the precedences are given by the order in the set. Each operation \(O_{i,k}\) has to be processed on a predetermined machine \(M_{i,k} \in \mathcal{M}\) where the necessary processing time is given by the machine. To extend the formalism from JSS to FJSS we say that each operation \(O_{i,k}\) can be processed either on a subset \(\mathcal{M}_{i,k} \subseteq \mathcal{M}\) (partial flexibility) or on all machines (\(\mathcal{M}_{i,k} = \mathcal{M}\), total flexibility). The FJSS is known to be NP-hard.
1.3 OTS vs FJSS

While OTS is closely related to FJSS, there are some important differences:

- In OTS, all machines are considered to be uniform. This is a relaxation compared to FJSS, enabling specialized approaches.

- Each job in OTS consists of only one operation, while in FJSS one job can contain several operations, where there are precedences between the operations.

- For OTS, some operations may require exclusive access to a global resource, preventing other operations from running at the same time.

1.4 Contributions of the paper

In this paper, we introduce TC-Sched, a cost-effective method for solving the optimal test scheduling problem. Our method uses the CUMULATIVES [1, 4] global constraint and efficient search techniques from the constraint programming domain. These ingredients are crucial to 1) automatically filtering invalid test execution schedules, that is, schedules that do not respect at least one of the constraints, and 2) searching among possible valid schedules, those that minimize the global test execution time (i.e., makespan). This paper contains an extensive experimental evaluation conducted over both randomly generated test suites and actual test suites used to test two industrial software systems, namely, a robotized painting system and a video conferencing system. The first and most important goal of our experimental evaluation is to demonstrate the scalability of the proposed approach up to CI processes involving hundreds of test cases and tens of machines, which corresponds to a realistic development environment. A secondary objective of our evaluation is to demonstrate the cost-effectiveness of integrating our approach within an actual CI process. To the best of our knowledge, this is the first time the problem of test case execution scheduling has been extended to a software testing problem and a fully automated solution proposed to address it in the context of CI.

1.5 Paper Outline

The rest of the paper is organized as follows: Section 2 provides a formal definition of the optimal test scheduling problem and reviews some
2. Notation and Background

This section starts by giving a formal definition of the OTS problem. We will also present the Cumulatives global constraint, and how a test case execution can be modeled as a scheduling problem. Finally, we will describe how global resources can be expressed as quasi-machines.

2.1 Optimal Test Case Execution Scheduling

Optimal test case scheduling (OTS) is an optimization problem 
\((\mathcal{T}, \mathcal{G}, \mathcal{M}, d, g, f)\), where \(\mathcal{T} = \{t_1, \ldots, t_n\}\) is a set of test cases along with a function \(d : \mathcal{T} \rightarrow \mathbb{N}\) giving each test case a duration \(d_i\); a set of global resources \(\mathcal{G} = \{g_1, \ldots, g_o\}\) along with a function \(g : \mathcal{T} \rightarrow \mathcal{G}\) that describes which resources are used by each test case; and a set of machines \(\mathcal{M} = \{m_1, \ldots, m_m\}\) and a function \(f : \mathcal{T} \rightarrow \mathcal{M}\) that assigns to each test case, a set of machines on which it can be executed. The function \(d\) is usually obtained by measuring the execution time of each test case in previous test campaigns and by over-approximating each duration. Interestingly, this function does not need to be computed by hand, since its computation can be fully automated within the CI process.

The problem addressed in this paper aims to execute each test case once, while minimizing the total execution time of the test cases. That is, to find an assignment \(a : \mathcal{T} \rightarrow \mathcal{M}\) an execution order for each machine to run its test cases.

In its most basic version, the OTS problem requires the following constraints to be strictly enforced.

- **Non-cumulative scheduling**: Two test cases cannot be executed at the same time on a single machine.

- **Non-preemptive scheduling**: The execution of a test case cannot be temporarily interrupted for the running machine to execute another test case instead.
• **Non-shared resources:** When a test case uses a global resource, no other test case using the same resource can be executed at the same time.

• **Machine-independent execution time:** The execution time of a test case is assumed to be independent of the machine on which it is executed on. This is a reasonable assumption for test cases in which the time is dominated by external physical factors such as a robot’s motion, the opening of a valve, or sending an Ethernet frame. Such test cases typically have execution times that are uncorrelated with machine performance (e.g., CPU type, CPU frequency, operating system). In any case, a sufficient over-approximation of the execution time will satisfy the assumption.

There are cases where the OTS problem can be trivially solved. When there is only a single machine available, executing all the test cases in sequence, without regard for their execution order solves the problem. Indeed, the global execution time remains unchanged, whatever the execution order. Similarly, when there are no global resources and when test cases can be executed on any available machine, then a simple round-trip algorithm that consists in allocating the longest test cases first to the available execution machine solves the OTS problem. Note that these cases are trivially handled by our approach presented in this paper.

### 2.2 The Cumulatives Constraint

The **Cumulatives** constraint is a powerful tool for modeling cumulative scheduling of multiple operations on multiple machines, where each operation can be set up to consume a given amount of a resources, and each machine can be set up to provide a given amount of resources. Here, Cumulatives($(O_1, \ldots, O_n), [c_1, \ldots, c_m])$ constrains $n$ (abstract) operations on $m$ (abstract) machines such that the total resource consumption on each machine $m_j$ does not exceed the given threshold $c_j$ at any time $t$. An operation $O_i$ is typically represented by a tuple $(S_i, d_i, E_i, r_i, M_i)$ where $S_i$ (resp. $E_i$) is a variable that denotes the starting (resp. ending) instant of the operation, $d_i$ is a constant representing the total duration

Through the paper, lower-case characters are used to represent constants and upper-case characters are used to represent variables.
2. Notation and Background

Paper 3

of the operation, \( r_i \) is a constant representing the amount of resource used by the operation.

When used in the context of finite domain constraint solving [16], the variables \( S_i, E_i \) and \( M_i \) are integer variables, with values from bounded domains. Typically, \( S_i \) and \( E_i \) have values within \( est_i \ldots let_i \), where \( est_i \) denotes the operation’s earliest starting time and \( let_i \) denotes its latest ending time and \( let_i \geq est_i + d_i \). The variable \( M_i \) is bounded by the number of machines available, \([1,m]\). However, by reducing the domain of \( M_i \) it is possible to force a specific operation to be scheduled on a subset of the available machines, or even to one specific machine.

It is worth noting that this formalization implicitly uses discrete time instants. Indeed, since \( est_i \) and \( let_i \) are integers, a function associating each time instant to the current executed operations can automatically be constructed from this. Formally, if \( j \) represents an instant in time, we have:

\[
r^j_i = \begin{cases} r_i & \text{if } S_i \leq j < S_i + d_i \\ 0 & \text{otherwise} \end{cases}
\]

The constraint \( \text{Cumulatives}\left([O_1,\ldots,O_n],[c_1,\ldots,c_m]\right) \) holds if and only if, for every operation \( O_i \), \( S_i + d_i = E_i \) and, for all instants \( j \), \( r^j_1 + \ldots + r^j_n \leq c_{M_i} \).

In fact, \( \text{Cumulatives}\left([O_1,\ldots,O_n],[c_1,\ldots,c_m]\right) \) captures a disjunctive relation between possible scenarios and applies deductive reasoning to the possible values in the domains of its variables. This constraint provides a cost-effective process for pruning the search space of some of its impossible schedules. When combined with a labeling process, which assigns values to the variables \( S_i \) and \( M_i \), it allows us to find feasible schedules, that satisfy all the constraints.

2.3 Modeling Test Case Execution Scheduling

This section shows how the \( \text{Cumulatives} \) constraint can be used to model a schedule. In this small example, we disregard the use of global resources, and the constraints that some operations can only be executed on a subset of the available machines, since that will be covered in Section 2.4.

By referring to the schedule in Figure [1], we have ten operations \( \mathcal{O} = \{O_1,\ldots,O_{10}\} \) and three available machines. By encoding the data
from Table 1, we get $O_1 = (S_1, 2, E_1, 1, M_1)$, $O_2 = (S_2, 4, E_2, 1, M_2), \ldots, O_{10} = (S_{10}, 5, E_{10}, 1, M_{10})$, $c_1 = 1$, $c_2 = 1$, $c_3 = 1$. Note that each operation has a resource consumption of one and all three machines have a resource capacity of one. This implies that one machine can only execute one operation at a time. Although the Cumulatives constraint can handle all combinations of resource consumptions / resource bounds, we only consider cases where each machine can execute one operation at a time in this paper.

Calling Cumulatives([O_1, O_2, \ldots, O_{10}], [c_1, c_2, c_3]) within a finite domains constraint solver leads to an assignment of $S_1, E_1, M_1, S_2, E_2, M_2, \ldots, S_{10}, E_{10}, M_{10}$ such that the constraints are satisfied. A possible solution is shown in Figure 1.

2.4 Introducing Global Resources

As explained above, global resources corresponding to physical equipment such as valves or air sensors, measurement instruments, or network devices, have limited and exclusive access. In the OTS formulation, two test cases cannot access them at the same time, so that, this needs to be modeled as additional constraints. Note that the use of global resources must not be confused with the resource consumption or resource bounds of operations and machines.

The naive approach to prevent two operations from being assigned overlapping execution times, is to consider constraints over the start and stop times of the operations. For instance, if $O_1$ and $O_2$ can’t be executed at the same time since both require exclusive access to a global resource, then the constraint $E_1 \leq S_2 \lor E_2 \leq S_1$ can be added to the model. However, it is possible to model the same in a more elegant way by using the Cumulatives constraint again.

Referring to the example in Figure 1, there are ten operations to be scheduled on three machines. There are also two global resources, $g_1$ and $g_2$. The basic scheduling constraint is set up as explained in Section 2.3. However, to model the global resources, we can treat each resource as a new quasi-machine shown as $m_{g_1}$ corresponding to $c_{g_1} = 1$ and $m_{g_2}$ corresponding to $c_{g_2} = 1$. For each operation requiring a global resource, we create a “mirrored” operation of the corresponding quasi-machine: $O_{g_1} = \{O'_2, O'_3, O'_4\}$ and $O_{g_2} = \{O'_{10}\}$. Finally, we can express the schedule in one single constraint: Cumulatives($O \cup O_{g_1} \cup O_{g_2}, [c_1, c_2, c_3, c_{g_1}, c_{g_2}]$).
3. The TC-Sched method

This section describes our method, coined TC-Sched, for solving the OTS problem. It is a time-constrained cumulative scheduling constraint-based technique because 1) it allows us to keep fine-grained control on the time allocated to the constraint solving process (i.e., time-constrained), 2) it encodes exclusive resource use with constraints (i.e., constraint-based), and 3) it solves the problem by using the Cumulatives constraint. The TC-Sched method is composed of three elements, namely, the constraint model described in Section 3.1, the search procedure described in Section 3.2, and the time-constrained minimization process described in Section 3.3.

3.1 Constraint Model

In our framework, we encode the OTS problem with a single Cumulatives(O,C) constraint, using the scheme introduced in Section 2.4 and a search procedure able to find an optimal schedule among many feasible schedules.
Each test case $t_i$ is encoded by an operation $(S_i, d_i, E_i, 1, M_i)$ where the variables $S_i$ and $E_i$ are the starting and ending times, respectively, of the test case execution; the constant $d_i$ is the duration of the test case; and the variable $M_i$ represents the actual machine that will be used to schedule the execution of test cases. Suppose there are three machines available for test case execution, numbered 1, 2, and 3; then, to say that $t_i$ can be executed on any machine, we just add the domain constraint $M_i \in \{1, 2, 3\}$, whereas to say that $t_i$ can only be executed on machine 1, we replace $M_i$ by $\{1\}$.

Using the approach of treating global resources as quasi-machines, each global resource $g_j$ used by a test case $t_i$ is encoded by an additional operation $(S_i, d_i, E_i, 1, g_j)$.

Finally, $O$ is simply the array of all such operations, and $C$ is an array of 1s of length equal to the number of machines plus the number of global resources.

To make the constraint model complete, we introduce the variable $MakeSpan$ representing the completion time of the entire schedule and seek to minimize it. Thus, $MakeSpan$ is lower bounded by the ending time of each individual test case. Finally, we state a search procedure, called the labeling process. By selecting a variable to enumerate first and a value to assign to it, the labeling process explores the search space of possible solutions to find one that optimizes a cost function. The generic model is captured by (1):

\[
\text{Cumulatives}(O, C) \wedge \\
\forall 1 \leq i \leq n : M_i \in f(t_i) \wedge \\
\forall 1 \leq i \leq n : E_i \leq MakeSpan \wedge \\
\text{LABEL(Minimize}(MakeSpan), [S_1, M_1, \ldots, S_n, M_n])
\]

Note that the labeling process considers both the starting times and the machine assignment of test cases to machines, whereas the ending times depend functionally on the starting times. Thus, a solution of the OTS problem can be obtained by searching among these variable assignments. Note also that the selection of the variable to enumerate first and the value to select first can be tuned by using different search strategies. The next subsection presents the strategy we have found to be the best.

The optimal schedule shown in Figure 1 results from solving the constraint model instance (2), where the three machines and the two global resources map to 1, 2, 3, 4, 5, respectively.
3. The TC-Sched method

Cumulatives( 
(S1, 2, E1, 1, M1)  
(S2, 4, E2, 1, M2), (S2, 4, E2, 1, 4)  
(S3, 3, E3, 1, M3), (S3, 3, E3, 1, 4)  
(S4, 4, E4, 1, M4), (S4, 4, E4, 1, 4)  
(S5, 3, E5, 1, M5)  
(S6, 2, E6, 1, M6)  
(S7, 1, E7, 1, 1)  
(S8, 2, E8, 1, 2)  
(S9, 3, E9, 1, 3)  
(S10, 5, E10, 1, M10), (S10, 5, E10, 1, 5)  
], [1, 1, 1, 1, 1])∧
M1 ∈ 1...3 ∧⋯∧ M6 ∈ 1...3 ∧ M10 ∈ {1, 3}∧
E1 ≤ MakeSpan ∧⋯∧ E10 ≤ MakeSpan∧
LABEL(Minimize(MakeSpan), [S1, M1, . . . , Sn, Mn])

By solving this constraint model with a finite domain constraint solver (taking no time for this example on a standard machine), we obtain the following optimal solution:

\[
\begin{align*}
S_1 &= 0, S_2 = 4, S_3 = 8, S_4 = 0, S_5 = 4, S_6 = 7, \\
S_7 &= 2, S_8 = 9, S_9 = 0, S_{10} = 3, \\
M_1 &= 1, M_2 = 1, M_3 = 1, M_4 = 2, M_5 = 2, M_6 = 2, \\
M_7 &= 1, M_8 = 2, M_9 = 3, M_{10} = 3, MakeSpan = 11
\end{align*}
\]

3.2 Search Procedure

The search procedure attempts to assign values to variables such that all the constraints are satisfied and the cost function is minimized. Many different strategies can be used to explore the search space but it is well known that programming the search strategy to depend on the characteristics of the scheduling problem to be solved, is the most effective approach [18].

Starting from the idea that allocating the most demanding test cases first, that is, first pick the test cases requiring the highest number of global resources, breaking ties by picking the test case with the longest duration, we encoded our own search strategy, called test case duration splitting. The strategy is based on the dichotomic division of the search space and allows the process to be backtracked at any step if an inconsistency is
detected. At each individual step, test case duration splitting works as follows:

(i) Select one test case $t_i$ using the above selection rule.

(ii) Compute $est_i + d_i$, where earliest starting time of $t_i$, denoted $S_i$, and its duration, $d_i$. Add the new constraint $S_i < est_i + d_i$ to the constraint model. If an inconsistency is detected, then backtrack to this step and add instead $S_i \geq est_i + d_i$ to the constraint model.

(iii) Assign $S_i$ to the lowest possible value of its current domain, that is, $S_i = est_i$, and check whether the rest of the constraints are still satisfiable. If an inconsistency is detected, then backtrack to the initial choice and try $est_i + 1, est_i + 2$ and so on, until $est_i + d_i - 1$.

(iv) If the constraints are satisfiable with one of these assignments, then also assign $M_i$ to the first machine on which $t_i$ can be run, where $M_i > M_{i-1}$. If this fails, try $M_i \leq M_{i-1}$. If this assignment does not violate any constraints, then move on to the next test case by going back to step 1.

The action done in step 4 has shown to be particularly effective in a CI context. By preferring a different machine from the one that the previous test case was assigned to, we achieve that the first solution found, before using branch and bound for minimization of $MakeSpan$, is a good compromise between solving time of the schedule, and execution time of the schedule which is one of the key factors in CI.

By also keeping the option to backtrack at any step, the search strategy presented above maintains completeness. This means that, in principle, the overall search space can be explored even if, ideally, only a small fraction would have to explored to find an optimal schedule. A formal proof of this statement is outside the scope of this paper, but a sketch can be obtained by observing the following:

(i) By using test-case duration splitting, all the test cases will be assigned a starting time $S_i$ and a machine $M_i$.

(ii) By successively considering $S_i < est_i + d_i$, $S_i \geq est_i + d_i$ and $S_i = est_i$, $S_i = est_i + 1$, and so on, all the possible assignments will eventually be considered.
3.3 Time-constrained Minimization

This process is the third necessary ingredient of the TC-Sched method. It is based on a classical branch-and-bound procedure [16], which consists in, in our case, estimating a candidate minimum value \( \hat{C} \) of MakeSpan and, each time a branch in the search tree is considered (e.g., \( S_i = est_i + 1 \)), pruning the search tree by using the constraint MakeSpan < \( \hat{C} \) to search among the remaining candidate solutions. Subtrees of the search space where MakeSpan ≥ \( \hat{C} \) are irrelevant to finding a better candidate minimum and can be eliminated without further examination. This process proved extremely powerful at finding optimal schedules to solve the OTS problem, as detailed in the experimental section. However, when the number of test cases grows to be several hundred, finding an optimal schedule may become an intractable problem and actually, approximating the global minimum (i.e., a quasi-optimum) is acceptable in practice. In the TC-Sched method, we considered a time-constrained version of this process to be more appropriate. The principle of this version is to allocate a contract of time (ranging from a few milliseconds to a couple of minutes) to the minimization process to find a quasi-optimum. Since the branch-and-bound procedure maintains a current candidate minimum during the search process, it can return this candidate upon interruption of the process. An important question, however, is the selection of the most appropriate contract of time to be allocated to the minimization process. This question is addressed specifically in the experimental evaluation.

4 Implementation and Exploitation

This section details our implementation of the TC-Sched method. It also discusses the challenges of inserting the TC-Sched method into a CI environment.

4.1 Implementation

We implemented the TC-Sched method in SICStus Prolog [7], a constraint logic programming environment. An implementation of the Cumulatives constraint is provided in its clpfd library [8], which also embeds an efficient constraint solver over finite domains. The clpfd library provides

\[\text{clpfd}\]

\[\hat{C}\]

The general cumulative scheduling problem is known to be NP-hard [2].
an implementation of the time-constraint branch-and-bound procedure. Using this library, we designed a generic and parametrizable constraint model for the TC-Sched method. The model is generic because it takes as input an OTS problem specified in a generic format and returns an optimal or quasi-optimal schedule. The clpfd API also allowed us to express our own search strategy. The setup of this strategy, as explained in Section 3.2, results from a long period of trial and error.

4.2 Insertion of TC-Sched into a Continuous Integration environment

An overview of the TC-Sched method inserted within a CI environment is shown in Figure 3. Since TC-Sched is designed to run as part of the CI cycle, we now introduce some important suggestions to the developer of the testing infrastructure:

- First, given that first-hand information on the execution time of each test case is crucial to the TC-Sched method, the infrastructure has to automatically monitor and store historical data about test case execution. For each test campaign run in a CI cycle, we suggest measuring the actual execution time of each test case and reusing this information to set up the OTS problem to be solved in the next cycle. Using over-estimation we can easily compute a schedule that will account for differences in machines used for test case execution. We do not claim that such estimation can be done for all types of test, because for many test cases the duration is highly dependent on the speed of the machines they are executed on. However, there exist large groups of test cases where the duration can either be calculated before testing or measured.

- Second, given that machines are constantly added or removed from the pool of available machines, it is important to design a constraint model that is highly configurable at testing time. We have seen examples where developers have embedded systems as part of their development environment in the office. During the day, the embedded system is reserved for the developer. However, during the night or when the developer is not using the embedded system, the embedded system can be included as part of the CI testing infrastructure. This means that, in practice, the constraint model has to be tuned
to handle specific execution machines and must be solved at each CI cycle. This also entails precisely controlling the solving time with respect to the test execution time to preserve the advantages of this approach.

- Finally, as shown in Figure 4, by knowing the calculated schedule from the previous CI ($CI_{n-1}$) cycle, the constraint model takes into account earliest start-time for each machine in the next CI cycle ($CI_n$). This will help generating a more efficient schedule compared to setting equal earliest starting time for all machines.

![Figure 3](image.png)

**Figure 3:** Integration of TC-Sched into a CI environment. The test case schedule solved by TC-Sched is transmitted for execution to the different machines in the machine pool, $m_1, \ldots, m_m$. The results including actual test case durations are then feed back into the repository.

A complete test campaign in a CI cycle is typically initiated upon a successful build of the software being tested. The next step is to query the machine pool to obtain a list of available machines. After all available machines are updated with the newly built software, our TC-Sched model takes as input data from the test cases of the campaign and historical data about execution times, from the storage repository. After TC-Sched has calculated an optimal (or quasi-optimal) schedule, the result is handed over to a dedicated dispatch server that is responsible for executing each test according to the calculated schedule. Finally, after the test execution process is finished, the repository is updated with the current test case execution times and the overall result of the test campaign is reported back to the CI environment. Of course, minimizing the *round-trip time* entails prioritizing the notification of the developers in case the software system fails.
5 Experimental evaluation

This section presents our findings from experimentally evaluating TC-Sched. To this end, we address the following three research questions:

- **RQ1:** How does the first solution provided by TC-Sched compare with simpler scheduling methods in terms of schedule execution time?

- **RQ2:** For TC-Sched, will an increased investment in the solving time reduce the overall time of a CI cycle? This question is about finding the most appropriate trade-off between the solving time and the execution time of the test campaign.

- **RQ3:** Can TC-Sched scale up to industrial OTS problems?

All experiments reported on in this paper were performed on a 2.4 GHz Intel Core i7 processor, running Windows 7.

5.1 Experimental Artifacts

To answer RQ1, we implemented two simple scheduling methods, referred to as the greedy method and the random method.

The random method works as follows: It first picks a test case at random and then picks a machine at random such that no resource constraints are violated. Finally, the test case is assigned the lowest possible start time on the selected machine.

\[ \text{Figure 4: One CI cycle continuing the previous. Earliest starting time of each machine is computed from the latest end time of the previous CI cycle.} \]
The greedy method is more advanced. We start by assigning to a machine the test case with the highest resource demand and continue to assign test cases with decreasing resource demand. Finally, test cases without any resource demands are assigned to machines. For each assignment, the machine that can provide the earliest starting time is selected. Note that none of the two methods employ backtracking to improve upon the initial solution.

The reason why we have chosen to compare with these two methods is twofold: 1) As explained in Section 1.1, we are not aware of any previously published work related to test case execution scheduling, which means that there is no baseline to compare against; 2) From cooperation with our industrial partners, we know that this is, in the best case, the level currently used in industry (i.e., non-optimal schedules computed manually).

To answer both RQ2 and RQ3, we have considered randomly generated benchmarks and industrial case studies.

Although there exist several excellent benchmark test suites for both JSS and FJSS like [20] and [3], they cannot be used as a comparison baseline. In fact, the OTS problem has not only theoretical differences with these problems as pointed out in Section 1.3 but it also has a different encoding format, meaning that there is no trivial translation between FJSS to OTS and conversely. To tame this difficulty, we developed a benchmark library containing 840 randomly generated OTS problems. The library is structured by the data collected from three different real-world test suites, provided by our industrial partners: a test suite for video conferencing systems (VCS) [13], a test suite for integrated painting systems (IPS) [14], and a test suite for a mobile application called TV-everywhere.

The VCS test suite contained 132 test cases and 74 machines were used to execute them. The duration of the test cases varied from 13 seconds (s) to 4 hours, where the vast majority of the tests had a duration between 100 s and 800 s.

The IPS test suite contained 33 test cases, with durations ranging from 1 s to 780 s, with 16 distinct execution units. There were two global resources for this test suite, namely, an airflow meter and a simulator for an optical encoder.

TV-everywhere is a mobile application that allows users to watch TV on tablets, smart phones, and laptops. Its test suite only contained manual test cases. However, it serves as a useful example of a test suite with
5. Experimental evaluation

Table 2: Randomly generated test suites.

<table>
<thead>
<tr>
<th># of tests</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>100</th>
<th>500</th>
</tr>
</thead>
<tbody>
<tr>
<td># machines</td>
<td>100</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TS11</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>TS8 TS12</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>-</td>
<td>TS2</td>
<td>TS4</td>
<td>TS6</td>
<td>TS9 TS13</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>TS1</td>
<td>TS3</td>
<td>TS5</td>
<td>TS7</td>
<td>TS10 TS14</td>
</tr>
</tbody>
</table>

a large number of constraints limiting the number of possible execution units for each test case.

Based on data gleaned from the above industry test suites, we generated 14 test suites, denoted TS1-TS14, by varying the number of test cases, $|T|$, and the number of execution units, $|M|$. Table 2 gives an overview of the generated test suites. For each test suite, we also varied the number of global resources, $|R| = \{3, 5, 10\}$. Thus, for test suite TSx, we may write TSxR3, TSxR5, or TSxR10 to indicate the number of resources.

Then, for each of the $14 \cdot 3$ variants, we generate 20 test suites at random. That is, the duration of the test cases, their use of resources, and available execution units were all selected at random. Therefore, in total, we generated $14 \cdot 3 \cdot 20 = 840$ different test suites. More specifically, based on our findings from the three industrial test suites, the following rules were used to generate the random test suites:

- **Duration**: The duration of each test case was chosen randomly between 1 s and 800 s.
- **Resources**: Each test case had a 30% chance of using a global resource. The number of resources was chosen randomly between 1 and $|R|$.
- **Execution units**: A total of 80% of the tests were considered to be executable on all machines, while the remaining 20% were executable on a smaller subset of machines. For these tests, the number of machines on which each test case could be executed was...
selected randomly between 1% and 40% of the number of available machines. This means that a test case was executable either on all machines (part of the 80% group) or only on at most 40% of the machines.

Many of the generated test suites had long durations. In our experience, this reflects quite well the situation faced by many companies, whose tests are of a similar nature.

5.2 RQ1: How does TC-Sched compare with simpler scheduling approaches?

To compare our TC-Sched method with the greedy and random methods, we record the first solution provided by TC-Sched. We name this solution $C^*_f$. The last solution found by TC-Sched, named $C^*_l$ is either a proved optimal solution, or the best solution found after optimizing for 5 minutes. For each of the 840 test suites, we compute the differences between the random and greedy, $C^*_f$ and greedy and $C^*_l$ and greedy where greedy is the baseline of 100%. The values are summarized in Figure 5 except for the values of random, which turn out to be 30%–60% higher than greedy.

From this experiment, we find that for all test suites except for TS1, $C^*_f$ is better than greedy. We also see that for large test suites (TS11-TS14) the difference between $C^*_f$ and $C^*_l$ is negligible. This means that there is almost no benefit obtained from running the solver for a longer time, once an initial solution has been found.

5.3 RQ2: Will an increased investment in solving time reduce the total execution time?

This RQ aims to find an appropriate trade-off between the time spent solving the constraint model, $T_s$, and the time spent executing the schedule, $C^*$. As mentioned in Section 1, the round-trip time is a critical factor in CI and it has to be kept as short as possible. The crucial question addressed here is to find the time limit which can be used as a time-out for the constraint solver in order to generate a schedule which is (quasi-)optimal with respect to total execution time, $T_t$. It means that there exist schedules where it is beneficial to terminate the solving process before finding of a global optimal schedule. By compromising the finding of a global optimum, one minimizes the overall round-trip time.
**Figure 5**: The differences in schedule execution times produced by the different methods for test suites TS1–TS14, with greedy as the baseline of 100%. The blue is the difference between $C^*_f$ and greedy and the red shows the difference between $C^*_l$ and greedy.
As mentioned above, our TC-Sched method can be given a contract of time to find a quasi-optimal solution when minimizing the time taken by the schedule. More precisely, four possible outcomes can be obtained with this time-constrained process.

- **No solution with proof**: The OTS problem has no solution, because the interaction between the constraints and TC-Sched actually demonstrates that the OTS problem is unsatisfiable.

- **No solution without proof**: TC-Sched was not able to find a solution within the given contract of time. This means that there could be a solution, but the time allocated to the search was insufficient to draw any conclusions.

- **Quasi-optimal solution**: At the end of the time contract, a solution is returned, meaning that TC-Sched calculated an estimate for the optimum but was interrupted while trying to prove the solution’s optimality.

- **Optimal solution**: Before the end of the time contract, TC-Sched returns an optimal solution. This means that TC-Sched has found a guaranteed global optimum, assuming its search heuristic is complete.

Each solution $i$ generated by TC-Sched can be represented with a tuple $(C_i^*, T_{s,i})$ where $C_i^*$ is the makespan of solution $i$ and $T_{s,i}$ is the time the solver spent getting to solution $i$. The goal of RQ2 is to find the value of $T_{s,i}$ that minimizes $(C_i^* + T_{s,i})$, $\forall i$ and use this value as the contract of time.

To answer RQ2, we executed TC-Sched on all 840 test suites, with a time contract of 5 minutes. During this process, we recorded all intermediate results from the start of the search to the end to be able to calculate the optimal value of $T_s$ for each test suite.

In Figure 6, we have shown the distribution in solving time for the first solution found by TC-Sched, the last solution and also how the optimal value of $T_s$ is distributed. For the group of 600 test suites containing up to 100 test cases (TS1-TS10), the results show that an optimal solution with respect to the total execution time, $T_t$ is found in $T_s < 5$ s for 95.5 % of the test suites. If we extend the search time to $T_s < 10$ s, the number grows to 97.1 % of the test suites. For this group, the worst case optimal solving time was $T_s = 162.4$ s. It is also worth mentioning that the worst
case for the first solution provided by TC-Sched is less than 0.5 s, which means that a solution is always found in less then 0.5 s.

For the group of 240 test suites containing 500 test cases (TS11-TS14), the results show that an optimal solution with respect to the total execution time, $T_t$ is found in $T_s < 180$ s for 78.7% of the test suites. For $T_s < 240$ s 96.6% of the test suites, an optimal $T_t$ is found.

5.4 RQ3: Can TC-Sched scale up to solve industrial OTS problems?

To answer RQ3, we considered two industrial OTS contexts: IPS, a test suite for testing a distributed paint control system for complex industrial robots, developed at ABB Robotics, Norway, and VCS, a test suite for commercial video conferencing systems, developed by CISCO Systems, Norway. Both test suites were mentioned in Section 5.1 and both case studies were developed in a CI environment.

When applying TC-Sched to the IPS test suite, we found the first quasi-optimal solution, $C^* = 1079$ s at $T_s = 20$ ms. During the next 900 ms, TC-Sched found an additional 111 quasi-optimal solutions before a guaranteed optimal solution, $C^* = 780$ s, was found at $T_s = 930$ ms. So, to integrate the IPS test suite into a CI cycle, the best compromise was obtained when $C^* = 780$ s, since $T_s$ was negligible in this case.

For the VCS test suite, an initial quasi-optimal solution, $C^* = 25773$ s was found at $T_s = 534$ ms, while another 63 solutions were found during the next 33 s. A guaranteed optimal solution was found at $T_s = 33.2$ s, where $C^* = 14637$ s. As for the IPS test suite, the best compromise was obtained when $C^*$ was minimized, yielding $T_t = 33.2 + 14637$ s.

In summary, TC-Sched can easily be applied to the two industrial test suites and in both cases the best compromise is achieved when $C^*$ is minimized.

5.5 Threats to validity

Despite the extensive amount of experiments performed to evaluate TC-Sched, several threats to validity exist.

To answer RQ1 and RQ2, we built a random generator for OTS problems. The degree to which this generator is representative of real-world test suites can be questioned, especially when it comes to interpreting its
Figure 6: The black box plot shows the distribution in solving time, \( T_s \), for the first solution found by TC-Sched. The blue box show the distribution in \( T_s \) where the total time execution time, \( T_t \), is optimal. The red box plot show the distribution in \( T_s \) for the last solution found by TC-Sched, which can be the optimal value or the last value found before timeout. The timeout was set to 5 min.
results. To mitigate such concerns, we fed the generator with meta-data from the three industrial test suites introduced in Section 5.1. Even though some of the parameters for the generated test suites were randomly selected, the size of the test suites, the number of execution units, the density of various data, and the use of global resources corresponded exactly to the industrial test suites.

As mentioned earlier, the current practice in industry is to manually craft the test case execution schedule. We note that the complexity of test schedules grows with the number of test cases, making it difficult to handcraft such schedules. To reduce the complexity of the problem, validation engineers tend to bundle several test cases into fewer but long-running tests. In our experimental evaluation, we decided to focus on test cases with random durations. However, further work is needed to determine the correlation between the duration of each test case execution and the solving time allocated to TC-Sched. If the complexity of manual test case scheduling can be offloaded to a fully automated deployment of TC-Sched, then we believe that this will lead developers to produce many more test cases of short duration. With many short-lived test cases that can be individually scheduled, we may be able to produce even more efficient schedules.

To answer RQ1, we compared TC-Sched with our own implementations of two simple methods, namely, greedy and random. This is an internal threat, since we did not attempt to optimize these methods, which could prove to be more efficient in some situations. Furthermore, it would have been preferable to compare our TC-Sched with other implementations but, unfortunately, we were unable to find any scheduling tools with the necessary capabilities to perform automated test case scheduling with multiple machines and resources.

6 Further work

As mentioned in Section 1.1, the automatic dynamic scheduling of test case execution as part of a CI environment is a new trend. However, many extensions can be foreseen.

Priorities. Adding execution priority to the test cases is an interesting feature that could be handled in TC-Sched in the future. In fact, prioritizing the execution of the most interesting test cases during the next
6. **Further work**

CI cycle would permit the validation engineers to further minimize the round-trip time. These priorities could be defined using fault detection capabilities, the execution time during a current CI cycle, and so on. Our TC-Sched method is currently indifferent with respect to these elements and we feel that some optimization opportunities may be found if these objectives are taken into account.

**Sharable resources.** With TC-Sched, we consider global resources in a simple way: Each resource is non-sharable, unique, and counted as one in the cumulative formalization of the OTS problem. This is a simplification since current industrial practices include pools of different resources. As an example, consider an infrastructure with two external instruments, where one of them is used by a running test case and another test case requires these two instruments to be used at the same time. An interesting extension of TC-Sched is, first, to consider non-unitary resources and to find schedules that keep the sum of resources under a given threshold. Thanks to the Cumulatives constraint, this is an easy extension of TC-Sched. Second, we could consider sharable resources, meaning that a global resource could be shared among two, three, or more test cases. This would require many modifications in the TC-Sched method, with the introduction of complex inequality constraints between test case execution times. This extension would also open the door for scheduling more complex test campaigns.

**Setup time.** When a machine switches from executing a test case in one group of test cases to a test case belonging to another group, then a setup time is usually necessary to reconfigure the machine and to load new configuration files. This setup cost is currently ignored in TC-Sched. An extension with setup time of the underlying constraint model of TC-Sched, would allow us to carry out more fine-grained test execution scheduling.

**Other solving methods.** As previously mentioned, scheduling has been addressed by the research community for a long time. There exist several other promising techniques as alternatives to use of CP. Comparing our current approach against techniques like Mixed-Integer Linear Program, genetic and tabu search algorithms is an interesting topic for further work.
7 Conclusion

This paper introduced TC-Sched, a Constraint Programming method to solve the optimal test suite scheduling (OTS) problem, where test cases can be executed on multiple execution machines and use global resources. The TC-Sched method is driven by use of a constraint called Cumulatives and a time-aware minimization process, and it is based on our design of a dedicated search strategy, called test case duration splitting. Our TC-Sched method is examined in the context of integration within a CI environment. According to our knowledge, this is the first time that the OTS problem has been rigorously formalized and a (time-aware) constraint-based method proposed to solve it. An experimental evaluation performed over 840 generated test suites revealed that TC-Sched outperforms simple automated test scheduling methods with respect to total execution time when the number of test cases grows up to hundreds. By considering the trade-off between the solving time and the total execution time, the evaluation also permits us to find the best compromise to allocate time-contracts to the solving process. Finally, by using TC-Sched with two industrial test suites, we demonstrated that finding the guaranteed optimal test execution time is not outside the scope of TC-Sched and thus allocating a few seconds to the solving process during each CI cycle is the best compromise. Further work on TC-Sched includes the implementation of test case priority, consideration of non-unitary sharable global resources, and integration of setup times to reconfigure the machines from one setup to another.

References


