Time-aware Test Case Execution Scheduling for Cyber-Physical Systems

Morten Mossige[1,3], Arnaud Gotlieb[2], Helge Spieker[2], Hein Meling[3], and Mats Carlsson[4]

[1] ABB Robotics, Bryne, Norway, `morten.mossige@uis.no`
[2] Simula Research Laboratory, Lysaker, Norway, {`arnaud,helge`}`@simula.no`*
[3] University of Stavanger, Stavanger, Norway, `hein.meling@uis.no`
[4] RISE SICS, Kista, Sweden, `mats.carlsson@ri.se`

Abstract. Testing cyber-physical systems involves the execution of test cases on target-machines equipped with the latest release of a software control system. When testing industrial robots, it is common that the target machines need to share some common resources, e.g., costly hardware devices, and so there is a need to schedule test case execution on the target machines, accounting for these shared resources. With a large number of such tests executed on a regular basis, this scheduling becomes difficult to manage manually. In fact, with manual test execution planning and scheduling, some robots may remain unoccupied for long periods of time and some test cases may not be executed.
This paper introduces TC-Sched, a time-aware method for automated test case execution scheduling. TC-Sched uses Constraint Programming to schedule tests to run on multiple machines constrained by the tests' access to shared resources, such as measurement or networking devices. The CP model is written in SICStus Prolog and uses the Cumulatives global constraint. Given a set of test cases, a set of machines, and a set of shared resources, TC-Sched produces an execution schedule where each test is executed once with minimal time between when a source code change is committed and the test results are reported to the developer. Experiments reveal that TC-Sched can schedule 500 test cases over 100 machines in less than 4 minutes for 99.5% of the instances. In addition, TC-Sched largely outperforms simpler methods based on a greedy algorithm and is suitable for deployment on industrial robot testing.

1 Introduction

Continuous integration (CI) aims to uncover defects in early stages of software development by frequently building, integrating, and testing software systems. When applied to the development of cyber-physical systems (CPS),[5] the process may include running integration test cases involving real hardware components

* These authors are supported by the Research Council of Norway (RCN) through the research-based innovation center Certus, under the SFI programme.
[5] CPS can simply be seen as communicating embedded software systems.

on different machines or machines equipped with specific devices. In the last decade, CI has been recognized as an effective process to improve software quality at reasonable costs [13, 14, 27, 35].

Different from 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 [15]. 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. But the achievable parallelism is limited by the availability of scarce global resources, such as a costly measurement instrument or network device, and the compatible machines per test case, targeting different machine architecture and operating systems. These global resources are required in addition to the machine executing the test case and thereby require parallel adjustments of the schedule for multiple machines.

Thus, computing an optimal test schedule with minimal round-trip time is a challenging optimization problem. Since different test cases have different execution times and may use different global resources that are locked during execution, finding an optimal schedule manually is mostly impossible. Nevertheless, manual scheduling still is state-of-the-practice in many industrial applications, besides simple heuristics. In general, successful approaches to scheduling use techniques from Constraint Programming (CP) and Operations Research (OR), additionally metaheuristics are able to provide good solutions to certain scheduling problems. We discuss these approaches further in Section 2.

Informally, the optimal test scheduling problem (OTS) is to find an execution order and assignment of all test cases to machines. Each test case has to be executed once and no global resource can be used by two test cases at the same time. The objective is to minimize the overall test scheduling and test execution time. The assignment is constrained by the compatibility between test cases and machines, that is, each test case can only be executed on a subset of machines.

This paper introduces TC-Sched, a time-aware method to solve OTS. Using the Cumulatives [1, 5] global constraint, we propose a cost-effective constraint optimization search technique. This method allows us to 1) automatically filter invalid test execution schedules, and 2) find among possible valid schedules, those that minimize the global test execution time (i.e., makespan). To the best of our knowledge, this is the first time the problem of optimal scheduling test suite execution is formalized and a fully automated solution is developed using constraint optimization techniques. TC-Sched has been developed and deployed together with ABB Robotics, Norway.

An extensive experimental evaluation is conducted over test suites from industrial software systems, namely an integrated control system for industrial robots and a product line of video-conferencing systems. The primary goal in this paper is to demonstrate the scalability of the proposed approach for CI processes involving hundreds of test cases and tens of machines, which corresponds to a realistic development environment. Furthermore, we demonstrate the cost-effectiveness of integrating our approach within an actual CI process.

2 Existing Solutions and Related Work

Automated solutions to address the OTS problem are not yet common practice. In industrial settings, test engineers manually design the scheduling of test case execution by allocating executions to certain machines at a given time 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 execute the test cases. Keeping this process manual in CI is paradoxical, since every activity should, in principle, be automated.

Regression testing [28], i.e. the repeated testing of systems after changes were made, in CI covers a broad area of research works, including automatic test case generation [9], test suite prioritization and test suite reduction [14]. There, the idea of controlling the time taken by optimization processes in test suite prioritization is not new [12]. In test suite prioritization, [38] proposed to use time-aware genetic algorithms to optimize the order in which to execute the test cases. Zhang *et al.* further refined this approach in [39] by using integer linear programming. On-demand test suite reduction [17] also exploits integer linear programming for preserving the fault-detection capability of a test suite while performing test suite reduction. Cost-aware methods are also available for selecting minimal subsets of test cases covering a number of requirements [16,23]. All these approaches participate in a general effort to better control the time allocated to the optimization algorithms when they are used in CI processes. Note however that test suite execution scheduling is different to prioritization or reduction as it deals with the notion of scheduling in time the execution of all test cases, without paying attention to any prioritization or reduction.

Scheduling problems have been studied in other contexts for decades and an extensive body of research exists on resource-constrained approaches. The scheduling domain is divided into distinct areas such as process execution scheduling in operating systems and scheduling of workforces in a construction project. The scheduling problem of this paper belongs to a scheduling category named resource-constrained project scheduling problem (RCPSP; see [7, 8, 18] for an extensive overview). RCPSP is concerned with finding schedules for resource-consuming tasks with precedence constraints in a fixed time horizon, such that the makespan is minimized [18]. From the angle of RCPSP, global resources can be expressed as *renewable resources* which are available with exactly one unit per timestep and can therefore only be consumed by a single job per timestep.

RCPSP has been addressed by both exact methods [22, 30, 32, 36], as well as heuristic methods [19, 21]. Due to the vast amount of literature, we will focus on CP/OR-methods most closely related to the work of this paper. The clear trend in both CP and OR is to solve such problems with hybrid approaches, like, for instance, the work by Schutt et al. [29] or Beck et al. [3]. Furthermore, *disjunctive scheduling problems*, a subfamily of RCPSP addressing unary resources (in our terms global resources), have been effectively solved, e.g. by lazy clause generation [33].

RCPSP is considered to be a generalization of *machine scheduling problems* where *job shop scheduling* (JSS) is one of the best known [20]. JSS is the special

case of RCPSP where each operation uses exactly one resource, and FJSS (*flexible job shop scheduling*) further extends JSS such that each operation can be processed on any machine from a given set. The FJSS is known to be NP-hard [4].

While OTS is closely related to FJSS, and efficient approaches to FJSS are known [6, 31], there are some differences. First, in OTS, execution times are machine-independent. Second, 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. Finally, some operations additionally require exclusive access to a global resource, preventing overlap with other operations.

3 Problem Modeling

This section contains a formal definition of the OTS problem for test suite execution on multiple machines with resource constraints. Based on this definition, we propose a constraint optimization model using CUMULATIVES global constraint.

3.1 Optimal Test Case Execution Scheduling

Optimal test case scheduling[6] (OTS) is an optimization problem $(\mathcal{T}, \mathcal{G}, \mathcal{M}, d, g, f)$, where $\mathcal{T}$ is a set of n test cases along with a function $d : \mathcal{T} \longrightarrow \mathbb{N}$ giving each test case a duration d_i; a set of global resources $\mathcal{G}$ along with a function $g : \mathcal{T} \longrightarrow 2^{\mathcal{G}}$ that describes which resources are used by each test case; and a set of machines $\mathcal{M}$ and a function $f : \mathcal{T} \longrightarrow 2^{\mathcal{M}}$ that assigns to each test case a subset of machines on which the test case 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 to account for small variations between the different execution machines. OTS is 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 needed to compute the schedule (T_s) plus the time needed to execute the schedule (C^*), $T_t = T_s + C^*$. Machine assignment and test case execution ordering can be described either by a time-discretized table containing a line per machine or a starting time for each test case and its assignment to a given machine.

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

In its basic version, the OTS problem includes the following constraints:
Disjunctive 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 to execute another test case on the same machine.

[6] OTS was part of the Industrial Modelling Competition at CP 2015.

Table 1. Test suite for example.

Test	Duration	Executable on	Use of global resource
1	2	$1,2,3$	-
2	4	$1,2,3$	1
3	3	$1,2,3$	1
4	4	$1,2,3$	1
5	3	$1,2,3$	-
6	2	$1,2,3$	-
7	1	1	-
8	2	2	-
9	3	3	-
10	5	$1,3$	2

Non-shared resources: When a test case uses a global resource, no other test case needing this 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 executing machine. This is reasonable 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. In any case, a sufficient over-approximation will satisfy the assumption.
There are cases where OTS can be trivially solved, e.g. with only one machine executing all test cases in sequence. 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 simply allocating the longest test cases first to the available execution machine easily calculates a best-effort solution.
Example Considering the test suite in Table 1, we present a small example. Let $\mathcal{T}$ be the test cases $\{1,\ldots,10\}$, $\mathcal{G}$ be the global resources $\{1,2\}$, and $\mathcal{M}$ be the machines $\{1,2,3\}$. The machines on which each test case in $\mathcal{T}$ can run is given in Table 1. This table can be extracted by analyzing the test scripts or querying the test management. By sharing the same resource 1, test cases $2,3,4$ cannot be executed at the same time, even if their execution is scheduled on different machines. Since test case 7 can only be executed on machine 1, test case 8 on machine 2, test case 9 on machine 3, and test case 10 on machines 1 or 3, we have to solve a complex scheduling problem. One possible *optimal* schedule 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 $T_t \approx C^*$.

3.2 The Cumulatives Global Constraint

The CUMULATIVES global constraint [5] is a powerful tool for modeling cumulative scheduling of multiple operations on multiple machines, where each opera-

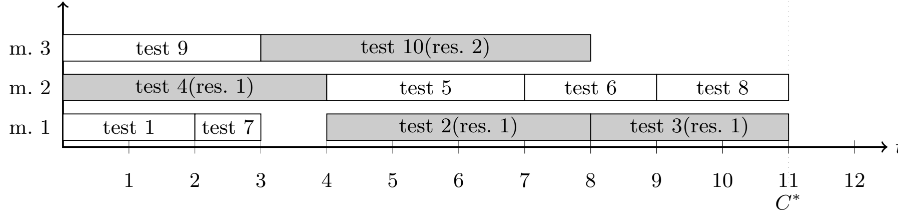

Fig. 1. An optimal solution to the scheduling problem given in Table 1. Test cases in light gray require exclusive access to a global resource.

tion 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.

Cumulatives($[O_1, \ldots, O_n], [c_1, \ldots, c_p]$)[7] constrains n operations on p machines such that the total resource consumption on each machine j does not exceed the given threshold c_j at any time [10]. An operation O_i is typically represented by a tuple $(S_i, d_i, E_i, r_i, M_i)$[8] 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 of the operation, r_i is a constant representing the amount of resource used by the operation. S_i, E_i and M_i are bounded integer variables. S_i and E_i have the domains $est_i \ldots let_i$, where est_i denotes the operation's earliest starting time and let_i denotes its latest ending time and $let_i \geq est_i + d_i$. M_i is bounded by the number of machines available, that is $1, \ldots, p$. By reducing the domain of M_i it is possible to force a specific operation to be assigned to only 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. Formally, if h represents an instant in time, we have:

$$r_i^h = \begin{cases} r_i & \text{if } S_i \leq h < S_i + d_i \\ 0 & \text{otherwise} \end{cases}$$

Cumulatives holds if and only if, for every operation O_i, $S_i + d_i = E_i$, and, for all machines k and instants h, $\sum_{i|M_i=k} r_i^h \leq c_k$. In fact, Cumulatives captures a disjunctive relation between different 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 impossible schedules.

3.3 Modeling Test Case Execution Scheduling

This section shows how the Cumulatives constraint can be used to model a schedule. In this small example, we disregard the use of global resources, and

[7] In [5] an additional third argument to Cumulatives, $Op \in \{\leq, \geq\}$ is defined. We omit it throughout our work and always set $Op = \leq$.

[8] Throughout the paper, lower-case characters are used to represent constants and upper-case characters are used to represent variables.

the constraints that some operations can only be executed on a subset of the available machines, since that will be covered in Section 3.4. By 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. Here, a resource refers to an execution machine and not to a global resource.

3.4 Introducing Global Resources

As mentioned above, global resources corresponding to physical equipment such as valves, air sensors, measurement instruments, or network devices, have limited and exclusive access. To avoid concurrent access from two test cases, additional constraints are introduced. Note that global resources must not be confused with the resource consumption or resource bounds of operations and machines.

The CUMULATIVES constraint does not support native modelling of these global resources without additional, user-defined constraints. However, there are ways to model exclusive access to such global resources by means of further constraints. The naive approach to prevent two operations from overlapping is to consider constraints over the start and stop time of the operations. For instance, if O_1 and O_2 both require exclusive access to a global resource, then the constraint $E_1 \leq S_2 \vee E_2 \leq S_1$ can be added. A less naive approach is to use a DISJUNCTIVE($\mathcal{O}^k$) constraint per global resource k, where $\mathcal{O}^k$ is the set of tasks that require that global resource, and DISJUNCTIVE prevents any pair of tasks from overlapping.

Referring to the example in Figure 1, there are ten operations to be scheduled on three machines, and two global resources, 1 and 2. The basic scheduling constraint is set up as explained in Section 3.3. Yet another way to model the global resources is to treat each resource as a new quasi-machine $1'$ corresponding to $c_{1'} = 1$ and $2'$ corresponding to $c_{2'} = 1$. For each operation requiring a global resource, we create a "mirrored" operation of the corresponding quasi-machine: $\mathcal{O}'_1 = \{O'_2, O'_3, O'_4\}$ and $\mathcal{O}'_2 = \{O'_{10}\}$. Finally, we can express the schedule with a single constraint: CUMULATIVES($\mathcal{O} \cup \mathcal{O}'_1 \cup \mathcal{O}'_2, [c_1, c_2, c_3, c_{1'}, c_{2'}]$). For each operation in $\mathcal{O}'_1$ and $\mathcal{O}'_2$ we also reuse the same domain variables for start-time, duration and end-time. The operation O_4 will be forced to have the same start-/end-time as O'_4, while they are scheduled on two different machines 2 and $1'$.

4 The TC-Sched Method

This section describes our method, TC-Sched, to solve the OTS problem. It is a *time-constrained cumulative scheduling technique*, as 1) it allows to keep fine-grained control over the time allocated to the constraint solving process (i.e., *time-constrained*), 2) it encodes exclusive resource use with constraints

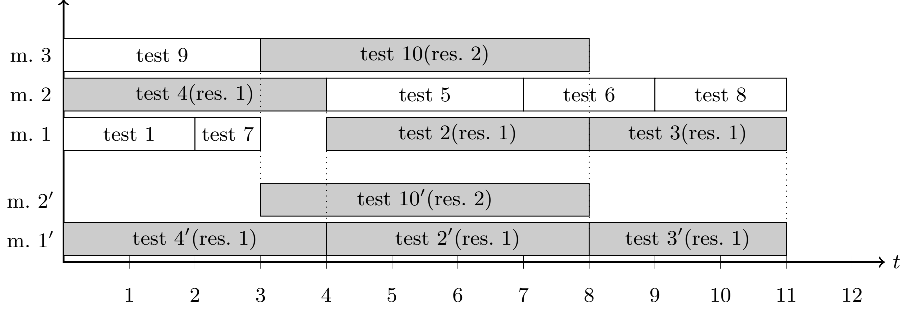

Fig. 2. Modeling global resources by creating quasi-machines and CUMULATIVES

(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 4.1, the search procedure described in Section 4.2, and the time-constrained minimization process described in Section 4.3.

4.1 Constraint Model

We encode the OTS problem with one CUMULATIVES$(\mathcal{O}, \mathcal{C})$ constraint, one DISJUNCTIVE$(\mathcal{O}^k)$ constraint per global resource k, using the second scheme from Section 3.4, and a search procedure able to find an optimal schedule among many feasible schedules. Each test case i is encoded as an operation $(S_i, d_i, E_i, 1, M_i)$ as explained in Section 3.2. $\mathcal{O}$ is simply the array of all such operations and $\mathcal{C}$ is an array of 1s of length equal to the number of machines. Suppose that there are three execution machines numbered $1, 2$, and 3; then, to say that test i can be executed on any machine, we just add the domain constraint $M_i \in \{1, 2, 3\}$, whereas to say that test i can only be executed on machine 1, we replace M_i by 1. Finally, to complete the model, we introduce the variable *MakeSpan* representing the completion time of the entire schedule and seek to minimize it. *MakeSpan* is lower bounded by the ending time of each individual test case. The generic model is captured by:

$$
\begin{aligned}
&\textsc{Cumulatives}(\mathcal{O}, \mathcal{C}) \wedge \\
&\forall \, \text{global resource } k : \textsc{Disjunctive}(\mathcal{O}^k) \wedge \\
&\forall \, 1 \leq i \leq n : M_i \in f(i) \wedge \\
&\forall \, 1 \leq i \leq n : E_i \leq \mathit{MakeSpan} \wedge \\
&\textsc{Label}(\textsc{Minimize}(\mathit{MakeSpan}), [S_1, M_1, \ldots, S_n, M_n])
\end{aligned}
\qquad (1)
$$

Note that the ending times depend functionally on the starting times. Thus, a solution to the OTS problem can be obtained by searching among the starting times and the assignment of test cases to execution machines.

4.2 Search Procedure

Our search procedure is called *test case duration splitting*, and is a branch-and-bound search that seeks to minimize the *Makespan*. The procedure makes two passes over the set of test cases. A key idea is to allocate the most demanding test cases first. To this end, the test cases are initially sorted by decreasing r_i where r_i is the number of global resources used by test case i, breaking ties by choosing the test case with the longest duration d_i.

In Phase 1, two actions are performed on each test case. First, in order to avoid a large branching factor in the choice of start time and to effectively fix the relative order among the tasks on the same machine or resource, we split the domain of the start variable, forcing an obligatory part of the corresponding task, as described in [34, Section 3.6]. Next, in order to balance the load on the machines, we choose machines in round-robin fashion. These two choices are of course backtrackable, to ensure completeness of the search procedure.

Note that at the end of Phase 1, the constraint system effectively forms a directed acyclic graph where every node is a task and every arc is a precedence constraint induced by the relative order. It is well known that such constraint systems can be solved without search by topologically sorting the start variables and assigning each of them to its minimal value. This is Phase 2 of the search.

In this procedure, the load-balancing component has shown to be particularly effective in a CI context and makes the first solution found a good compromise between solving and execution time of the schedule, which is one of the key factors in CI. Our preliminary experiments concluded, that the presented strategy provided the best compromise between cost and solution quality. Furthermore, we tried a more precise but costlier load-balancing scheme, but it did not significantly improve the quality. We also tried to sort the tests by decreasing $d_i \cdot (r_i + 1)$, which did not significantly improve the quality, either.

4.3 Time-constrained Minimization

The third necessary ingredient of the TC-Sched method is to perform branch-and-bound search under a time contract. That is, to settle on the schedule with the shortest *MakeSpan* found when the time contract ends. When the number of test cases grows to be several hundred, finding a globally optimal schedule may become an intractable problem,[9] but in practical applications it is often sufficient to find a "best-effort" solution. This leads to the important question to select the most appropriate contract of time for the minimization process, as the time used to optimize the schedule is not available to actually execute the schedule. We address this question in the experimental evaluation.

5 Implementation and Exploitation

This section details our implementation of the TC-Sched method and its insertion into CI. We implemented the TC-Sched method in SICStus Prolog [11].

[9] The general cumulative scheduling problem is known to be NP-hard [2].

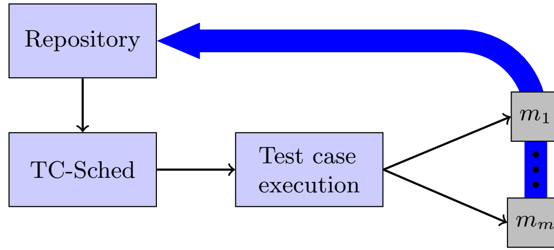

Fig. 3. Integration of TC-Sched into a CI process. The test case schedule solved by TC-Sched is transmitted for execution to the machines in the machine pool, $\mathcal{M}$. The results including actual test case durations are then feed back into the repository.

The Cumulatives constraint is available as part of the `clpfd` library [10]. The `clpfd` library also provides an implementation of the time-constrained branch-and-bound with the option to express individual search strategy (see Section 4.2). Using `clpfd`, a generic constraint model for the TC-Sched method is designed, which takes an OTS problem as input and returns an (quasi-)optimal schedule.

Since TC-Sched is designed to run as part of a CI process, we describe how it can be integrated within the CI environment. Because CI environments change and test cases and agents are constantly added or removed, TC-Sched has to be provided with a list of test cases and available machines at runtime. Furthermore, an estimation of the test case durations on the available agents has to be provided. This can either be gathered from historical execution data and then (over-)estimated to account for differences in execution machines, or, for some kinds to test suites, they are fixed and can be precisely given [26], e.g. for robotic applications where the duration is determined by the movement of the robot.

A test campaign in a CI cycle is typically initiated upon a successful build of the software being tested. As a first step, all machines available for test execution are identified and updated with the newly built software. Then, TC-Sched takes as input the test cases of the test campaign and the previous test case execution times from the storage repository. After TC-Sched calculated an optimal schedule, that schedule is handed over to a dedicated dispatch server which is responsible for distributing the test cases to the physical machines and the actual execution. Finally, after the test execution finished, the overall result of the test campaign is reported back to the users and the storage repository is updated with the latest test case execution times. Of course, minimizing the *round-trip time* leads to earlier notifications of the developers in case the software system fails and helps to improve the development cycle in CI.

6 Experimental Evaluation

This section presents our findings from the experimental evaluation of 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? This research question

states the crucial question of whether using complex constraint optimization is useful despite simpler approaches being available at almost no cost to implement.
RQ2: For TC-Sched, will an increased investment in the solving time in TC-Sched 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 in the proposed approach.
RQ3: In addition to random OTS problem instances, can TC-Sched efficiently and effectively handle industrial case studies? These cases can lead to structured problems which exhibit very different properties than random instances.

All experiments were performed on a 2.7 GHz Intel Core i7 processor with 16 GB RAM, running SICStus Prolog 4.3.5 on a Linux operating system.

6.1 Experimental Artifacts

To answer RQ1, we implemented two scheduling methods, referred to as the *random* method and the *greedy* 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 constraint is violated. Finally, the test case is assigned the lowest possible starting time on the selected machine. The *greedy* method is more advanced. At first, it assigns test cases by decreasing resource demands. Afterwards, test cases without any resource demands are assigned to the remaining machines. For each assignment, the machine that can provide the earliest starting time is selected. Note that none of the two methods can backtrack to improve upon the initial solution.

The reason we have chosen to compare with these two methods is threefold: 1) As explained in Section 2, 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 industrial state of the art (i.e., non-optimal schedules computed manually); 3) We manually checked the results on simple schedules and found them to be satisfactory, so they are a suitable comparison.

To answer our research questions, we have considered randomly generated benchmarks and industrial case studies. Although there are benchmark test suites for both JSS and FJSS, e.g., [37] or [4], they cannot be used as a comparison baseline. Furthermore, as our method approaches testing applications, a thorough evaluation on data from the target domain is justifiable.

We generated a benchmark library containing 840 OTS instances.[10] The library is structured by data collected from three different real-world test suites, provided by our industrial partners: a test suite for video conferencing systems (VCS) [24], a test suite for integrated painting systems (IPS) [26], and a test suite for a mobile application called *TV-everywhere*.

VCS is a test suite for testing commercial video conferencing systems, developed by CISCO Systems, Norway. It contains 132 test cases and 74 machines.

[10] All generated instances are available in CSPLib, a library of test problems for constraint solvers [25]

Table 2. Randomly generated test suites.

	# of tests	20	30	40	50	100	500
# machines	100	-	-	-	-	-	TS11
	50	-	-	-	-	TS8	TS12
	20	-	TS2	TS4	TS6	TS9	TS13
	10	TS1	TS3	TS5	TS7	TS10	TS14

The duration of test cases varies from 13 seconds to 4 hours, where the vast majority has a duration between 100 s and 800 s. The IPS test suite aims at testing a distributed paint control system for complex industrial robots, developed at ABB Robotics, Norway. It contains 33 test cases, with duration ranging from 1 s to 780 s, and 16 distinct machines. There are two global resources for this test suite, 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 contains manual test cases, but, in our benchmark, it serves as a useful example of a test suite with a large number of constraints limiting the number of possible machines for each test case.

Based on data from the three industrial test suites, we composed 14 groups of test suites, denoted TS1-TS14, with randomized assignments of test cases to machines and exclusive usages of global resources. Let $|T|$ be the number of test cases, and $|M|$ be the number of machines, and $|R| = \{3, 5, 10\}$ be the number of resources. Table 2 gives an overview of the groups of test suites. For test suite TSx, we write TSxR3, TSxR5, or TSxR10 to indicate the number of resources.

For each of the $14 \cdot 3$ variants, we generated 20 random test suites. The duration of each test case was chosen randomly between 1 s and 800 s, and each test case had a 30 % chance of using a global resource. The number of resources was chosen randomly between 1 and $|R|$. 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. In total, we generated $14 \cdot 3 \cdot 20 = 840$ different test suites.

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

To compare our TC-Sched method with the *greedy* and *random* methods, we recorded the first solution, C_f^*, found by TC-Sched. We also recorded the last solution, C_l^*. This is either a proved optimal solution, or the best solution found after 5 minutes of solving time. For each of the 840 test suites, we computed the differences between the *random* and *greedy*, C_f^* and *greedy*, and C_l^* and *greedy*, where *greedy* is the baseline of 100 %. The results show that *random* is 30 %-60 % worse than *greedy*, which means that *random* can clearly be discarded from further analysis. Our findings are summarized in Figure 4, showing the difference

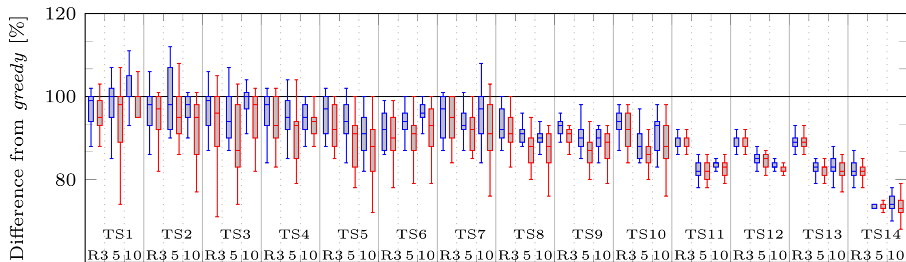

Fig. 4. The differences in schedule execution times produced by the different methods for test suites TS1–TS14, with *greedy* as the baseline of 100%. The blue is the difference between the first solution C_f^* and *greedy* and the red shows the difference between the final solution C_l^* and *greedy*.

between TC-Sched and *greedy*. For all test suites but the hardest subset of TS1 and some instances of TS2, C_f^* is better than *greedy*. We also observe that for larger test suites, i.e., TS11-TS14, there is only a marginal difference between C_f^* and C_l^*. Hence, running the solver for a longer time has only little benefit.

Furthermore, to evaluate the effectiveness of the test case duration splitting search strategy, we compared it to standard strategies available in SICStus Prolog's `clpfd` with the same constraint model on the test suites TS1 and TS14. The search first enumerates on the machine assignments increasingly, i.e. without load-balancing, and afterwards assigns end times via domain splitting by bisecting the domain, starting from the earliest end times. As variable selection strategies, we tested both the default setting, selecting the leftmost variable, and a first-fail strategy, selecting the variable with the smallest domain. Additionally, we tried sorting the variables by decreasing resource usage.

All variants of the standard searches performed substantially worse than test case duration splitting, with first-fail search on sorted variables being the best. After finding an initial solution, further improvements are rare and the makespan of the final solution is in average 4 times larger compared to using test case duration splitting with the same time contract of 5 minutes.

6.3 RQ2: Will longer solving time reduce the total execution time?

RQ2 aims at finding an appropriate trade-off between the time spent in solving the constraint model, T_s, and the time spent in executing the schedule, C^*. As mentioned in Section 1, the round-trip time is critical in CI and has to be kept low. It is therefore crucial to determine the most appropriate timeout for the constraint optimizer. The ultimate goal being to generate a schedule which is quasi-optimal w.r.t. total execution time, $T_t = T_s + C^*$.

As mentioned above, TC-Sched can be given a time-contract for finding a quasi-optimal solution when minimizing the execution time of the schedule. More precisely, with this time-constrained process four outcomes are possible.

No solution with proof: TC-Sched proves that the OTS problem has no solution due to unsatisfiable constraints.
No solution without proof: TC-Sched was not able to find a solution within the given time. Thus, there could be a solution, but it has not been found.
Quasi-optimal solution: At the end of the time-contract, a solution is returned, but TC-Sched was interrupted while trying to prove its optimality. Such a best-effort solution is usually sufficient in the examined industrial settings.
Optimal solution: Before the end of the time-contract, TC-Sched returns an optimal solution along with its proof. This is obviously the most desired result.

Each solution i generated by TC-Sched can be represented by 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 finding 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 time-contract.

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 search results to calculate the optimal value of T_s for each test suite.

Figure 5 shows 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 a solution that minimizes the total execution time, noted T_t, is found in $T_s < 5\,\text{s}$ for 96.8 % of the test suites. If we extend the search time to $T_s < 10\,\text{s}$, the number grows to 98 % of the test suites. For this group, the worst case optimal solving time was $T_s = 122.3\,\text{s}$. We see that a solution is always found in less than 0.1 s. For the group of 240 test suites containing 500 test cases (TS11-TS14), the results show that a solution that minimizes T_t is found in $T_s < 120\,\text{s}$ for 97.5 % of the test suites. A solution minimizing T_t is found in less than 240 s for all test suites, except one instance with $T_t = 264\,\text{s}$.

An increased investment in the solving part does not seem to necessarily pay off if one considers the total execution time. The reported experiments give hints to evaluate and select the optimal test contract for the solving part.

6.4 RQ3: Can TC-Sched efficiently solve industrial OTS problems?

To answer RQ3, we consider two of the three industrial case studies, namely, IPS and VCS. These case studies are composed of automated test scripts, which makes the application of the TC-Sched method especially pertinent.

In both case studies, the guaranteed optimal solution is already found as the first solution in less than 200 ms. This avoids the necessity to compromise between C^* and T_s for these industrial applications.

When applying TC-Sched to the IPS test suite, we find the optimal solution, $C^* = 780$ s, at $T_s = 10$ ms. For the VCS test suite, the optimal solution, $C^* = 14637$ s is found at $T_s = 160$ ms.

In summary, TC-Sched can easily be applied to both VCS and IPS, and in both cases, the best result is achieved when C^* is minimized and T_s is neglected.

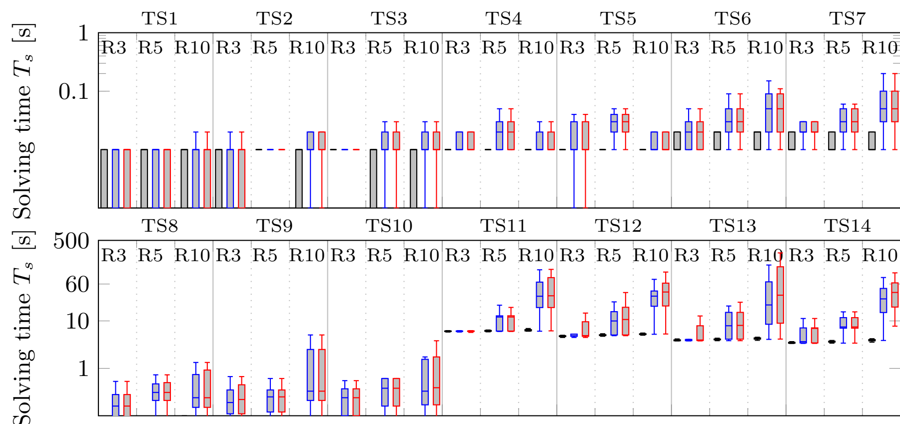

Fig. 5. The black boxes show the distribution in solving time, T_s, for the first solution found by TC-Sched. The blue boxes show the distribution in T_s where the total execution time, T_t, is optimal. Finally, the red boxes show the distribution in T_s for the last solution found by TC-Sched, which can be the optimal value or the last value found before timeout. The timeout was set to 5 min.

7 Conclusion

This paper introduced TC-Sched, a time-aware method for solving the optimal test suite scheduling (OTS) problem, where test cases can be executed on multiple execution machines with non-shareable global resources. TC-Sched exploits the Cumulatives global constraint and a time-aware minimization process, and a dedicated search strategy, called *test case duration splitting*. To our knowledge, the OTS problem is rigorously formalized for the first time and a method is proposed to solve it in CI applications. An experimental evaluation performed over 840 generated test suites revealed that TC-Sched outperforms simple scheduling methods w.r.t. total execution time. More specifically, we showed that automatic optimal scheduling of 500 test cases over 100 machines is reachable in less than 4 minutes for 99.5% instances of the problem. By considering trade-offs between the solving time and the total execution time, the evaluation allowed 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 possible and that TC-Sched can effectively solve the OTS problem in practice.

Further work includes consideration of test case priorities, non-unitary shareable global resources, as well as explicit symmetry breaking in the model. Additional evaluation and comparison against heuristic methods, such as evolutionary algorithms, or Mixed-Integer Linear Programming could extend the presented work and support the integration of TC-Sched in practical CI processes.

References

1. Aggoun, A., Beldiceanu, N.: Extending CHIP in order to solve complex scheduling and placement problems. Mathematical and Computer Modelling 17(7), 57–73 (1993)
2. Baptiste, P., Le Pape, C., Nuijten, W.: Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems, vol. 39. Springer Science & Business Media (2001)
3. Beck, J.C., Feng, T.K., Watson, J.P.: Combining constraint programming and local search for job-shop scheduling. INFORMS Journal on Computing 23(1), 1–14 (2011)
4. Behnke, D., Geiger, M.J.: Test instances for the flexible job shop scheduling problem with work centers. Tech. Rep. RR-12-01-01, Helmut-Schmidt University, Hamburg, Germany (2012)
5. Beldiceanu, N., Carlsson, M.: A new multi-resource cumulatives constraint with negative heights. In: CP 2002. LNCS, vol. 2470, pp. 63–79. Springer (2002)
6. Brandimarte, P.: Routing and scheduling in a flexible job shop by tabu search. Annals of Operations research 41(3), 157–183 (1993)
7. Brucker, P., Knust, S.: Complex Scheduling (GOR-Publications). Springer-Verlag New York, Inc., Secaucus, NJ, USA (2006)
8. Brucker, P., Drexl, A., Möhring, R., Neumann, K., Pesch, E.: Resource-constrained project scheduling: Notation, classification, models, and methods. European Journal of Operational Research 112(1), 3–41 (1999)
9. de Campos, J., Arcuri, A., Fraser, G., de Abreu, R.: Continuous test generation: Enhancing continuous integration with automated test generation. In: ASE 2014. pp. 55–66. Västerås, Sweden (2014)
10. Carlsson, M., Ottosson, G., Carlson, B.: An open-ended finite domain constraint solver. In: PLILP 1997. LNCS, vol. 1292, pp. 191–206 (1997)
11. Carlsson, M., et al.: SICStus Prolog user's manual, release 4. Tech. rep., SICS - Swedish Institute of Computer Science (2007)
12. Do, H., Mirarab, S., Tahvildari, L., Rothermel, G.: The effects of time constraints on test case prioritization: A series of controlled experiments. IEEE Trans. on Soft. Eng. 36(5), 593–617 (2010)
13. Duvall, P.M., Matyas, S., Glover, A.: Continuous Integration: Improving Software Quality and Reducing Risk. Pearson Education (2007)
14. Elbaum, S., Rothermel, G., Penix, J.: Techniques for improving regression testing in continuous integration development environments. In: FSE 2014 (2014)
15. Fowler, M., Foemmel, M.: Continuous integration (2006), `http://martinfowler.com/articles/continuousIntegration.html`
16. Gotlieb, A., Marijan, D.: Flower: Optimal test suite reduction as a network maximum flow. In: ISSTA 2014. pp. 171–180. San José, CA, USA (2014)
17. Hao, D., Zhang, L., Wu, X., Mei, H., Rothermel, G.: On-demand test suite reduction. In: ICSE 2012. pp. 738–748 (2012)
18. Hartmann, S., Briskorn, D.: A survey of variants and extensions of the resource-constrained project scheduling problem. European Journal of Operational Research 207(1), 1–14 (2010)
19. Hartmann, S., Kolisch, R.: Experimental evaluation of state-of-the-art heuristics for the resource-constrained project scheduling problem. European Journal of Operational Research 127(2), 394–407 (2000)

20. Herroelen, W., De Reyck, B., Demeulemeester, E.: Resource-constrained project scheduling: a survey of recent developments. Computers & Operations Research 25(4), 279–302 (1998)
21. Kolisch, R., Hartmann, S.: Experimental investigation of heuristics for resource-constrained project scheduling: An update. European Journal of Operational Research 174(1), 23–37 (2006)
22. Kreter, S., Schutt, A., Stuckey, P.J.: Modeling and solving project scheduling with calendars. In: Pesant, G. (ed.) CP 2015. LNCS, vol. 9255, pp. 262–278. Springer (2015)
23. Lin, C., Tang, K., Kapfhammer, G.: Test suite reduction methods that decrease regression testing costs by identifying irreplaceable tests. Information and Software Technology 56, 1322–1344 (2014)
24. Marijan, D., Gotlieb, A., Sen, S.: Test case prioritization for continuous regression testing: An industrial case study. In: ICSM 2013. Eindhoven, The Netherlands (2013)
25. Mossige, M.: CSPLib problem 073: Test scheduling problem. `http://www.csplib.org/Problems/prob073`
26. Mossige, M., Gotlieb, A., Meling, H.: Using CP in automatic test generation for ABB Robotics' paint control system. In: O'Sullivan, B. (ed.) CP 2014. LNCS, vol. 8656. Springer (2014)
27. Orso, A., Rothermel, G.: Software testing: a research travelogue (2000–2014). In: FOSE 2014. pp. 117–132. Hyderabad, India (2014)
28. Orso, A., Shi, N., Harrold, M.J.: Scaling regression testing to large software systems. In: FSE 2014. pp. 241–251. ACM Press, Newport Beach, CA, USA (2004)
29. Schutt, A., Feydy, T., Stuckey, P.J., Wallace, M.G.: Why Cumulative Decomposition Is Not As Bad As It Sounds. In: CP 2009. LNCS, vol. 5732, pp. 746–761 (2009)
30. Schutt, A., Chu, G., Stuckey, P.J., Wallace, M.G.: Maximising the net present value for resource-constrained project scheduling. In: CPAIOR 2012. LNCS, vol. 7514, pp. 362–378. Springer (2012)
31. Schutt, A., Feydy, T., Stuckey, P.J.: Scheduling optional tasks with explanation. In: Schulte, C. (ed.) CP 2013. LNCS, vol. 8124, pp. 628–644. Springer (2013)
32. Schutt, A., Feydy, T., Stuckey, P.J., Wallace, M.G.: Solving RCPSP/max by lazy clause generation. Journal of Scheduling 16(3), 273–289 (2013)
33. Siala, M., Artigues, C., Hebrard, E.: Two clause learning approaches for disjunctive scheduling 9255, 393–402 (2015)
34. Simonis, H., O'Sullivan, B.: Search strategies for rectangle packing. In: Stuckey, P.J. (ed.) CP 2008. LNCS, vol. 5202, pp. 52–66. Springer (2008)
35. Stolberg, S.: Enabling agile testing through continuous integration. In: AGILE 2009. pp. 369–374. IEEE (2009)
36. Szeredi, R., Schutt, A.: Modelling and Solving Multi-mode Resource-Constrained Project Scheduling. In: CP 2016. LNCS, vol. 9892, pp. 877–878. Springer (2016)
37. Taillard, E.: Benchmarks for basic scheduling problems. European Journal of Operational Research 64(2), 278–285 (1993)
38. Walcott, K.R., Soffa, M.L., Kapfhammer, G.M., Roos, R.S.: Time-aware test suite prioritization. In: ISSTA 2006. pp. 1–12. Portland, Maine, USA (2006)
39. Zhang, L., Hou, S., Guo, C., Xie, T., Mei, H.: Time-aware test-case prioritization using integer linear programming. In: ISSTA 2009. pp. 213–224. Chicago, IL, USA (2009)