On Optimization of Test Parallelization with Constraints

Masoumeh Parsa, Adnan Ashraf, Dragos Truscan, and Ivan Porres

firstname.lastname@abo.fi

Abstract: Traditionally, test cases are executed sequentially due to the dependency of test data. For large system level test suites, when a test session can take hours or even days, sequential execution does not satisfy any more the industrial demands for short lead times and fast feed-back cycles. Parallel test execution has appeared as an appealing option to cut down on the test execution time. However, running tests in parallel is not a trivial task due to dependencies and constraints between tests cases. Therefore, we propose an approach to parallelize test execution based on the available resources, constraints between tests, and the duration of tests that creates parallel test execution schedules that can be run in multiple testing environments simultaneously. We formulate and solve the problem as Ant Colony System, in order to find a near-optimal solution.

1 Introduction

The number of test cases required to ensure the quality of a software system grows hand-in-hand with its complexity, and consequently, the total test execution time increases proportionally. For large software systems, test execution time becomes increasingly critical in automated regression testing, where a large suite of tests is executed frequently on continuous integration servers.

Different approaches like test selection, test prioritization or test case reduction are typically used to speed up test execution, especially in the context of regression testing [YH12]. However, these improvements can be limited for large test suites.

Test suites are trivially parallelizable if tests are independent, that is, if one test does not rely on the system state established by a previous test and it is free of interference from other tests. An example of an interference is when two or more tests require exclusive access to a shared resource such as a database. However, one cannot always assume that these conditions hold and tests that pass when executed in a certain sequence may fail under trivial parallelization. A third, perhaps less obvious but even more important issue is what we refer to as state incompatibility. In this case, one test may leave the system in a state from which another test cannot proceed. One concrete example of this is a test that deletes data from a database that is expected by other tests. While such cases can often be handled by resetting the system under test to a known initial state, resets can be time-consuming and an efficient test execution approach should strive to minimize or even remove the need for system resets.

Although test designers may strive to create independent test cases, recent studies show that in practice tests are often not completely independent [ZJW+14, BMDK15], while other researchers consider these useful and exploit test dependencies [HM13]. As a conse-

quence, there is a need to plan and schedule the execution of complex test suites in order to avoid undesired interactions between tests and time consuming system resets. Although it is possible to create a test schedule manually, this is not viable in practice if the number of tests is large. To address this issue, we propose an approach to build test schedules automatically by analyzing known dependencies among tests and their execution time. The dependencies between the tests can be derived automatically [ZJW+14] or defined manually [dep]. The scheduler is run before each integration testing while preparing the system under test. We aim to find the best possible groups and order of tests in each group to be distributed between a number of agents which share or do not share the same resources with the objective of reducing the total time that is required for all the tests to be executed. The other objective is to decrease the number of failed tests which are caused by the dependencies between the tests. We define the relations between the tests as a set of constraints in scheduling problems. The first objective of the work is to reduce the tests failing due to their state incompatibilities, interferences or dependencies while executing in parallel. The second objective is to minimize the test execution time of the entire test suite, by searching for the best possible ordering of tests between different agents.

2 Background and Related Work

Recent work exploits test case dependencies as a means for prioritization of test suites [HM13], online reduction of test suites [AMPW15], or for protocol conformance testing in the context of distributed systems [MGM15]. Other works investigate how test dependencies can be detected in order to improve the independence of test cases [GSHM15] or to shorten the test execution time by minimizing the number of database resets [HKK05]. However, none of these works investigated approaches for executing tests in parallel.

Haftmann et al. propose an approach for running test cases in parallel [HKL05]. The approach involves partitioning the test sequences between test executors and ordering the execution sequence on each executor, as an extension of their work in [HKK05], in which three scheduling strategies were proposed for resolving the test incompatibilities on database applications systems with minimum number of resets.

In the previously mentioned paper, the constraints that are taken into consideration are the incompatibility and interference constraints between the tests. However, we also cover the dependency constraint. Furthermore, the goal of reordering the tests in previous works is to reduce the system reset in order to reduce the test execution time, but we are aiming on generating a near optimal schedule which satisfy the constraint and reduce the total test execution time.

Since test scheduling optimization is a complex combinatorial optimization problem, it is computationally expensive to find an exact solution for a very large number of test cases. Therefore, we formulate it as a search problem and apply a highly adaptive online optimization [HLS+13] approach called Ant Colony Optimization (ACO) [DG97] to find a near-optimal solution in polynomial time. There are a number of ant algorithms, such as, Ant System (AS), Max-Min AS (MMAS), and Ant Colony System (ACS) [DDCG99] [DG97]. ACS [DDCG99] was introduced to improve the performance of AS and it is currently one of the best performing ant algorithms. Therefore, in this paper, we apply ACS to the test scheduling optimization problem.

3 The Test Scheduling Problem

In this section, we represent the problem of test scheduling by defining the dependencies, interference and state incompatibilities between the tests. We assume that we have at our disposal one or more test environments (TE) that can execute test cases. Each TE has a number of agents (AG). Each AG can run one test case at a time, in parallel with the other AGs. The AGs in a given TE have a shared system under test and state (memory, file system, database), while each TE is isolated from the other TEs and cannot collaborate or interfere with each other.

In Figure 2 we represent an abstraction of the problem by having two TEs. The first test environment TE_1 includes two AGs while the second test environment TE_2 contains three AGs. The number of test cases to be executed are 13 and each sequence of tests that are scheduled for each AG are represented in front of the agent with respecting the order. In this example, $test_3$ depends on $test_1$ and $test_2$ while $test_9$ is dependent on $test_1$ and $test_8$; and $test_8$ has interference with $test_1$ and $test_2$. Furthermore, $test_{10}$ has state incompatibility with $test_6$ and interference with $test_9$. The maximum execution time represents the time when all tests are completely executed on the AGs. The example constraints between the tests are represented in Figure 1. The notations that are used in Figure 1 are explained later in this section.

The schedule for two different tests running in the same agent cannot overlap. Furthermore, in the case of test interference, we cannot schedule two tests interfering simultaneously in the same TE. We represent the test interference as a relation G_{inf} where $G_{inf} = (TC, E_{inf})$ where TC is the set of tests and $E_{inf} \subseteq TC \times TC$. It is assumed that tests share resources if they are executed on the same test environment.

$$(t_1, t_3) \in E_{dep}, (t_2, t_3) \in E_{dep}$$
$$(t_1, t_9) \in E_{dep}, (t_8, t_9) \in E_{dep}$$
$$(t_1, t_9) \in E_{inc}, (t_{10}, t_9) \in E_{inf}$$
$$(t_1, t_8) \in E_{inf}, (t_2, t_8) \in E_{inf}$$

Fig. 1: Constraints for the example

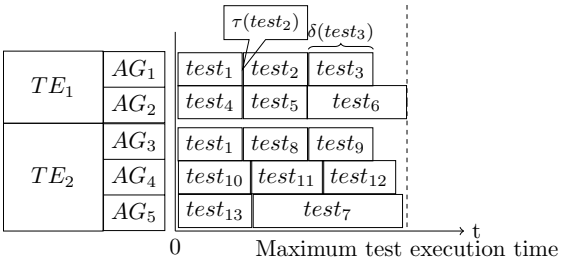

Fig. 2: Test scheduling of the abstract example

In state dependency, some tests might be preconditions for other tests which requires them to be executed in order to succeed. This dependency can be represented as a relation $(t_1, t_2) \in E_{dep}$ which implies that test t_1 should be executed before test t_2.

A third, perhaps less obvious but relevant constraint, is what we refer to as state incompatibility that occurs when one test leaves a TE in a state from which another test cannot proceed. We can represent the incompatibility relation as G_{inc} where $G_{inc} = (TC, E_{inc})$, $E_{inc} \subseteq TC \times TC$.

To optimise the test execution time, we need to minimise the maximum execution time of tests on the agents. Given the defined constraints the goal is to minimise the finishing time of each test which cause the minimum ending time of all the tests in each agent.

This value is described as TET or overall test execution time. By minimising TET, we actually minimise our objective which is the maximum execution time of tests.

4 ACS-Based Test Scheduling Optimization Algorithm

In this section, we present our **ACS**-based **T**est **S**cheduling **O**ptimization algorithm (ACS-TSO). ACO is a multi-agent approach to difficult combinatorial optimization problems, such as, traveling salesman problem and network routing [DDCG99]. It is inspired by the foraging behavior of real ant colonies. While moving from their nest to the food source and back, ants deposit a chemical substance on their path called pheromone. Other ants can smell pheromone and they tend to prefer paths with a higher pheromone concentration. Thus, ants behave as agents who use a simple form of indirect communication called *stigmergy* to find better paths between their nest and the food source. It has been shown experimentally that this simple pheromone trail following behavior of ants can give rise to the emergence of the shortest paths [DDCG99]. It is important to note here that although each ant is capable of finding a complete solution, high quality solutions emerge only from the global cooperation among the members of the colony who concurrently build different solutions.

In the context of test case schedule, each test case is allocated to an agent. Therefore, ACS-TSO makes a set of tuples T, where each tuple $t \in T$ consists of two elements: test case tc, and agent ag

$$t := (tc, ag) \tag{1}$$

The output of the ACS-TSO algorithm is a test case schedule plan Ψ, which, when enforced, would result in a reduced overall test execution time. Thus, the objective function for the proposed ACS-TSO algorithm is

$$minimize\ f(\Psi) := \max_{s \in ag}\{TET_s\} \tag{2}$$

where Ψ is the test case schedule plan and TET_s is the test execution time on ag. Since the main objective of test case schedule is to minimize the overall test execution time, the objective function is primarily defined in terms of test execution time TET on each ag.

Each of the nA ants uses a state transition rule to choose the next tuple to traverse. According to the following rule, an ant k chooses a tuple s to traverse next by applying

$$s := \arg\max_{u \in T_k}\{[\tau_u] \cdot [\eta_u]^{\beta}\} \tag{3}$$

where τ denotes the amount of pheromone and η represents the heuristic value associated with a particular tuple. β is a parameter to determine the relative importance of the heuristic value with respect to the pheromone value. The expression *arg max* returns the tuple for which $[\tau] \cdot [\eta]^{\beta}$ attains its maximum value. $T_k \subset T$ is the set of tuples that remain to be traversed by ant k. The heuristic value η_s of a tuple s is defined as

$$\eta_s := \begin{cases} (TET_{ag})^{-1} \cdot \propto_{tc}, & \text{if } |D_{tc}| = 0 \\ |D_{tc}|/|D_{all}| \cdot \propto_{tc}, & \text{if } TET_{ag} = 0 \\ |D_{tc}|/|D_{all}| \cdot (TET_{ag})^{-1} \cdot \propto_{tc} & \text{else} \end{cases} \tag{4}$$

$$\propto_{tc} := \frac{TET_{tc}}{\sum\limits_{tc \in TC} TET_{tc}} \tag{5}$$

where D_{tc} is the set of dependencies of test case tc in tuple s and TET_{ag} is the current test execution time of ag in tuple s. The heuristic value η is based on the product of the number of dependencies of the test case and the multiplicative inverse of the current test execution time of the agent in tuple s. Therefore, the tuples in which the test case has a higher number of dependencies and the agent has a shorter current test execution time receives the highest heuristic value. Moreover, in the calculation of heuristic value, we consider the execution time of each test in proportion of the execution time of all the tests, in which a test with higher execution time will get a higher heuristic value. The heuristic value favors dependent test cases over independent test cases. Therefore, a test case with a higher number of dependencies receives higher heuristic value than a test case with fewer dependencies. Similarly, the main reason for favoring agents with a shorter current test execution time is to minimize the overall test execution time.

4.1 Pheromone Distribution

The stochastic state transition rule in (3) prefers tuples with a higher pheromone concentration which leads in a reduced overall test execution time. (3) is called exploitation [DG97]. It chooses the best tuple that attains the maximum value of $[\tau] \cdot [\eta]^{\beta}$. In addition to the stochastic state transition rule, ACS also uses a global and a local pheromone trail evaporation rule. The global pheromone trail evaporation rule is applied towards the end of an iteration after all ants complete their test suite schedule plans. It is defined as

$$\tau_s := \begin{cases} (1-\alpha) \cdot \tau_s + \alpha \cdot \Delta^+_{\tau_s}, & \text{if } s \notin \text{Violations} \\ (1-3*\alpha) \cdot \tau_s + \alpha \cdot \Delta^+_{\tau_s}, & \text{if } s \in \text{Violations} \end{cases} \tag{6}$$

where $\Delta^+_{\tau_s}$ is the additional pheromone amount that traditionally is given only to those tuples that belong to the global best test schedule plan Ψ^+ in order to reward them. However, we only add the additional pheromone to the subset of tuples that contribute to the global best test schedule plan and did not violate the constraints. Moreover, we define another global updating rule for the violated tuples to have higher decay pheromone in compared to the tuples that were not existed in the solution. The additional pheromone is defined as

$$\Delta^+_{\tau_s} := \begin{cases} f(\Psi^+), & \text{if } s \in \Psi^+ \wedge s \notin \text{Violations} \\ 0, & \text{otherwise} \end{cases} \tag{7}$$

$\alpha \in (0, 1]$ is the pheromone decay parameter, and Ψ^+ is the global best test schedule plan from the beginning of the trial.

The local pheromone trail update rule is applied on a tuple when an ant traverses the tuple while making its test schedule plan. It is defined as

$$\tau_s := (1-\rho) \cdot \tau_s + \rho \cdot \tau_0 \tag{8}$$

where $\rho \in (0, 1]$ is similar to α and τ_0 is the initial pheromone level, which is computed as the multiplicative inverse of the execution time of all the test cases.

$$\tau_0 := (\sum_{tc \in TC} TET_{tc})^{-1} \quad (9)$$

D_{tc}	set of dependencies of test case tc
Int_{tc}	set of interference of test case tc
Inc_{tc}	set of incompatibilities of test case tc
P	set of test schedule plans
T	set of tuples
T_k	set of tuples not yet traversed by ant k
t	a tuple
tc	test case in a tuple
ag	agent in a tuple
TET_{ag}	current test execution time of agent ag
Ψ	a test schedule plan
Ψ^+	the global best test schedule plan
Ψ_k	ant-specific test schedule plan of ant k
η	heuristic value
τ	amount of pheromone
τ_0	initial pheromone level
$\Delta^+_{\tau_s}$	additional pheromone amount given to the tuples in Ψ^+
α	pheromone decay parameter in the global updating rule
β	parameter to determine the relative importance of η
ρ	pheromone decay parameter in the local updating rule
nA	number of ants that concurrently build their test schedule plans
nI	number of iterations of the for loop that creates a new generation of ants
$f(\Psi)$	objective function that minimizes the overall test execution time

Fig. 3: Summary of concepts and their notations 3

```
1: Ψ+ := ∅, P := ∅
2: ∀t ∈ T|τt := τ0
3: for i ∈ [1, nI] do
4:     for k ∈ [1, nA] do
5:         Ψk := ∅
6:         while all tc ∈ TC are allocated do
7:             choose a tuple t ∈ T to traverse by
               using (3)
8:             apply local update rule in (8) on t
9:             if ant k has not already allocated tc in t
               then
10:                if Dtc of tc in t is not empty then
11:                    if test in Dtc is not already alloca-
                       ted to the same test environment as t
                       then
12:                        if allocating test is not creating
                           an interference or incompatibili-
                           ty in Ψk then
13:                            add tuple (test,ag) to Ψk whe-
                               re test is in Dtc and ag is the
                               same agent as t
14:                            update TETag of ag in t
15:                        end if
16:                    end if
17:                end if
18:                if allocating t is not creating an interfe-
                   rence or incompatibility in Ψk then
19:                    add t to Ψk
20:                    update TETag of agent ag in t
21:                end if
22:            end if
23:        end while
24:        if Ψk is complete then
25:            add Ψk to P
26:        end if
27:    end for
28:    Ψ+ := arg max_{Ψk ∈ P}{f(Ψk)}
29:    apply global update rule in (6) on all s ∈ T
30: end for
31: return Ψ+
```

Fig. 4: ACS algorithm for test execution time optimization (ACS-TSO)

The pseudocode of the proposed ACS-TSO algorithm is given in Figure 4. The algorithm makes a set of tuples T using (1) and sets the pheromone value of each tuple to the initial pheromone level τ_0 by using (9) (line 2). Then, it iterates over nI iterations (line 3), where each iteration $i \in nI$ creates a new generation of nA ants that concurrently build their test schedule plans (lines 4–24). Each ant $k \in nA$ iterates its loop until all test cases in TC are allocated (lines 6–23).

Afterwards, based on the state transition rule in (3) each ant chooses a tuple t to traverse

next (line 7). The local pheromone trail update rule in (8) and (9) is applied on t (line 8).

If ant k has not already allocated tc in t and tc is an independent test case, t is added to the ant-specific test schedule plan Ψ_k if it cause no incompatibility and no interference in the solution and the test execution time of agent ag in t, TET_{ag}, is updated to reflect the impact of the test case schedule (line 18–21). However, if tc in t is a dependent test case (line 10), it is essential to allocate all test cases on which tc is dependent on the same test environment before allocating tc on ag in t. Therefore, the algorithm uses the set of test cases on which tc in t is dependent denoted as D_{tc} from (4)(line 11–16).

However, to prevent multiple scheduling of the same test cases on the same test environment, the algorithm removes any test cases from D_{tc} which are already allocated to ag in t (line 12). Then, it adds all tuples to the ant-specific test schedule plan Ψ_k where tc is in D_{tc} and ag is the same as in t (line 13). At this point, it is possible that a test case in D_{tc} may be allocated to more than one server. Such a situation may arise when more than one dependent test cases are dependent on the same test case(s). (line 18–21).

Afterwards, when all ants complete their test schedule plans, all ant-specific test schedule plans are added to the set of test schedule plans P (line 24–26), each test schedule plan $\Psi_k \in P$ is evaluated by applying the objective function in (2), the thus far global best test schedule plan Ψ^+ is selected (line 28), and the global pheromone trail update rule in (6) and (7) is applied on all tuples (line 29). Finally, when all iterations $i \in nI$ complete, the algorithm returns the global best test schedule plan Ψ^+ (line 31).

5 Conclusions and Future Work

In this paper, we presented a novel approach for scheduling tests in parallel. The aim is to optimize the execution time of the tests in a test suite while satisfying incompatibility, dependency and interference constraints between the tests. In order to decrease the execution time we are looking for the best possible partitioning of tests in a number of groups and reordering of tests in each group to reduce the false positive and false negative test execution results. Since computing the exact solution is impractical for a large number of tests, we proposed a metaheuristics based approach to achieve an approximate solution in polynomial time. The time complexity of our proposed algorithm is not linear, however, the algorithm can be parallelised to accelerate the execution time. Currently, we require the algorithm to be run before each integration to take the possible modification of tests into consideration. We implemented the proposed algorithm in a Java based framework.

There is a difference between scheduling the tests and the traditional resource constrained scheduling that required us to propose a more general scheduling strategy rather than using one of the known algorithms. There is no notion of state in performing the tasks in resource constrained scheduling. If a task has been performed once, it will not be necessary to be performed again. However, in test scheduling, the state achieved on a test environment is important for the other tests. In this case, it is not enough that test has been executed before on another environment.

In the proposed approach, we defined the interference and incompatibility as hard constraints in which should be satisfied while the solution is constructed, and the dependency constraint as a soft constraint in which we aim at reducing in the next cycles. Moreover,

we define the objective function to reduce the execution time of tests. For the future work, we may focus on incorporating SMT (Satisfiability Modulo Theories) solvers to tackle the problem to achieve an optimal schedule.

Acknowledgements

This work was supported by the Need for Speed (N4S) Program (http://www.n4s.fi).

Literatur

[AMPW15] S. Arlt, T. Morciniec, A. Podelski und S. Wagner. If A Fails, Can B Still Succeed? Inferring Dependencies between Test Results in Automotive System Testing. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, Seiten 1–10, April 2015.

[BMDK15] J. Bell, E. Melski, M. Dattatreya und G.E. Kaiser. Vroom: Faster Build Processes for Java. *Software, IEEE*, 32(2):97–104, Mar 2015.

[DDCG99] Marco Dorigo, Gianni Di Caro und Luca M. Gambardella. Ant algorithms for discrete optimization. *Artif. Life*, 5(2):137–172, April 1999.

[Dep] DepUnit. `https://code.google.com/p/depunit/`. [Online].

[DG97] M. Dorigo und L.M. Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *Evolutionary Computation, IEEE Transactions on*, 1(1):53–66, 1997.

[GSHM15] Alex Gyori, August Shi, Farah Hariri und Darko Marinov. Reliable Testing: Detecting State-polluting Tests to Prevent Test Dependency. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, Seiten 223–233, New York, NY, USA, 2015. ACM.

[HKK05] Florian Haftmann, Donald Kossmann und Er Kreutz. Efficient regression tests for database applications. In *In Conference on Innovative Data Systems Research (CIDR*, Seiten 95–106, 2005.

[HKL05] Florian Haftmann, Donald Kossmann und Eric Lo. Parallel Execution of Test Runs for Database Application Systems. In Klemens Bhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-ke Larson und Beng Chin Ooi, Hrsg., *VLDB*, Seiten 589–600. ACM, 2005.

[HLS+13] Mark Harman, Kiran Lakhotia, Jeremy Singer, David R. White und Shin Yoo. Cloud engineering is Search Based Software Engineering too. *Journal of Systems and Software*, 86(9):2225 – 2241, 2013.

[HM13] S. Haidry und T. Miller. Using Dependency Structures for Prioritization of Functional Test Suites. *Software Engineering, IEEE Transactions on*, 39(2):258–275, Feb 2013.

[MGM15] Alberto Marroquin, Douglas Gonzalez und Stephane Maag. A Novel Distributed Testing Approach Based on Test Cases Dependencies for Communication Protocols. In *Proceedings of the 2015 Conference on Research in Adaptive and Convergent Systems*, RACS, Seiten 497–504, New York, NY, USA, 2015. ACM.

[YH12] S. Yoo und M. Harman. Regression Testing Minimization, Selection and Prioritization: A Survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, Marz 2012.

[ZJW+14] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kivanç Muşlu, Wing Lam, Michael D. Ernst und David Notkin. Empirically Revisiting the Test Independence Assumption. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, Seiten 385–396, New York, NY, USA, 2014. ACM.