Using Global Constraints to Automate Regression Testing

Arnaud Gotlieb, Dusica Marijan

■ *Communicating or autonomous systems rely on high-quality software-based components. that must be thoroughly verified before they are released and deployed in operational settings. Regression testing is a crucial verification process that compares any new release of a software-based component against its previous versions, by executing available test cases. However, limited testing time makes selection of test cases in regression testing challenging, and some selection criteria must be respected. Validation engineers usually address this problem, coined as test suite reduction (TSR), through manual analysis or by using approximation techniques. In this paper, we address the TSR problem with sound artificial intelligence techniques such as constraint programming (CP) and global constraints. By using distinct cost-value-aggregating criteria, we propose several constraint-optimization models to find a subset of test cases that cover all the test requirements and optimize the overall cost of selected test cases. Our contribution includes reuse of existing preprocessing rules to simplify the problem before solving it and the design of structure-aware heuristics that take into account the notion of the costs associated with test cases. The work presented in this paper has been motivated by an industrial application in the communication domain. Our overall goal is to develop a constraint-based approach of test suite reduction that can be deployed to test a complete product line of conferencing systems in continuous delivery mode. By implementing this approach in a software prototype tool and experimentally evaluating it on both randomly generated and industrial instances, we hope to foster a quick adoption of the technology.*

More and more, communicating or autonomous systems rely on software-based components that have high standards in terms of reliability, robustness, and quality. For instance, professional conferencing systems or industrial robotics systems are released in markets where high quality is considered a competitive advantage. For these systems, an increased effort in software validation and verification is required to produce high-quality components that can be deployed in operational settings.

Software validation and verification include several distinct phases such as functional testing, performance testing, and regression testing. Regression testing verifies that a new release of a software component still performs as expected after new features are implemented. By executing the software component with existing test cases that were used to test previous releases, regression testing checks for the absence of regression faults, that is, faults that may have been reintroduced into the application during development of new features. In order to keep the time to market of new releases short, a judicious selection of test scripts to execute has to be performed.

Dealing with multiple criteria when performing regression testing is important. For example, selecting a test suite that minimizes total execution time while preserving its coverage of user requirements is highly desirable for testing of software components. Yet the budget allocated to testing is limited, and optimizing the selection of test cases is a time-consuming activity. In practice, validation engineers solve the test suite reduction (TSR) problem through manual analysis or by approximation techniques. However, automated means to solve TSR instances efficiently are required when software components are developed in continuous delivery mode (Stolberg 2009). In fact, continuous integration involves frequent execution of regression test scripts to detect faults as early as possible, which means that automated selection of regression tests is indispensable in this context.

Overview

Formally speaking, given a set of requirements and a test suite that covers these requirements, the test suite reduction problem aims at finding a smallest subset of test cases in the test suite such that any requirement is covered at least once. By considering a cost value associated with each test case, a natural extension of this problem is to minimize the overall cost of the test suite, not just its size. Unfortunately, solving TSR is intractable in general (Harrold, Gupta, and Soffa 1993), and compromises have to be found either by adopting heuristics-based approximation algorithms or by using time-aware exact approaches.

Existing Results

The topic of test suite reduction has received considerable attention in the last two decades. Roughly speaking, we can distinguish greedy techniques (Tallam and Gupta 2005, Jeffrey and Gupta 2005), search-based testing techniques (Ferrer et al. 2015; Wang, Ali, and Gotlieb 2015), and exact approaches (Hsu and Orso 2009; Chen, Zhang, and Xu 2008; Campos et al. 2012; Li et al. 2014; Gotlieb and Marijan 2014).

Greedy techniques for test suite reduction usually select first the test cases that cover the most requirements and iterate until all requirements are covered. In the 1990s, Harrold, Gupta, and Soffa (1993) proposed a technique that approximates the computation of minimum-cardinality hitting sets. This work was further refined with different variable orderings (Offutt, Pan, and Voas 1995). More recently, Tallam and Gupta (2005) introduced the delayed-greedy technique, which exploits implications among test cases and requirements to refine further the reduced test suite. The technique starts by removing test cases that cover the requirements already covered by other test cases. Then it removes test requirements that are not in the minimized requirements set, and finally it determines a minimized test suite from the remaining test cases by using a greedy approach. Jeffrey and Gupta (2005) extended this approach by retaining test cases that improve a fault-detection capability of the test suite. Comparing to the paper by Harrold, Gupta, and Soffa (1993), the approach produces bigger solutions, but with higher fault-detection effectiveness.

One shortcoming of greedy techniques is that they only approximate global optima without providing any guarantee of optimality. Search-based testing techniques have also been tailored for test suite reduction. Wang, Ali, and Gotlieb (2015) explore classical metaheuristics such as hill climbing, simulated annealing, or weight-based genetic algorithms for (multiobjective) test suite reduction. By comparing 10 distinct algorithms for different criteria, they observed that random-weighted multiobjective optimization is the most efficient approach. However, by assigning weights at random, this approach is unfortunately not able to place priority over the various objectives. Ferrer et al. (2015) examine other algorithms based on metaheuristics. All these techniques can scale up to problems that have a large number of test cases and requirements, but they cannot explore the overall search space and thus they cannot guarantee global optimality.

On the contrary, exact approaches, which are based either on Boolean satisfiability or integer linear programming (ILP), can reach true global minima. The best-known approach for exact test suite minimization is implemented in MINTS (Hsu and Orso 2009). MINTS has been used to perform test suite reduction for various criteria including energy consumption on mobile devices (Li et al. 2014). Similar exact techniques have also been designed to handle fault localization (Campos et al. 2012). Generally speaking, the theoretical limitation of exact approaches is the possible early combinatorial explosion to determine the global optimum, which exposes these techniques to serious limitations even for small problems. A hybrid method based on ILP and search, called DILP, is proposed by Chen, Zhang, and Xu (2008), where a lower bound for the minimum is computed and a search for finding a smaller test suite close to this bound is performed. Recently, another ILP-based approach is proposed by Hao and colleagues (Hao et al. 2012) to set up upper limits on the loss of fault-detection capability in the test suite. Mouthuy, Deville, and Dooms (2007) proposed a constraint called SC for the set-covering problem. They created a propagator for SC by using a lower bound based on an ILP relaxation. Finally, Gotlieb and Marijan (2014) introduced an approach for test suite reduction based on the computation of maximum flows in a network flow. This initial idea has partly triggered the work reported in the present article.

Contributions

This article proposes a new approach of test suite reduction based on constraint programming (CP) and global constraints. Global constraints encode relations over a nonfixed number of variables with dedicated and efficient filtering algorithms. Our approach uses three special global constraints developed in CP, namely NVALUE, GLOBALCARDINALITY, and SCALAR_PRODUCT. NVALUE constrains the number of distinct values that can be taken by a set of variables (Pachet and Roy 1999), while GLOBALCARDINALITY generalizes this relation by considering explicit cardinality values for these variables (Régin 1996). SCALAR_PRODUCT simply encodes the scalar product between two vectors of variables as a relation. By combining these global constraints with advanced preprocessing rules and sophisticated structure-aware search heuristics, the proposed approach creates a constraint-optimization model that competes with the best known exact approach for test suite reduction, namely MINTS (Hsu and Orso 2009). As said above, associating a cost value to each test case is a natural extension of TSR. Indeed, such a cost value can represent or aggregate distinct notions such as execution time, code coverage, energy consumption (Li et al. 2014), or fault-detection capabilities (Campos et al. 2012). Using these cost values, TSR reduces to the problem of selecting a subset of test cases such that all the requirements are covered and the overall cost of the test suite is minimized. The proposed approach is also capable of optimizing an overall cost function depending of these cost values, while preserving the full coverage of requirements. We implemented our approach in a tool called Flower/C and performed a set of experiments with both randomly generated TSR instances and industrial instances. The experimental results show that Flower can be deployed into an industrial context and its route for exploitation is discussed. Next section formally defines TSR and gives some background on CP and global constraints. The following section shows three CP optimization models involving distinct combinations of global constraints. It also introduces preprocessing rules for TSR that can simplify the instances beforehand, and a dedicated search heuristics. The following section presents an experimental evaluation of the proposed models as well as a comparison with other approaches. Finally, the last sections draw perspectives for the industrial exploitation of the proposed approach and conclude the article.

Background

This section formalizes the test suite reduction problem and briefly reviews the notion of global constraints.

Test Suite Reduction

Test suite reduction aims to select a subset of test cases out of a test suite, which minimizes its overall cost, while retaining its coverage of requirements. Roughly speaking, a TSR instance is defined by an initial test suite T composed of m test cases $\{t_1, \ldots, t_m\}$, each test case being associated with a cost value noted $c(t_i)$, a set of n requirements $R = \{r_1, \ldots, r_n\}$, and a function called $cov(r_i)$ that maps each requirement r_i to the subset of test cases that cover it. We suppose that each requirement is covered by at least one test case and each test case covers at least one requirement. An example with five test cases and five requirements is given in table 1, where the value given in the table denotes the cost of each test case. Solving TSR aims at finding a subset of test cases such that every requirement is covered at least once, and the overall cost is minimized.

cost	r_1	r_2	r_3	r_4	r_5
t_a	2	2	-	-	-
t_b	1	-	1	-	-
t_c	-	3	3	-	3
t_d	-	-	-	2	2
t_e	-	-	-	1	-

Table 1. A TSR Problem Instance.

A labeled bipartite graph can be used to encode any TSR instance, with edges denoting the relation *cov* and labels denoting the costs over the test cases, as shown in figure 1. The overall cost of a test suite can be computed as the sum of each individual cost of its test cases, but other functions can be considered as well (for example, the max of costs). Note that the cost associated to any test case does not differ with respect to the covered requirement. The framework can be extended with a distinct cost for each requirement, but this brings more complexity without much benefit for validation engineers. Note also that the optimal solution shown in figure 1 is not unique. For example, $\{t_a, t_b, t_d\}$ covers all the requirements and has also $TotalCosts = c(t_a) + c(t_b) + c(t_d) = 5$. When the cost associated to each test case are all the same, then TSR reduces to the problem of finding a subset of minimal size.

Constraint Programming and Global Constraints

Constraint programming is a powerful declarative paradigm where logic and control are driven by constraint solving. Any constraint enforces a symbolic relation over a set of unknown variables, which take their values in a domain (Rossi, van Beek, and Walsh 2006). When the domain is finite, it can be mapped to a finite subset of integers without any loss. A con-

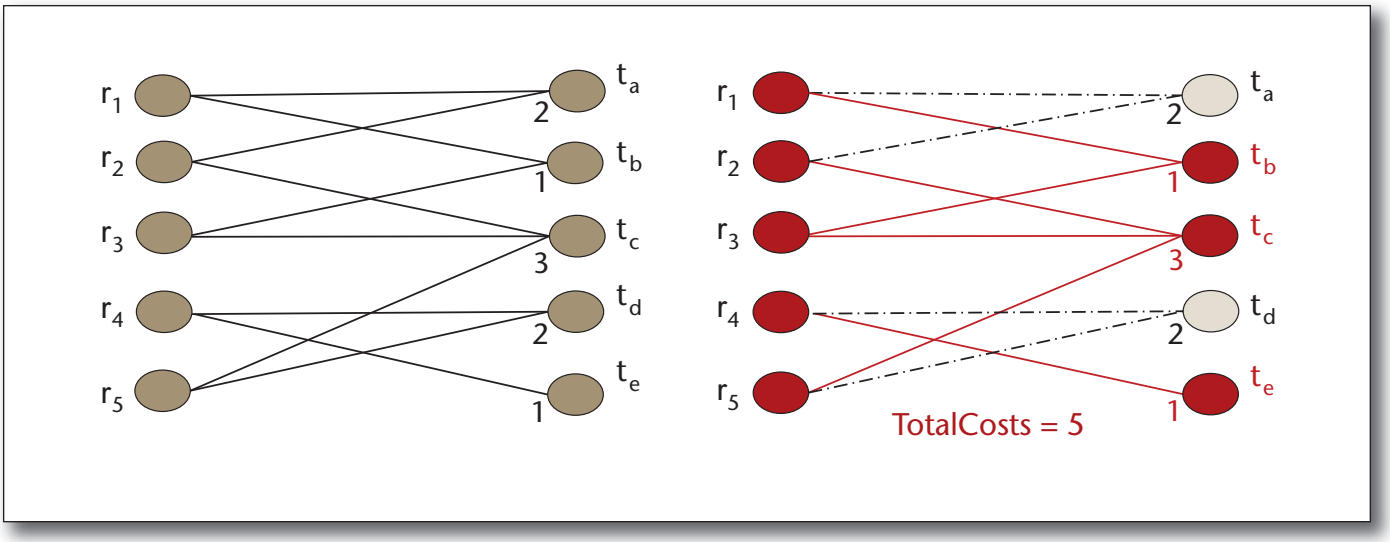


Figure 1. TSR as a Bipartite Graph with an Optimal Solution.

(a) Bipartite Graph. (b) Optimal Solution.

straint program over finite domain variables is a finite set of constraints, which come with filtering algorithms. These algorithms prune the domains of the constraint variables from some of their inconsistent values. For instance, if X takes an unknown value in the finite domain {2, 3, 5} and Y takes a value in {3, 4, 5, 6} then the filtering algorithm associated with $X = Y$ can prune the domains of both X and Y to {3, 5}. In this context an assignment is just a mapping from any variable, noted with uppercase letters in the article, to a value from its respective domain, noted with lowercase letters. A constraint program is satisfiable when there exists at least one assignment that satisfies all the constraints. It is unsatisfiable otherwise. Among the satisfiable assignments, some can minimize a cost function and thus, CP can be used to solve optimization problems as well.

In CP, two types of constraints can be distinguished, namely the relations that hold over a known number of variables (typically 1, 2, or 3) and relations that hold over a nonfixed number of variables. Constraints from this latter category are called global constraints, especially when they implement dedicated and efficient filtering algorithms.

A first example of global constraints is given by the following constraint:

> Definition 1. (NVALUE [Pachet and Roy 1999]).
> Let N be a domain variable and V be a vector of domain variables, NVALUE(N, V) holds iff the number of distinct values in V is equal to N.

For instance, NVALUE(N, [3, 1, 3]) entails $N = 2$ and is solved, NVALUE(3, [X_1, X_2]) is unsatisfiable, and NVALUE(1, [X_1, X_2, X_3]) entails $X_1 = X_2 = X_3$.

Another example of global constraint, which generalizes NVALUE is now given:

> Definition 2. (GLOBALCARDINALITY [Régin 1996]).
> Let $T = (T_1, \ldots, T_n)$ be a vector of domain variables, let $d = (d_1, \ldots, d_m)$ be a vector of distinct integers, and let $C = (C_1, \ldots, C_m)$ be a vector of domain variables, GLOBALCARDINALITY(T, d, C) holds iff for each i ∈ 1..m the number of occurrences of d_i in T is C_i. The C_i variables are the occurrence variables of the constraint.

For instance, GLOBALCARDINALITY(((T_1, T_2, 5), (5, 7), (C_1, C_2)) prunes the domains of T_1 and T_2 to {5, 7}, the domain of C_1 to {1, 2, 3} and the domain of C_2 to {0, 1, 2}. A polynomial filtering algorithm for this constraint was given by Regin (1996).

CP Models of the TSR Problem

In this section, we present distinct constraint-optimization models based on NVALUE, GLOBALCARDINALITY for the TSR problem.

A Naive Model (NVALUE)

In CP, TSR can easily be encoded with the following scheme: each requirement to be covered can be associated with a domain variable R having as finite domain, which is composed of the test cases that cover the requirement. More precisely, R belongs to {t_1, …, t_n}, where each t_i corresponds to an integer associated with a test case that covers R. So, for example, the instance reported in table 1 can be encoded as follows:

$R_1 \in \{1, 2\}$, $R_2 \in \{1, 3\}$, $R_3 \in \{2, 3\}$, $R_4 \in \{4, 5\}$, $R_5 \in \{4\}$

where t_a is associated with 1, t_b is associated with 2, and so on.

Figure 2 shows a first constraint-optimization program for an instance of test suite reduction.

This model aims to minimize the number of different values that can be taken by R_1, …, R_n, that is, the number of distinct test cases that cover all the

Minimize Ns.t.NVALUE(N, (R_1, . . . ,R_n))
for i = 1 to n s.t. $R_i \neq R_j$ for any j do $\sum_i c(t_{R_i})$ = *Total Costs.*

Figure 2. A First Constraint-Optimization Model for TSR (Naive).

Maximize N s.t.

GLOBALCARDINALITY((R_1, . . . ,R_n); (t_1, . . . , t_m); (O_1, . . . ,O_m))∧
GLOBALCARDINALITY((O_1, . . . ,O_m); (0); (N))
for i = 1 to n s.t. $R_i \neq R_j$ for any j do $\sum_i c(t_{R_i})$ = *Total Costs.*

Figure 3. A Constraint-Optimization Model for TSR Based on GLOBALCARDINALITY.

requirements. Using NVALUE enables the minimization process to reduce the number of test cases, while the second part of the model computes the sum of costs. This model is naive for two reasons: firstly, it does not guarantee finding the minimum of costs even though it finds the minimum number of test cases (issue 1), and secondly, it allows us only to search on a tree composed of the requirement variables (issue 2). In fact, the only variables of this model are the R_i, which means that branching on the selection of test cases is unfortunately not possible. For example, selecting first the test cases that cover the most requirements while searching for a minimum is not possible. In order to tame this problem (issue 2), another model based on GLOBALCARDINALITY can be proposed.

A Model with GLOBALCARDINALITY(*GCC*[2])

Let O_i be a domain variable representing the number of times test case t_i is selected to cover R_1, …, R_n. The model shown in figure 3 addresses TSR by using two GLOBALCARDINALITY constraints.

The first GLOBALCARDINALITY enforces the coverage relation between test cases and requirements by constraining the occurrence variables O_i, while the second GLOBALCARDINALITY counts the number of 0 (zeros) in the list of occurrence variables. This allows the model to constrain the selection of test cases by maximizing the number of unselected test cases. Thus, branching on the number of occurrences of each test case becomes possible with this model. Still, this model does not address issue 1 mentioned above, as it does not guarantee to reach the minimum of the overall cost of test cases. Another model can be proposed to deal with both issue 1 and issue 2.

An Optimized Model *(Mixt)*

In this third model, (R_1, …, R_n.), (O_1, …, O_m) are decision variables, only known through their domain. The Boolean variables B_1, …, B_m are local variables introduced to establish the link with costs. By using the global constraint SCALAR_PRODUCT((B_1, …, B_m), (c_1, …, c_m), *TotalCosts*), which enforces the relation

$$TotalCosts = \sum_{1 \leq i \leq m} B_i * c_i$$

this model actually minimizes *TotalCosts,* the sum of the costs of selected test cases. In fact, the nonnull B_i variables correspond to the selected test cases. The constraint GLOBALCARDINALITY allows us to constrain the variables O_i, which are associated with the number of selected test cases. This model can be solved by searching the space composed of the possible choices for (R_1, …, R_n), (O_1, …, O_m). Interestingly, it allows us to branch either on the choice of requirements or on the choice of test cases. Hence, it addresses both issues 1 and 2. An optimal solution of this model is an optimal solution of TSR and vice versa, as proved by the following sketch of proof.

(⇒) An optimal solution of TSR corresponds to the assignment of (R_1, …, R_n) with test cases that mini-

Minimize *TotalCosts* s.t.
GLOBALCARDINALITY((R$_1$, . . . ,R$_n$); (1, . . . ,m); (O_1, . . . ,O_m)),
for i = 1 to m do $B_i = (O_i > 0)$,
SCALAR_PRODUCT((B_1, . . . ,B_m); (c_1, . . . , c_m), *TotalCosts*).

Figure 4. A Constraint-Optimization Model for Solving TSR.

mize the sum of costs. Let us call $\{t_p, \ldots, t_q\}$ this solution and minimum this sum. This is also an optimal solution of our model. In fact, the variables $\{O_p, \ldots, O_q\}$ are strictly positive because their associated test case is selected in the solution through GLOBALCARDINALITY, which means that only the corresponding $\{B_p, \ldots, B_q\}$ are equal to 1 and thus SCALAR_PRODUCT((B_1, …, B_m), (c_1, …, c_m), *TotalCosts*) is equal to *minimum*.

(⇐) An optimal solution m of our constraint-optimization model is also an optimal solution of TSR. In the model, *TotalCosts* is assigned to the sum of costs of selected test cases and there exists no other assignment of B_i, which gives a smaller value than m. Then, it means that m is actually the minimum cost of the TSR instance, and the test cases selected by the B_i are the solution of this problem.

Even if the model given in figure 4 is generic, it involves searching a space of exponential size $O(D_n)$ where D denotes the size of the greatest domain of any requirement variable and n is the number of test cases. This does not come as a surprise as TSR has been shown to be NP-hard (Hsu and Orso 2009).

Solving TSR can be improved by considering a number of optimizations, including preprocessing rules and specialized search heuristics.

Preprocessing

Preprocessing can be used to reduce the size of the problem beforehand, by using the following rules:

Rule 1.
For two test cases t_1, t_2, if all the requirements covered by t_1 are included in the subset of requirements covered by t_2, then t_1 can be safely ignored during search, as it is always be preferable to select t_2 instead of t_1.

Rule 2.
Conversely, for two requirements r_1, r_2, if all test cases covering r_1 are included in the subset of test cases covering r_2, then r_2 can be safely removed from the set of requirements to be covered. Indeed, any test case covering r_1 will automatically cover r_2 as well.

Rule 3.
If there is a requirement that is covered by only a single test case t then t must be included in the solution set. Figure 5 illustrates these preprocessing rules.

A Dedicated Heuristic

Search heuristics include strategies for selecting a variable to be enumerated first and a value to be selected first. Both strategies can be tuned by the available constraints and variables of the model. The first idea is to use the classical first-fail principle, which selects first the variable representing the requirement that is covered by the least number of test cases. As all the requirements have to be covered, it means that these test cases are most likely to be selected. However, this strategy ignores the selection of the test case having the least cost or the test cases covering the most requirements. Regarding value selection, it is thus better to define a special heuristic for our problem.

Unlike the static variable selection strategy used in greedy algorithms, such as, for example, the selection of variables based on the number of covered requirements, our TSR-dedicated strategy is dynamic and the ordering is revised at each step of the selection process. It selects first the variable O_i associated with the test case with the smallest cost. Then, among the remaining test cases that cover any requirement not yet covered, it selects the variable O_j with the smallest cost and iterates until all the requirements are covered. In case of a choice that does not lead to a global minimum, the process backtracks and selects a distinct test case, not necessarily associated with the smallest cost. Regarding the value-selection strategy, each time a value selection is made, our TSR-dedicated heuristics select first the test cases that cover the most requirements. Property 1 formalizes this idea.

Property 1.
Let each test case t_i be represented by an occurrence variable O_i taking its values in $0..max_i$ where max_i is dynamically updated with the current partial assignment. Then, for each solution X of the TSR problem with cost $f(X)$ where $Oi = n_i$ such that $0 < n_i < max_i$ (strict inequalities), there is at least one other solution Y with cost $f(Y) \leq f(X)$ where either $O_i = 0$ or $O_i = max_i$.

Note that our proposed TSR-dedicated heuristic is incomplete, meaning that some parts of the search tree can remain unexplored. Indeed, symmetrical solutions can be ignored as explained in figure 6, but, refer-

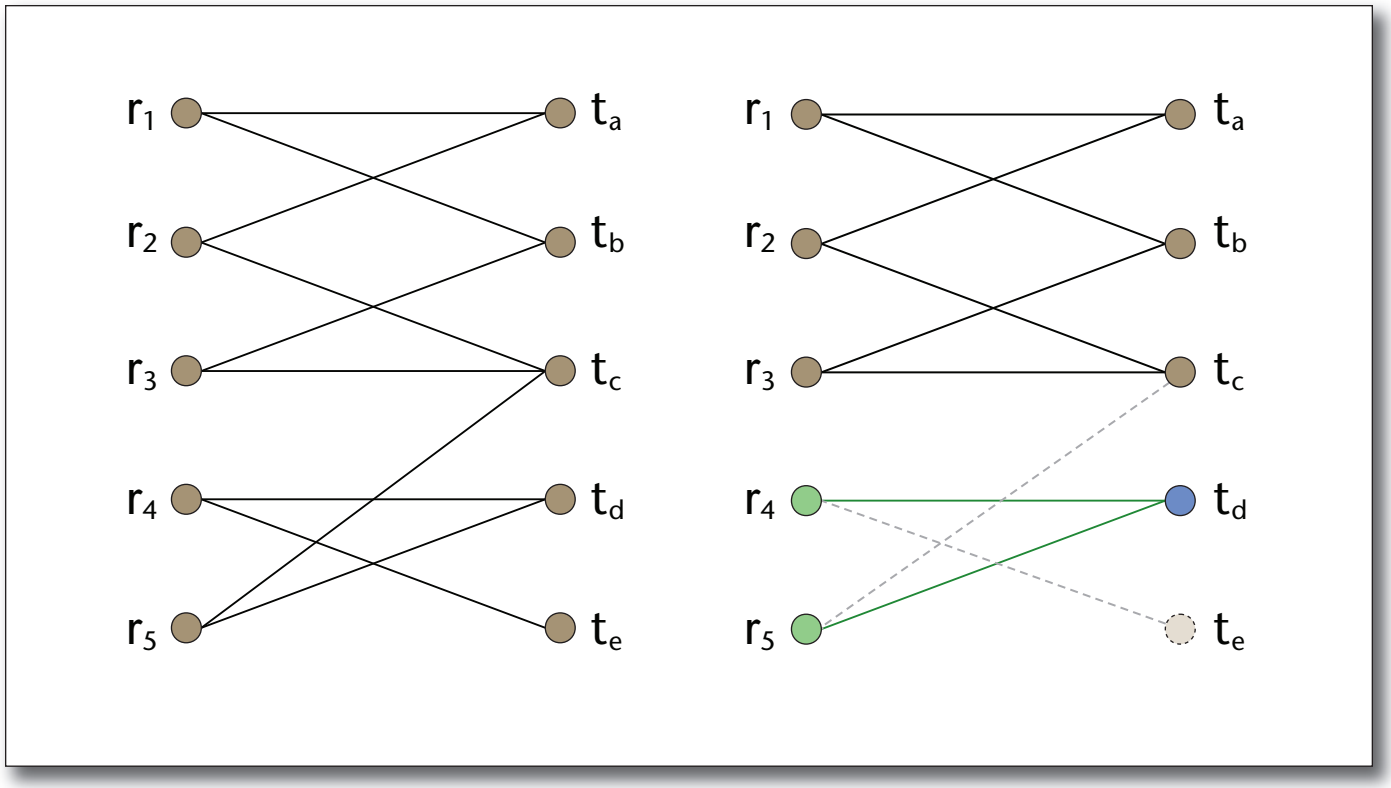


Figure 5. Preprocessing.

The edge (r_4, t_c) can be safely removed by rule 1, since r_4 is also covered by t_d, which covers another requirement. Rule 2 allows one to remove edges (r_5, t_c) and (r_5, t_d) as any test case covering r_4 also covers r_5. Finally, t_d is included in the solution set by rule 3, since it remains the only covering r_4 and r_5.

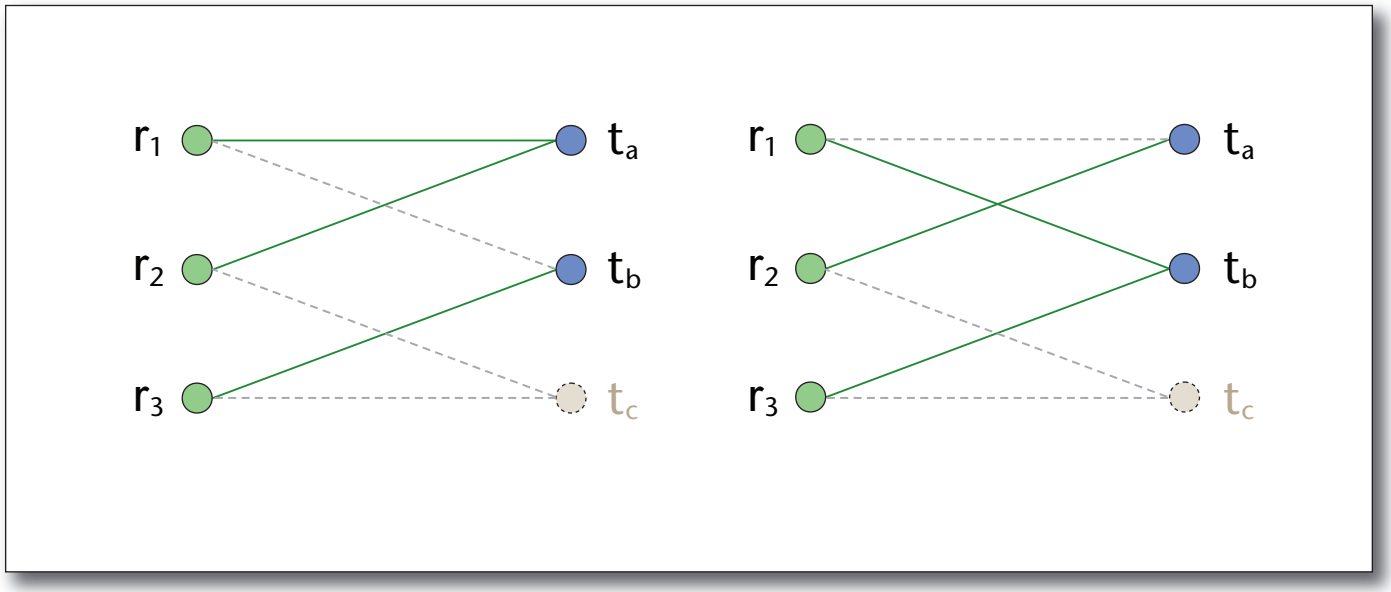
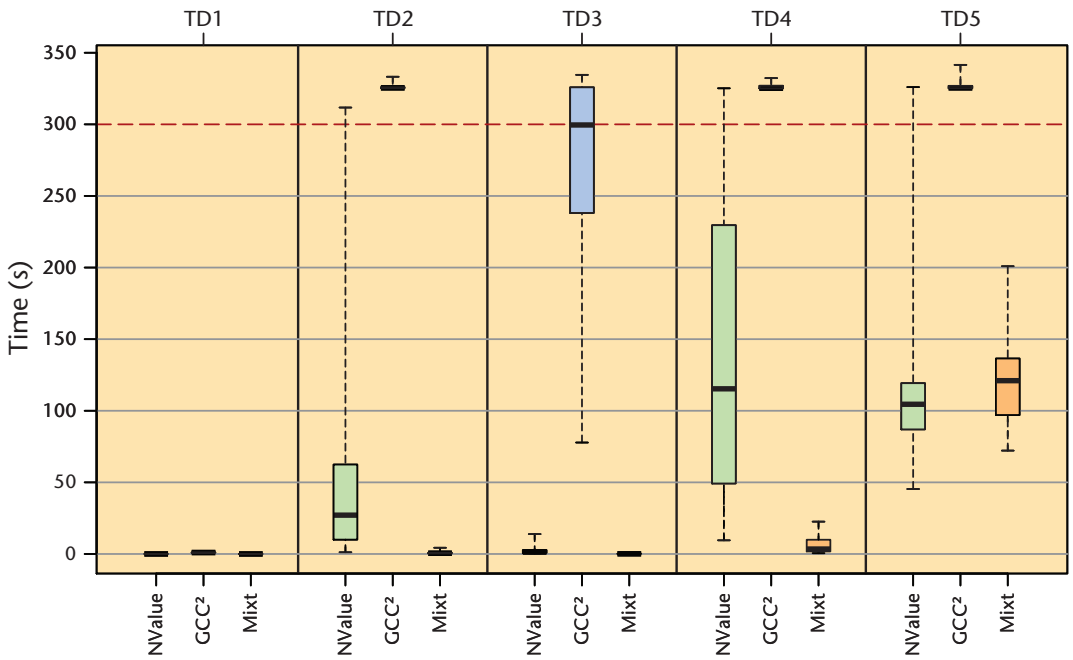


Figure 6. Two Symmetrical Solutions for CP, a Single Solution of TSR.

In both graphs, the same optimal test suite is obtained, $T' = \{t_a, t_b\}$. However, it is associated with distinct solutions for CP because the R_i are assigned to distinct values: on the left, R_1 is assigned to t_a while on the right R_1 is assigned to t_b. With our dedicated heuristic, an arbitrary selection is made, for example,, the occurrence variable O_a representing t_a is assigned to 2 as shown on the left. In case of necessary backtrack, it would be assigned to 0, but never to 1, as shown on the right.



	TD1	TD2	TD3	TD4	TD5
Requirements	20	90	60	60	30
Test cases	70	100	100	200	500
Density	8	20	20	20	8

Figure 7. Comparison of CPU Time for the CP Models.

(time-out = 300 seconds).

ring to Property 1, our TSR-dedicated heuristic guarantees that at least one optimal solution is found.

Experimental Evaluation

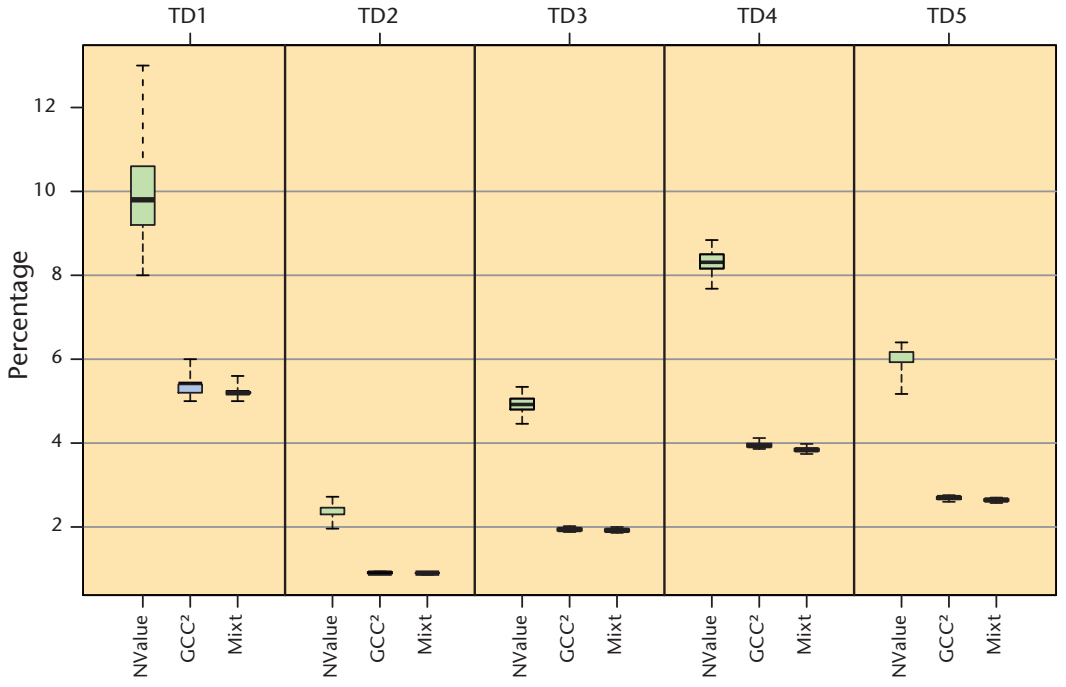
We implemented the constraint-optimization models and search heuristic described above in a tool called Flower/C, by using SICStus Prolog and its clpfd library. This library implements a finite domains constraint solver. Flower/C reads a file that contains the data about test cases, the covered requirements, and the costs associated to test cases and processes these data by constructing a corresponding bipartite graph and tuning the constraint-optimization models for solving the TSR instance. Solving the model involves preprocessing and search among feasible solutions with the proposed TSR-dedicated search heuristics. These steps are encoded in SICStus Prolog.

Both random and industrial instances of TSR were considered for the experimental evaluation. For random problems, we created a generator of TSR instances, which takes several parameters as inputs, such as the number of requirements, the number of test cases along with their associated costs, and the density of the relation *cov,* which is captured with *d* representing the maximum arity of any links in *cov.* The generator draws a number *a* at random between 1 and *d* and creates *a* edges in the bipartite graph, which represents *cov.* For industrial instances, we used data (test cases, coverage, costs) from regression testing of communication software provided by industry.

All our experiments were run on a standard i7-2929XM CPU machine at 2.5 GHz with 16 GB RAM.

Comparison of the Various CP Models

Figure 7 compares the CPU time required for finding optima with the three distinct CP models. In order to keep the comparison fair, we ignored costs in this first experiment so that optimality was only considered



	TD1	TD2	TD3	TD4	TD5
Requirements	250	500	1000	1000	1000
Test cases	500	5000	5000	5000	7000
Density	20	20	20	8	8

Figure 8. Comparison of Reduction Rate.

(as percentage of remaining test cases, time-out = 30 seconds).

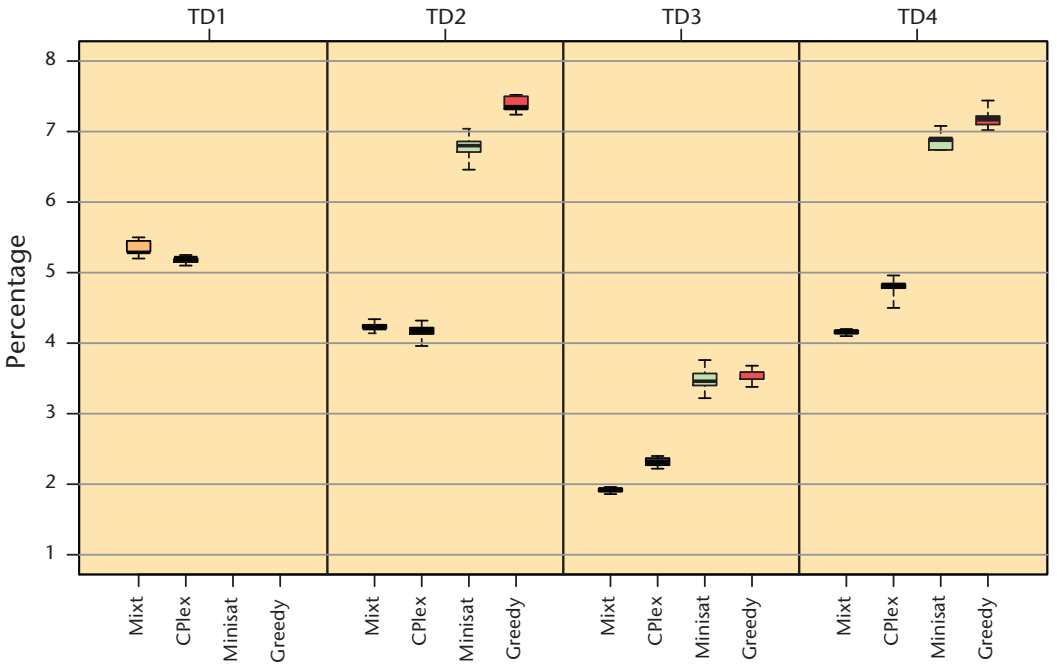
on the number of selected test cases. In each data set, 20 random samples were generated. For all but TD1, the *GCC*^{2} model times out (after 300 seconds). For the NVALUE model, we observe that the variation is very high in most cases (TD2, TD4, TD5). Sometimes, this models also times out. On the contrary, the Mixt model does not present much variation, which means that the TSR-dedicated heuristic is robust and useful in most cases. In figure 8, we compute the percentage of test cases remaining in the solution set after 30 seconds. A good reduction rate in a limited amount of time is crucial for any industrial adoption, as test suite reduction has to be performed within a continuous integration process, where the reduction is computed each time a new software release is committed.

We observe in this experiment that NVALUE is outperformed by both *GCC*^{2} and Mixt, which both reach the same reduction rate. This is due to the selection of the branching heuristic, which is different for the NVALUE model, where only the requirement variables are available for branching.

Comparison with Other Approaches

In the first experiment, we compared our implementation, Flower/C, with three other approaches, namely MINTS/MiniSAT+, MINTS/CPLEX, and Greedy on randomly generated instances. MINTS is a generic tool that handles the test suite reduction problem as an integer linear program (Hsu and Orso 2009). For each requirement to be covered, a linear inequality over Boolean variables is generated that enforces the coverage of the requirement. The Boolean variables ensure the selection of test cases. MINTS can be interfaced with distinct black-box constraint solvers, including MiniSAT+ and CPLEX. We also implemented a simple greedy approach for solving the TSR problem, which is based on a static ordering of the test cases covering the most requirements.

Figure 9 shows the results of comparison of the



	TD1	TD2	TD3	TD4
Requirements	1000	1000	1000	2000
Test cases	2000	5000	5000	5000
Density	20	7	20	20

Figure 9. Results of Comparison of the Four Approaches in Terms of Reduction Rate.

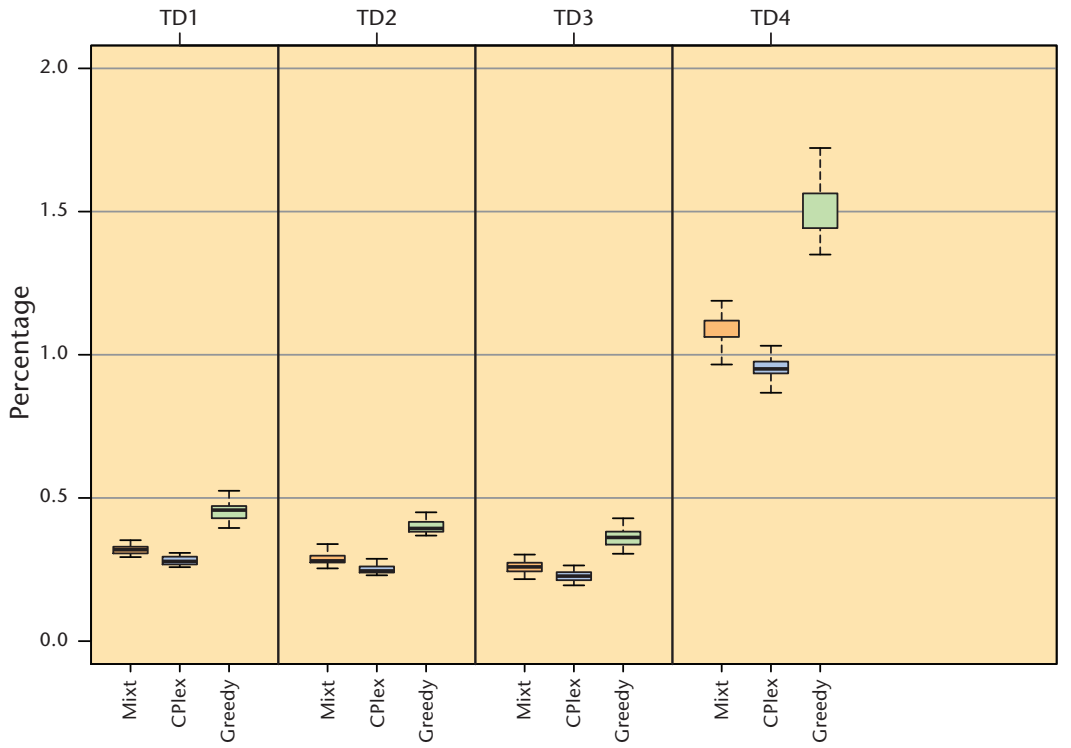
Comparison of reduction rate of Flower/C (Mixt), MINTS/MiniSAT+, MINTS/CPLEX, and Greedy on random instances with uniform costs (time-out = 60 seconds).

four approaches in terms of reduction rate, obtained in 60 seconds running time. In this experiment, the same cost values are used for all test cases. We observe that for the four groups of random instances (ranging from 1000 to 2000 requirements with two distinct maximal density values, 7 and 20), Flower/C achieves equal or better results than all the three other approaches in terms of reduction rate in a limited amount of time. Regrading the two last groups (TD3 and TD4), Flower/C performs strictly better than all the three other approaches, reaching exceptional reduction rates. It is worth noticing that for each group 100 random instances were generated, which means that the results are quite stable with respect to random variations. It is also quite clear that CPLEX performs much better for these problems than MiniSAT+. This does not come as a surprise as TSR has a simple formulation in terms of integer linear program (CPLEX), while MiniSAT+ requires translation into SAT clauses.

In the second experience, reported in figure 10, we gave different cost values to each instance by making the random generator select at random a value between 1 and a maximum value for each test case. In this experiment, no result is reported for MINTS/MiniSAT+ because the objective function as the sum of cost values cannot easily be encoded into Boolean SAT clauses. Therefore, only the results with MINTS/CPLEX, Flower/C, and Greedy are reported. Figure 10 shows that the results are in favor of MINTS/CPLEX on the four groups of random instances, which means that more effort is needed to find better CP models and search heuristics when costs are present.

Evaluation on Industrial Instances

We conducted the third experiment on industrial



	TD1	TD2	TD3	TD4
Requirements	1000	1000	1000	2000
Test cases	5000	5000	5000	5000
Density	20	20	20	20
Cost	50	100	1000	100

Figure 10. Results of Comparison of Reduction Rate on the Four Groups of Random Instances with Nonuniform Cost Values

instances coming from an industrial partner involved in the development of communication systems. The data were extracted from the continuous integration process during one cycle and converted to the specific format processed by Flower/C. The results are shown in table 2. The CPU time required to solve industrial instances of TSR shows that the Mixt model performs the best. Interestingly, the reduction rate shown in the fifth row (obtained with Mixt) is quite high for all the five industrial instances (ranging from 61.80 percent to 26.67 percent). This shows the importance of solving TSR in practice for our industrial partner. Finally, the last row of the table shows the number of removed requirements during preprocessing.

Evaluating Preprocessing Rules

We performed other experiments to evaluate precisely the effectiveness of preprocessing rules for both randomly generated TSR problems and industrial instances, as compared to the preprocessing used in MINTS/CPLEX. In figure 11, we evaluated the importance of MINTS/CPLEX's own preprocessing[1] in reaching an optimal solution by observing the size of the solution sets at different time points, and compared it with our own preprocessing. We found some data sets where Flower/C's preprocessing rules were more efficient than MINTS/CPLEX's preprocessing as shown in figure 11. However, there are also other cases where the opposite was observed. In fact, Flower/C preprocessing rules cannot be well compared with

Requirements	59	53	50	37	37	156
Test cases	107	90	93	100	100	377
CPU Time Nvalue(s)	0.00	0.10	0.01	0.01	0.01	0.03
CPU Time GCC2(s)	300.00	102.00	91.80	59.16	6.09	300.00
CPU Time Mixt(s)	0.00	0.01	0.00	0.00	0.00	0.01
Reduction rate (%)	28.97	26.67	29.03	40.00	37.00	61.80
Removed requirements (%)	32.00	30.19	30.00	32.43	45.95	44.87

Table 2. Evaluation of Flower/C on Industrial Instances.

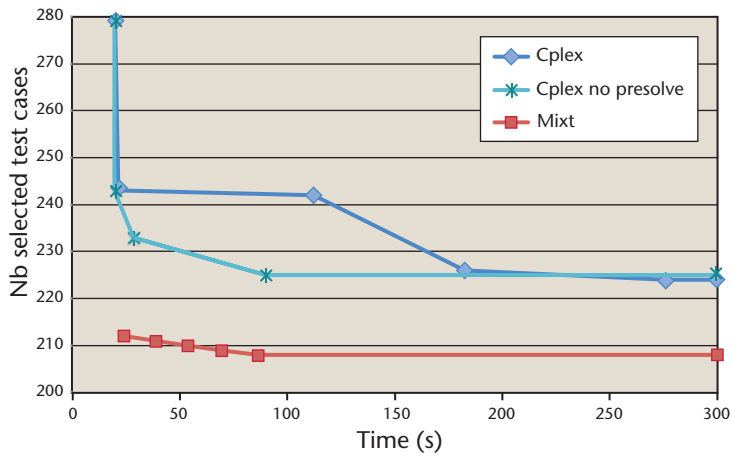


Figure 11. Evaluation of CPLEX Preprocessing versus Mixt.

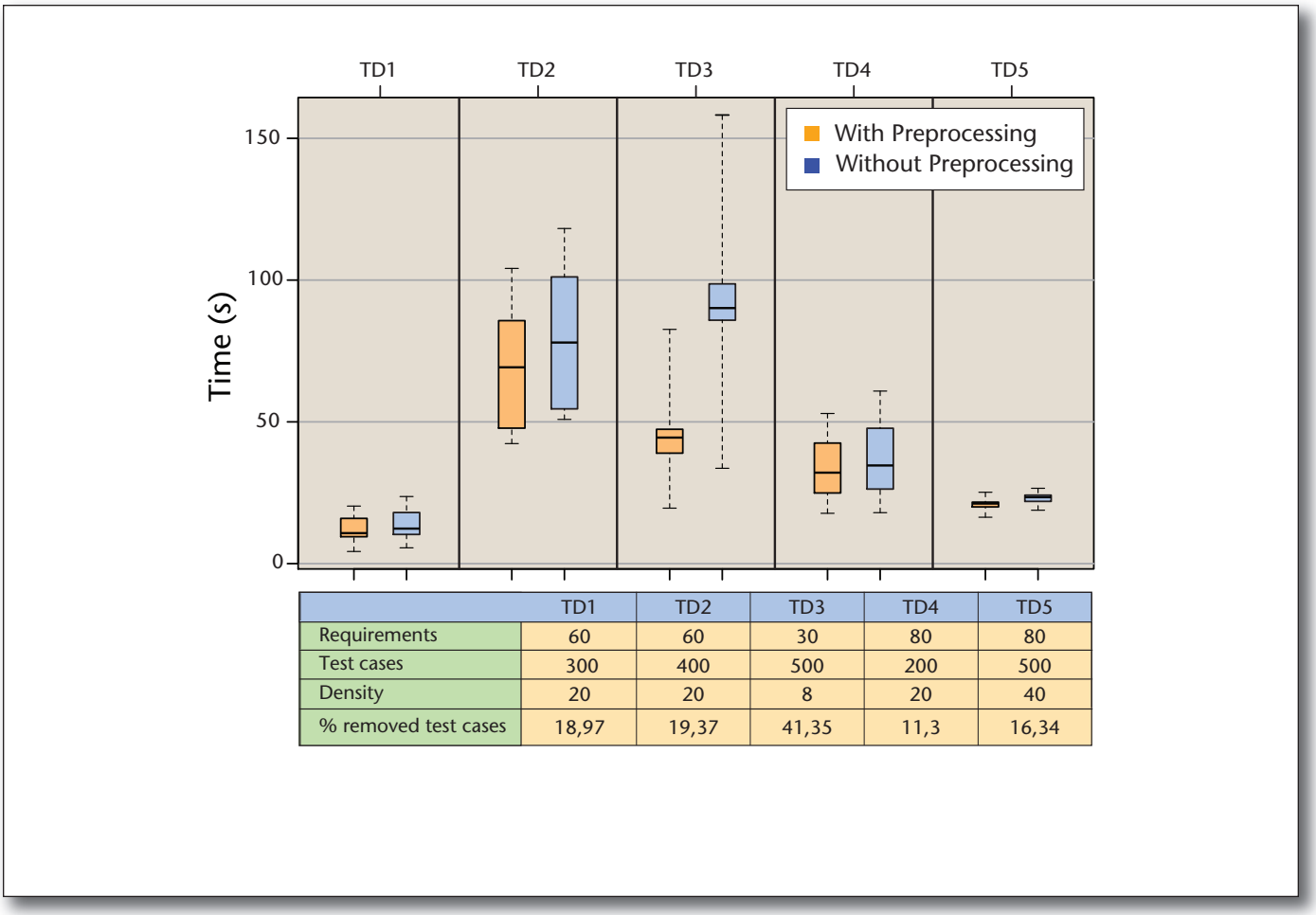
MINTS/CPLEX's preprocessing as both tools work on very different data structures. Finally, we looked at the gain in terms of CPU time while activating and deactivating Flower/C's preprocessing as shown in figure 12. The gain is not really spectacular even if the percentage of removed test cases is quite good.

Comparison of Several Search Heuristics

Figure 13 shows the CPU time for three variable-selection heuristics (that is, max, min, ff) used together with the CP Mixt model, while the value-selection heuristic remains unchanged. The heuristic max selects the variable with the greatest upper bound, min selects the variable with smallest lower bound, while ff selects the variable with the smallest domain. In this experiment, max achieves better result by selecting the occurrence variable that has the greatest arity, that is, the one associated with a test case that covers the most requirements. We selected it to be employed with our CP Mixt model.

Figure 14 compares different value-selection heuristics with max, including our own heuristics called value(enum), step, and bisect. The heuristic step branches on all the values of the domain of occurrence variables in increasing order, bisect performs domain-splitting using the middle point of the domain of each variable, while our heuristic only branches on *Max* and 0 for domain {0, 1, …, *Max*}.

As expected, figure 14 shows much better results for our heuristic. However, it is worth keeping in



	TD1	TD2	TD3	TD4	TD5
Requirements	60	60	30	80	80
Test cases	300	400	500	200	500
Density	20	20	8	20	40
% removed test cases	18,97	19,37	41,35	11,3	16,34

Figure 12. Comparison of CPU Time Versus Preprocessing.

mind that our strategy is incomplete. Even though it may not explore parts of the search space that contain optimal solutions, it preserves at least one optimal solution. When sufficient time is allocated to the search, it always has the opportunity to reach an optimal value faster than complete heuristics.

Path Toward Deployment

The work presented in this article has been motivated by the industrial problem of software regression testing in the communication domain. Software is characterized by a high degree of configurability, providing flexibility for end users to adapt systems to their specific needs. However, configurability involves higher complexity of software testing, and typically larger test suites. At the same time, software is developed following a continuous integration practice, which is characterized by a short test feedback loop. Extensive test suites, limited test time, and high requirements for software quality together set the challenge of implementing an efficient test suite

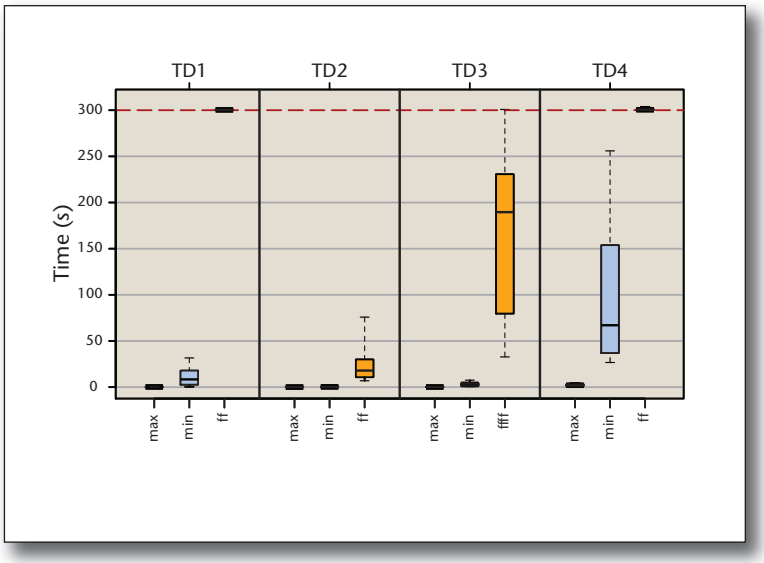


Figure 13. CPU Time of the Variable-Selection Heuristics.

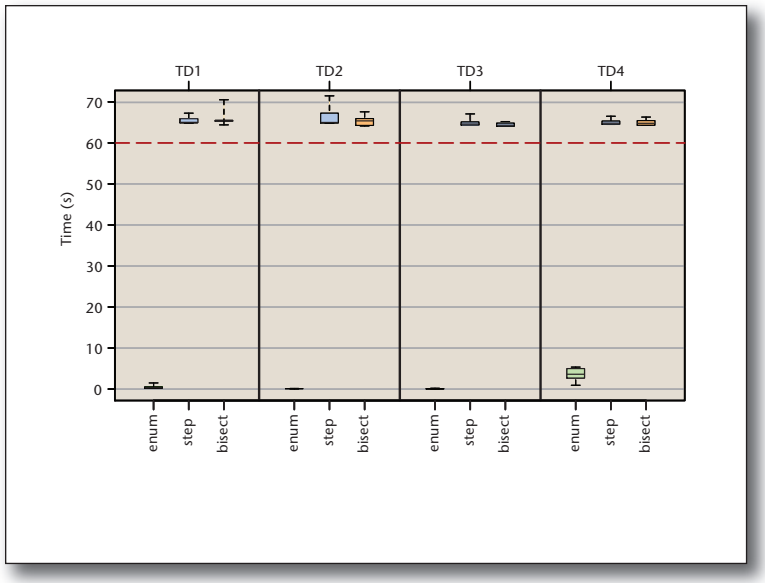


Figure 14. CPU Time of the Value-Selection Heuristics.

reduction that is able to reduce costs and improve the effectiveness of regression testing in practice.

Our approach has been designed in interaction with test engineers. The process involved observing current test selection practice done manually by engineers, interviews with engineers to understand the objective behind test selection, and capturing typical metrics such as the frequency and size of regression test runs, regression test selection criteria, and test failure rates. We evaluated the performance of the approach on several instances of industrial test suites coming from the described domain. The results shown that the approach is applicable, and that it can improve the speed and quality of regression test selection in continuous integration in practice. However, more work is needed to enable seamless integration with an industrial testing framework.

We see the deployment of our approach as a staged process. As part of first-phase deployment, we developed a prototype tool, and we provided training for engineers on the key concepts of CP used in our approach. We deem this step necessary for adopting the approach by industry, as we observed a limited familiarity with CP in this particular setting. We envisage full deployment as an iterative process, where we will be enhancing the tool functionality and usability based on industry feedback. The enhancements will relate more flexible test optimization, including test prioritization, to support achieving various testing objectives. At the final stage, we expect the tool to be deployed organizationwide, supporting cost-effective test automation much needed in complex continuous integration environments.

Conclusion

This article presents the application of CP techniques using global constraints to improve the cost efficiency of software regression testing. Three CP models using the global constraints NVALUE and GLOBALCARDINALITY are proposed to encode test suite reduction (TSR) in such a way to ensure the coverage of all user requirements while additionally minimizing the overall cost of a test suite. According to our knowledge, this is the first time that these global constraints are applied to the reduction of test suites in software testing. We find that some preprocessing rules can drastically reduce the size of initial problem instances and that our proposed TSR-dedicated strategy can outperform other more classical labeling heuristics such as those based on the first-fail principle.

Note that the proposed labeling heuristic is not complete, which means that it does not explore the overall search space. This may explain why it has a stronger competitive advantage over other heuristics. At the same time, this incompleteness in the search does not compromise reaching a true global optimum for the constraint-optimization model, since only symmetrical solutions are removed. The three CP models are compared on random instances with the state-of-the-art academic tool MINTS interfaced with MiniSAT+ and CPLEX. Our results show that CP is efficient and competitive with MINTS in terms of percentage of test suite reduction.

Furthermore, we evaluate our approach on industrial regression testing on communication software systems. Initial results show that the approach is useful for improving the speed and quality of regression test selection in continuous integration. However, there are challenges in fully applying complex CP techniques to testing in practice. Although the proposed search heuristics are quite efficient to prune the search space beforehand, we do not know yet if other heuristics could be more beneficial. Exploring this question is part of our planned further work. We also want to ease the adoption of CP-based solutions in industry by the design of tailored software tools encapsulating the complexity of constraint solving and providing scalable integrations with software testing tool chains.

Acknowledgments

This work is supported by the Research Council of Norway through the Certus SFI grant. Initial experiments were performed by Alexandre Pétillon. He was supported by Mats Carlsson from the Swedish Institute in Computer Science and got access to real data provided by Marius Liaaen from Cisco Systems, Norway. We would like to thank them all for their enthusiasm and support of this work. We are grateful to Jean-Charles Régin for fruitful discussions on the topic of the article.

Note

1. CPLEX processing can be deactivated on demand.

References

Campos, J.; Riboira, A.; Perez, A.; and Abreu, R. 2012. Gzoltar: An Eclipse Plug-In for Testing and Debugging. In *IEEE/ACM International Conference on Automated Software Engineering (ASE'12),* 378–381. New York: Association for Computing Machinery.

Chen, Z.; Zhang, *X.*; and Xu, B. 2008. A Degraded ILP Approach for Test Suite Reduction. Paper presented at the Twentieth International Conference on Software Engineering and Knowledge Engineering (SEKE'08), San Francisco, CA, USA, July 1–3.

Ferrer, J.; Kruse, P.; Chicano, F.; and Alba, E. 2015. Search Based Algorithms for Test Sequence Generation in Functional Testing. *Information and Software Technology* 58(0): 419–432.

Gotlieb, A., and Marijan, D. 2014. Flower: Optimal Test Suite Reduction as a Network Maximum Flow. In *International Symposium on Software Testing and Analysis,* ISSTA'14. New York: Association for Computing Machinery.

Hao, D.; Zhang, L.; Wu, *X.*; Mei, H.; and Rothermel, G. 2012. On-Demand Test Suite Reduction. In *Proceedings of the 34th International Conference on Software Engineering* (ICSE'12), 738–748. Los Alamitos, CA: IEEE Computer Society.

Harrold, M. J.; Gupta, R.; and Soffa, M. L. 1993. A Methodology for Controlling the Size of a Test Suite. *ACM Transactions on Software Engineering and Methodology* (TOSEM) 2(3): 270–285.

Hsu, H.-*Y.*, and Orso, A. 2009. MINTS: A General Framework and Tool for Supporting Test-Suite Minimization. In *Proceedings of the 31st International Conference on Software Engineering* (ICSE'09), 419–429. Los Alamitos, CA: IEEE Computer Society.

Jeffrey, D., and Gupta, N. 2005. Test Suite Reduction with Selective Redundancy. In *Proceedings of the 21st IEEE International Conference on Software Maintenance,* 549–558. Los Alamitos, CA: IEEE Computer Society.

Li, D.; Jin, Y.; Sahin, C.; Clause, J.; and Halfond, W. G. J. 2014. Integrated Energy-Directed Test Suite Optimization. In *International Symposium on Software Testing and Analysis,* ISSTA'14. New York: Association for Computing Machinery.

Mouthuy, S.; Deville, *Y.*; and Dooms, G. 2007. Global Constraint for the Set Covering Problem. Paper presented at Journées Francophones de Programmation par Contraintes,Université d'Orléans, Orléans, France, 3–5 June.

Offutt, A. J.; Pan, J.; and Voas, J. M. 1995. Procedures for Reducing the Size of Coverage-Based Test Sets. Paper presented at the Twelfth International Conference on Testing Computer Software, Washington, DC, June.

Pachet, F., and Roy, P. 1999. Automatic Generation of Music Programs. In *Principles and Practice of Constraint Programming,* volume 1713 of Lecture Notes in Computer Science. Berlin: Springer.

Régin, J.-C. 1996. Generalized Arc Consistency for Global Cardinality Constraint. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference*, 209–215. Menlo Park, CA: AAAI Press.

Rossi, F.; van Beek, P.; and Walsh, T. 2006. *Handbook of Constraint Programming (Foundations of Artificial Intelligence).* Amsterdam: Elsevier Science.

Stolberg, S. 2009. Enabling Agile Testing Through Continuous Integration. In *Proceedings of the 2009 Agile Conference,* 369–374. Los Alamitos, CA: IEEE Computer Society.

Tallam, S., and Gupta, N. 2005. A Concept Analysis Inspired Greedy Algorithm for Test Suite Minimization. In *Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE'05,* 35–42. New York: Association for Computing Machinery.

Wang, S.; Ali, S.; and Gotlieb, A. 2015. Cost-Effective Test Suite Minimization in Product Lines Using Search Techniques. *Journal of Systems and Software* 103 (May): 370–391.

Arnaud Gotlieb Ph.D., is a senior research scientist at Simula Research Laboratory, where he leads the Certus Software Validation and Verification Center. He obtained his Ph.D. degree in computer science from the University of Nice-Sophia Antipolis in 2000. He worked for seven years in industry at Thales and then joined Inria, France, as a research scientist before moving to Simula, Norway. He coauthored more than 80 academic publications, led several research projects in software testing, cochaired several program committees including the SEIP track of ICSE'14 and the testing and verification track of CP'16.

Dusica Marijan Ph.D. is a research scientist in software engineering at Simula Research Laboratory, working on practical strategies for more cost-effective software testing. Her research interests include test automation and optimization, with a goal to help practitioners perform higher quality testing at lower costs and with fewer resources. Prior to joining Simula, she worked as a senior software engineer in the mobile and multimedia software industry.