Proceedings of the Twenty-Eighth AAAI Conference on Innovative Applications (IAAI-16)

Automated Regression Testing Using Constraint Programming*

Arnaud Gotlieb,[1] Mats Carlsson,[2] Marius Liaaen,[3]
Dusica Marijan,[1] Alexandre Pétillon[1]
[1] SIMULA, Norway (simula.no)
[2] SICS, Sweden (sics.se)
[3] CISCO, Norway (cisco.com)

Abstract

In software validation, regression testing aims to check the absence of regression faults in new releases of a software system. Typically, test cases used in regression testing are executed during a limited amount of time and are selected to check a given set of user requirements. When testing large systems, the number of regression tests grows quickly over the years, and yet the available time slot stays limited. In order to overcome this problem, an approach known as test suite reduction (TSR), has been developed in software engineering to select a smallest subset of test cases, so that each requirement remains covered at least once. However solving the TSR problem is difficult as the underlying optimization problem is NP-hard, but it is also crucial for vendors interested in reducing the time to market of new software releases. In this paper, we address regression testing and TSR with Constraint Programming (CP). More specifically, we propose new CP models to solve TSR that exploit global constraints, namely NVALUE and GCC. We reuse a set of preprocessing rules to reduce a priori each instance, and we introduce a structure-aware search heuristic. We evaluated our CP models and proposed improvements against existing approaches, including a simple greedy approach and MINTS, the state-of-the-art tool of the software engineering community. Our experiments show that CP outperforms both the greedy approach and MINTS when it is interfaced with MiniSAT, in terms of percentage of reduction and execution time. When MINTS is interfaced with CPLEX, we show that our CP model performs better only on percentage of reduction. Finally, by working closely with validation engineers from Cisco Systems, Norway, we integrated our CP model into an industrial regression testing process.

Introduction

Context. Software testing is the main validation technique used to check the reliability and robustness of software systems. It includes several phases such as functional testing, performance testing and regression testing, as the goal is to detect distinct faults in the system. By executing the application with a set of existing test cases used to test previous releases, regression testing checks the absence of regression faults, that is, faults that may have been re-introduced into the application during development of new features. Regression testing is a time-consuming activity. In order to keep short the time to market of new releases, a judicious selection of regression test scripts to re-execute has to be performed, ideally without jeopardizing the quality of the overall regression testing process.

An industrial case study. The video-conferencing software system developed by Cisco Systems, Norway, is a system that needs to be thoroughly tested before release. Its recognized quality is seen as a strong advantage in a competitive market where cheaper competitor products (but with lower quality) exist. Performing regression testing of this system involves verifying all its features at least once before release. For instance, testing features such as audio call, video call, multi-site call, is crucial to preserve the quality of the system. As the system has been developed over many years, a database repository of about 5000 test cases is maintained. Each test case includes a manual preparation step to make the system testable (setting up the necessary devices, establishing calls to distant sites, etc.), so that the average time required to execute each test case is about half-an-hour. Note that this time dominates the time required to execute the other more automated parts of the scripts. Since only a couple of days are available for testing a new release, one has to select among the test cases those to be executed in regression testing. Of course, an army of testers could work in parallel to execute all the regression tests, but the testing costs would rapidly become prohibitive. So, there is a challenge in selecting a smallest subset of test cases in order to minimize the overall test execution time, such that every feature is executed at least once. This challenge is known in the software engineering community as the *test suite reduction* (TSR) problem.

Test Suite Reduction. Formally speaking, given a set of test cases T, a set of requirements R and a covering relation Cov saying which requirement is covered by the test cases, TSR aim to find a smallest subset T' of test cases such that each requirement is covered by at least one test case from T'. This problem is related to *Minimum Set Cover* which is known to be NP-complete (Garey and Johnson 1990). Unlike this problem, TSR always considers all the possible subsets of T, meaning that the number of subsets is exponential in the cardinality of T. In the lit-

*This work is supported by the Research Council of Norway (RCN) through the research-based innovation center Certus, under the SFI programme

erature, TSR has been approached with at least three distinct techniques: greedy algorithms (Rothermel et al. 2002; Tallam and Gupta 2005; Jeffrey and Gupta 2005), search-based testing techniques (Ferrer et al. 2015; Wang, Ali, and Gotlieb 2015), and exact methods (Hsu and Orso 2009; Chen, Zhang, and Xu 2008; Gotlieb and Marijan 2014).

Greedy approaches. A classical approximation algorithm used for TSR is reported in (Chvátal 1979). The heuristic selects first the test case that covers the most features and repeats the process until all features are covered. The main limitation of Chvátal's and similar greedy approaches concerns its possible inclusion of redundant test cases. In the 90's, (Harrold, Gupta, and Soffa 1993) proposed a technique which approximates the computation of minimum-cardinality hitting sets. More recently, (Tallam and Gupta 2005) introduced the delayed-greedy technique, which exploits implications among test cases and requirements to further refine the reduced test suite. One shortcoming of greedy algorithms is that they only approximate the global optima without providing optimal test suite reduction. It is problematic in our industrial context as the execution time of each test case (about half-an-hour) is much larger than other industrial cases. Reaching a truly optimum is thus highly desirable, so that the total execution time of the test suite can be strongly shortened. **Search-based testing techniques.** Meta-heuristics have been used to deal with TSR. By comparing 10 distinct algorithms for different criteria in (Wang, Ali, and Gotlieb 2015), the authors observed that random-weighted multi-objective optimization is the most efficient approach. However, this approach assigns weights at random, meaning actually that no priority can be established between the criteria. All these techniques can scale up to problems having a large number of test cases and requirements but they cannot explore the overall search space and thus they cannot guarantee global optimality.

Exact approaches. To the best of our knowledge, exact approaches for the TSR problem are based either on SAT (Boolean satisfiability) or ILP (Integer Linear Programming) solving. The best-known approach for exact test suite minimization is called MINTS (Hsu and Orso 2009). MINTS can possibly be interfaced with either MiniSAT or CPLEX and it can deal with multi-criteria ILP formulation, such as weighted sum, prioritized optimization or hybridization. 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. In the context of feature covering for highly-configurable software systems, an approach based on SAT solving is proposed in (Uzuncaova, Khurshid, and Batory 2010). TSR is encoded as a Boolean formula that is evaluated by a SAT solver. An hybrid method based on ILP and search, called DILP, is proposed in (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. In (Mouthuy, Deville, and Dooms 2007), Monthuy *et al.* 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, we introduced in (Gotlieb and Marijan 2014) an approach for test suite reduction based on the computation of maximum flows in a network flow. This initial idea has triggered the work reported in the present paper, where there is an in-depth analysis of different CP model based on global constraints.

Contributions of the paper. This paper introduces and compares three distinct CP models based on global constraints, namely the NVALUE (Pachet and Roy 1999) and the GCC (Régin 1996) constraints. By reusing existing preprocessing rules to reduce a priori the size of the problem, and by introducing dedicated search heuristics for TSR, we are able to deal with TSR instances which are outside the scope of current approaches. We compare the running time and the reduction percentage of test suites for three CP models on both random instances and industrial instances. We also evaluate individually each proposed improvement (preprocessing, search heuristics). Finally, we report on the adoption of CP to the testing of Cisco's video-conferencing systems. To the best of our knowledge, these CP models and the proposed improvements are original for solving the TSR problem, and thus, the automatic reduction of test suites for software systems can be considered as a new application of CP.

Organization of the paper. Section 2 formally defines TSR and gives some background on CP. Section 3 contains the three CP models and introduces some improvements. Section 4 presents our experimental evaluation to compare the CP models with other approaches. It also contains some elements on Cisco's adoption of CP. Finally, section 5 concludes the paper.

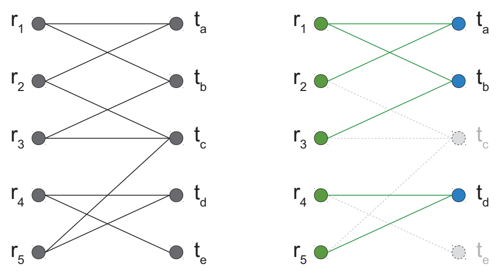
Background

Definition 1 (Test Suite Reduction (TSR)). *A* TSR *instance is a quadruple* (T, R, Cov, f) *where* T *is a set of test cases* $\{t_1, ..., t_m\}$*,* R *is a set of requirements* $\{r_1, ..., r_n\}$*,* Cov *is a relation over* R *and* T*, and* f *is a function* $f : 2^T \mapsto \mathbb{Z}$*. An optimal solution to TSR is a subset* $T' \subseteq T$ *such that for each* $r \in R$*, there exists* $t \in T'$ *such that* $(r, t) \in Cov$ *and has minimal cost* $f(T')$*.*

Any TSR problem can be encoded by a bipartite graph with edges representing the relation Cov as shown in Fig.1. A *domain variable* V is a logical variable with an associated finite domain $D(V) \subset \mathbb{Z}$. Our CP models use the following global constraints:

Definition 2 (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*.*

Definition 3 (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,* GCC(T, d, C) *holds iff for each* $i \in 1..m$ *the number of occurrences of* d_i *in* T *is* C_i*. The* C_i *variables are the* occurrence variables *of the constraint.*



(a) (b)

Figure 1: TSR as a bipartite graph and an optimal solution. Let $T = \{t_a, t_b, t_c, t_d, t_e\}$, $R = \{r_1, r_2, r_3, r_4, r_5\}$, Cov given in (a), and f be $f = card(T')$, then $\{t_a, t_b, t_d\}$ is an optimal solution as shown in (b)

CP models of the TSR problem

Our CP models are based on the following encoding: each requirement r_i is represented by a finite domain variable R_i with domain $\{t_1, ..., t_n\}$, where each t_i represents a test case covering R_i.

A Naive Model (NVALUE)

$$\text{Minimize } N \text{ s.t.} \textsc{NValue}(N, (R_1, \ldots, R_n))$$

This model aims to minimize the number of different values that can be taken by $R_1, \ldots, R_n$, that is, the number of distinct test cases covering all the requirements. This is a naive model because it does not include ways to perform the search of optimal solutions through the selection of the most promising test cases. For example, branching on the selection of test cases that cover the most requirements is highly desirable. So, we introduced another model based on GCC.

A Model with GCC(GCC^2)

Let the domain variables O_i be the number of times test case t_i is selected to cover requirements $R_1, ..., R_n$, then:

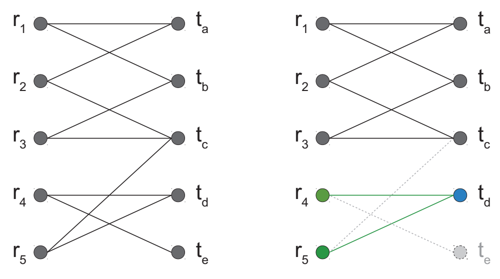
$$\begin{array}{l}\text{Maximize } N \text{ s.t.}\\ \text{GCC}((R_1, \ldots, R_n), (t_1, \ldots, t_m), (O_1, \ldots, O_m)) \wedge\\ \text{GCC}((O_1, \ldots, O_m), (0), (N))\end{array}$$

is another model for TSR. The second GCC allows us to constrain the selection of test cases, maximizing the number of unselected test cases, This model offer us the opportunity to branch on the number of occurrences of each test case. Since the filtering algorithm of GCC is somewhat costly (Régin 1996), we proposed a third model, which implements a number of optimizations and combines the advantages of the first two models.

An Optimized Model (*Mixt*)

The following model keeps the advantage of permitting us to branch on the O_i.

$$\begin{array}{l}\text{Minimize } N \text{ s.t.}\\ \textsc{NValue}(N, (R_1, \ldots, R_n)) \wedge\\ \text{GCC}((R_1, \ldots, R_n), (t_1, \ldots, t_m), (O_1, \ldots, O_m))\end{array}$$



(a) (b)

Figure 2: Preprocessing. The edge (r_4, t_e) is removed from Cov by r1, since $\{r \mid (r, t_e) \in Cov\} = \{r_4\}$ is included in $\{r \mid (r, t_d) \in Cov\} = \{r_4, r_5\}$. The edges (r_5, t_c) and (r_5, t_d) are removed from Cov by r2, since $\{t \mid (r_4, t) \in Cov\} = \{t_d\}$ is included in $\{t \mid (r_5, t) \in Cov\} = \{t_c, t_d\}$. t_d must be included in the solution set by r3, since $\{t \mid (r_4, t) \in Cov\} = \{t_d\}$.

In addition, we propose a number of optimizations including preprocessing and specialized search heuristics.

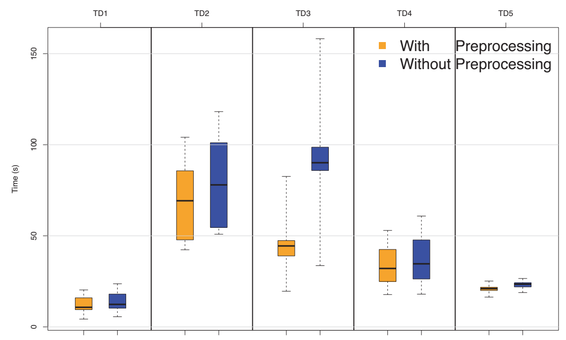
Preprocessing allows us to reduce the size of the problem beforehand, by using the following rules:

r 1. If t_1, t_2 exist s.t. $\{r \mid (r, t_1) \in Cov\} \subseteq \{r \mid (r, t_2) \in Cov\}$, then remove all tuples (r, t_1) from Cov.

r 2. If r_1, r_2 exist s.t. $\{t \mid (r_1, t) \in Cov\} \subseteq \{t \mid (r_2, t) \in Cov\}$, then remove all tuples (r_2, t) from Cov.

r 3. If r_1, t_1 exist s.t. $\{t \mid (r_1, t) \in Cov\} = \{t_1\}$, then t_1 must be included in the solution set.

Fig 2 illustrates these preprocessing rules.



	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 3: Comparison of CPU time w.r.t. preprocessing

We performed experiments to evaluate the benefits of these preprocessing rules for both randomly generated TSR problems and industrial instances, as shown in Fig. 3 and Fig. 11.

Search Heuristics

Search heuristics consist of a variable selection strategy and a value assignment strategy, which both relate to the vari-

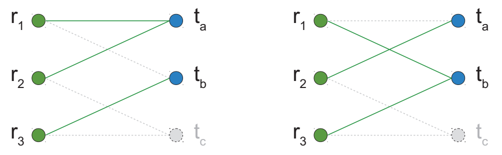


Figure 4: Two symmetrical solutions for CP, a single solution of TSR. In both graphs, the same optimal test suite is obtained, $T' = \{t_a, t_b\}$. However, it is associated with distinct solutions for CP because the R_i are assigned to distinct values: on the left, R_1 is assigned to t_a while on the right R_1 is assigned to t_b. With our dedicated heuristic, an arbitrary selection is made, e.g., 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

ables used in the model. For the NVALUEmodel, we use the first-fail principle, which selects the domain 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 those test cases are most likely to be selected. The default value selection strategy is based on a statically ordered list of test cases.

As said above, the naive model does not permit us to branch on other variables than the requirement variables. By introducing occurrence variables in the GCC^2 and *Mixt* models, it is possible to branch on these variables. For those models, we select the occurrence variable V that maximizes $\max(D(V))$. The underlying idea is to explore first the search tree where test cases cover the most requirements. Unlike static strategies used in classical approximation algorithms, our strategy is dynamic and the ordering is revised at each step of the selection process. The following non-trivial observation formalizes this idea:

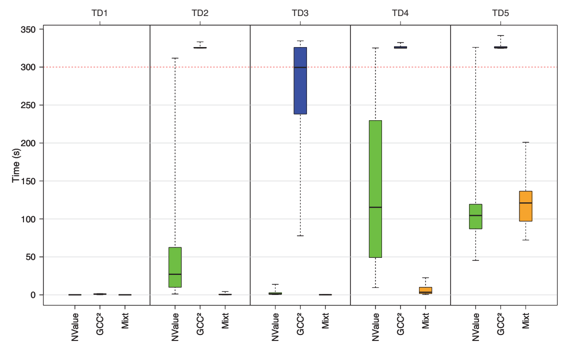
Property 1. *Let each test case t_i be represented by an occurrence variable O_i taking its values in $0..max_i$ where max_i is dynamically updated with the current partial assignment. Then, for each solution X of the TSR problem with cost $f(X)$ where $O_i = n_i$ such that $0 < n_i < max_i$ (strict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

The TSR-dedicated heuristic is incomplete, meaning that some parts of the search tree might stay unexplored. However, based on Prop. 1, the TSR-dedicated heuristic guarantees that at least one optimal solution is found, as shown in Fig.4.

Experimental Evaluation

All our experiments were run on a standard i7-2929XM CPU machine at 2.5GHz with 16GB RAM. We performed experiments on both random and industrial instances of TSR provided by Cisco. We built a random generator of TSR instances, which takes three parameters as inputs: the number of requirements, the number of test cases and, d, the density of the relation Cov. By density, we mean the greatest arity value for all requirement in Cov. Our generator draws at random a number a between 1 and d and creates a random edges in the bipartite graph of Cov.

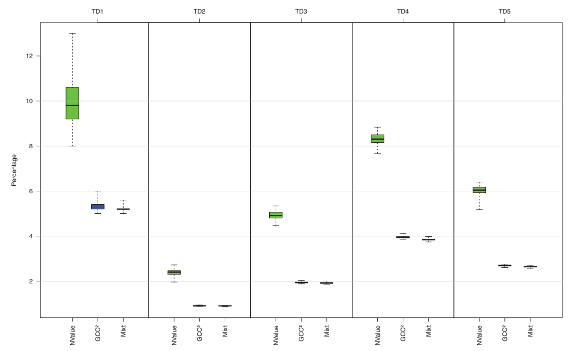
Results and Analysis



	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 5: Comparison of CPU time for the CP models (time-out = 300s)

Comparison of the various CP models Fig. 5 compares the CPU time of our CP models. In each data set, 20 random samples were generated. For all but TD1, the GCC^2 model times out (after 300 sec). 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 Fig. 6, we compute the percentage

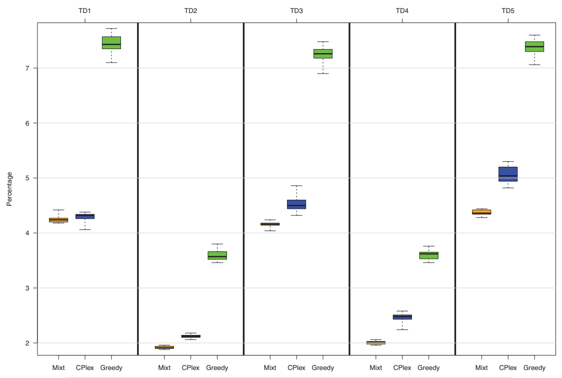


	TD1	TD2	TD3	TD4	TD5
Requirements	250	500	1000	1000	1000
Test cases	500	5000	5000	5000	7000
Density	20	20	20	8	8

Figure 6: Comparison of reduction rate for the CP models (in percentage of remaining test cases, time-out = 30s)

of test cases remaining in the solution set after 30 seconds. A good reduction rate is crucial for any industrial adoption, as test suite reduction has to be performed within a continuous integration process, where this 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. We interpret this to be due to the selection of the branching heuristic, which is different for the NValue model, where only the requirement variables are available for branching.



	TD1	TD2	TD3	TD4	TD5
Requirements	1000	1000	2000	2000	4000
Test cases	5000	5000	5000	5000	5000
Density	7	20	20	50	50

Figure 7: Comparison of reduction rate of Mixt, CPLEX and Greedy (in percentage of test cases, time-out = 60s)

Comparison with other approaches Fig. 7 shows the results of experiments performed on 4 approaches, namely, our CP Mixt model, MINTS (Hsu and Orso 2009) interfaced with CPLEX, MINTS interfaced with MiniSAT+, and our own implementation of Chvátal's greedy algorithm. In all the cases, a time-out of 60 seconds is set up and the smallest solution set is returned. In Fig.8, we evaluated the importance of CPLEX's own preprocessing in reaching an optimal solution by observing the size of the solution sets at different time-instants. From these charts, we observe that MINTS/MiniSAT+ and Chvátal's greedy algorithm reach more or less the same reduction rate, while the CP Mixt and MINTS/CPLEX approaches reach significantly better reduction rates. With these test data sets, Mixt outperforms

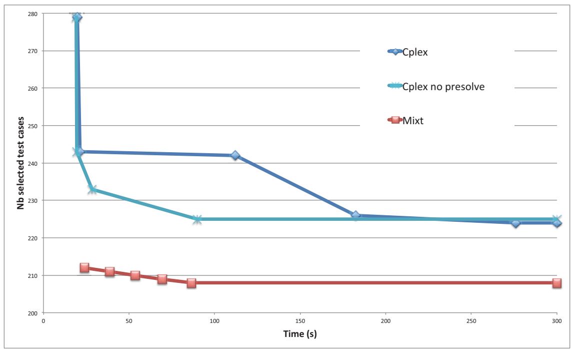


Figure 8: Evaluation of CPLEX preprocessing w.r.t. Mixt

MINTS/CPLEX, but this is something that cannot be generalized without paying attention to the level of density of the generated problems. Note also that both the preprocessing rules and the TSR-dedicated search heuristic we came up with play an important role in these results in favor of CP. The preprocessing of Mixt cannot be compared with CPLEX's own preprocessing as they both work on different data structures.

Comparison of several search heuristics Fig. 9 shows the CPU time for three variable-selection heuristics (i.e., `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 the better result by selecting the occurrence variable that has the greatest arity, i.e., the one associated with a test case that covers the most requirements. We selected it to be employed with our CP Mixt model.

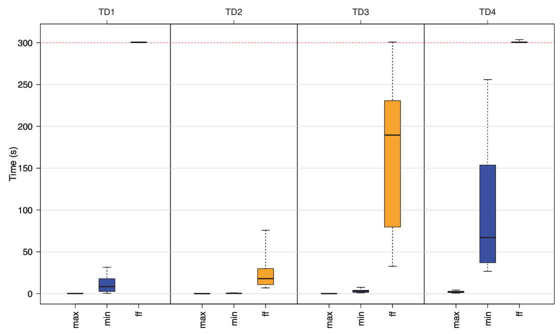


Figure 9: CPU time of several variable-selection heuristics.

Fig. 10 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 $\{0, 1, \ldots, Max\}$.

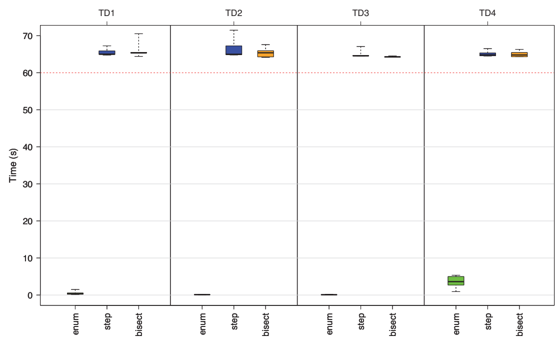


Figure 10: CPU time of several value-selection heuristics.

As expected, Fig. 10 shows much better results for our

Table 1: Comparison of CPU time between CP models

	ID1	ID2	ID3	ID4	ID5	ID6
Requirements	59	53	50	37	37	156
Test cases	107	90	93	100	100	377
Mixt	0.0s	0.0s	0.0s	0.0s	0.0s	0.0s
NValue	0.0s	0.1s	0.0s	0.0s	0.0s	0.1s
GCC^2	> 300s	102.0s	91.8s	59.2s	6.1s	> 300s

heuristic. However, it is worth keeping in 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.

Cisco Case Study. The industrial adoption of our CP model for the TSR problem was not easy. One of the issues was to explain to Cisco's validation engineers abstract concepts such as constraint propagation, variable-selection heuristics, global constraints and other AI-related notions such as constraint satisfaction. This is due to the strong expectations of validation engineers to control all parts of the software validation process. In order to facilitate the adoption of constraint satisfaction as part of the validation process, we casted our CP model into a tool, which includes many other features, such as variability management (for the Cisco's video-conferencing systems), test-suite prioritization, and priority-based selection of test cases. We also performed an extensive experimental study on Cisco's data.

The results of Tab.1 show that the Mixt CPmodel can solve industrial instances in no time, while the GCC^2 model is clearly discarded. These results convinced Cisco to integrate the Mixt model into their continuous integration process. Fig. 11 shows the percentage of test cases and requirements that are removed by the preprocessing rules. These results indicate that such preprocessing is crucial in any industrial usage.

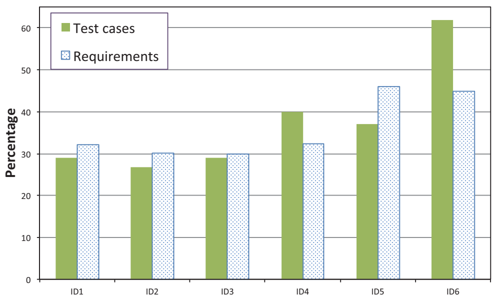


Figure 11: Percentage of test cases and requirements removed by preprocessing on Cisco case study.

Conclusion

This paper presented the results of the application of CP to the test suite reduction problem, which is crucial for regression testing. Three CP models using the global constraints NVALUE and GCCwere compared through extensive experiments on both random and industrial instances. According to our knowledge, this is the first time that CP is applied to the reduction of test suites in software testing. The performance of our best CP model was compared to the state-of-the-art tool MINTS, interfaced with MiniSAT+ and CPLEX. Our results show that CP is not only efficient, but that it outperforms MiniSAT+ and it is competitive with CPLEX in terms of percentage of reduction.

References

Chen, Z.; Zhang, X.; and Xu, B. 2008. A degraded ILP approach for test suite reduction. In *20th Int. Conf. on Soft. Eng. and Know. Eng.*

Chvátal, V. 1979. A greedy heuristic for the set-covering problem. *Math. of Operations Research* 4(3).

Ferrer, J.; Kruse, P. M.; Chicano, F.; and Alba, E. 2015. Search based algorithms for test sequence generation in functional testing. *Information and Software Technology* 58(0):419 – 432.

Garey, M. R., and Johnson, D. S. 1990. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. NY: W. H. Freeman & Co.

Gotlieb, A., and Marijan, D. 2014. Flower: Optimal test suite reduction as a network maximum flow. In *Proc. of Int. Symp. on Soft. Testing and Analysis (ISSTA'14)*.

Harrold, M. J.; Gupta, R.; and Soffa, M. L. 1993. A methodology for controlling the size of a test suite. *ACM TOSEM* 2(3):270–285.

Hsu, H.-Y., and Orso, A. 2009. MINTS: A general framework and tool for supporting test-suite minimization. In *31st Int. Conf. on Soft. Eng. (ICSE'09)*, 419–429.

Jeffrey, D., and Gupta, N. 2005. Test suite reduction with selective redundancy. In *21st Int. Conf. on Soft. Maintenance*, 549–558.

Mouthuy, S.; Deville, Y.; and Dooms, G. 2007. Global constraint for the set covering problem. In *Journées Francophones de Programmation par Contraintes*, 183–192.

Pachet, F., and Roy, P. 1999. Automatic generation of music programs. In *Principles and Practice of Constraint Prog.*, volume 1713 of *LNCS*.

Régin, J.-C. 1996. Generalized arc consistency for global cardinality constraint. In *13th Int. Conf. on Artificial Intelligence (AAAI'96)*, 209–215.

Rothermel, G.; Harrold, M. J.; Ronne, J.; and Hong, C. 2002. Empirical studies of test-suite reduction. *Soft. Testing, Verif. and Reliability* 12:219–249.

Tallam, S., and Gupta, N. 2005. A concept analysis inspired greedy algorithm for test suite minimization. In *6th Workshop on Program Analysis for Software Tools and Eng. (PASTE'05)*, 35–42.

Uzuncaova, E.; Khurshid, S.; and Batory, D. 2010. Incremental test generation for software product lines. *IEEE Trans. on Soft. Eng.* 36(3):309–322.

Wang, S.; Ali, S.; and Gotlieb, A. 2015. Cost-effective test suite minimization in product lines using search techniques. *Journal of Systems and Software* 103:370–391.