Using Knapsack Problem Model to Design a Resource Aware Test Architecture for Adaptable and Distributed Systems

Mariam Lahami, Moez Krichen, Mariam Bouchakwa, and Mohamed Jmaiel

Research Unit of Development and Control of Distributed Applications
National School of Engineering of Sfax, University of Sfax
Sokra Road km 4, PB 1173 Sfax, Tunisia
{mariam.lahami,moez.krichen,mariam.bouchakwa}@redcad.org,
mohamed.jmaiel@enis.rnu.tn
http://www.redcad.org

Abstract. This work focuses on testing the consistency of distributed and adaptable systems. In this context, *Runtime Testing* which is carried out on the final execution environment is emerging as a new solution for quality assurance and validation of these systems. This activity can be costly and resource consuming especially when execution environment is shared between the software system and the test system. To overcome this challenging problem, we propose a new approach to design a resource aware test architecture. We consider the best usage of available resources (such as CPU load, memory, battery level, etc.) in the execution nodes while assigning the test components to them. Hence, this work describes basically a method for test component placement in the execution environment based on an existing model called *Multiple Multidimensional Knapsack Problem*. A tool based on the constraint programming Choco library has been also implemented.

1 Introduction

Adaptable and distributed systems are characterized by the possibility of dynamically changing their behaviors or structures at runtime in order to preserve their usefulness and achieve new requirements. They evolve continuously by integrating new components, deleting faulty or unneeded ones and substituting old components by new versions without service interruption.

In order to preserve the system safety and consistency and to check functional as well as non-functional requirements during and after dynamic adaptation, a validation technique, such as *Runtime Testing*, has to be applied. It is defined in [1] as any testing method that has to be carried out in the final execution environment of a system while it is performing its normal work.

In a previous work [2], we have proposed a flexible and evolvable distributed test architecture made of two kinds of test components. These test components execute unit tests (respectively integration tests) on the affected components (respectively component compositions) with the aim of detecting reconfiguration faults. To do

B. Nielsen and C. Weise (Eds.): ICTSS 2012, LNCS 7641, pp. 103–118, 2012.
© IFIP International Federation for Information Processing 2012

this, they send stimuli to the System Under Test (SUT) in order to verify that it responds as expected. The main challenging problem here is that the test execution is done while the SUT is running. Also, test components are usually deployed and executed in the same execution environment with the SUT. Consequently, SUT performance can be highly influenced especially when execution nodes have scarce resources or testing processes are very resource consuming.

Therefore, placing efficiently test components can be a useful solution to reduce runtime testing cost. This activity has to be resource aware by respecting all resource constraints in order not to disturb the SUT performance and not burden the execution nodes.

Various research efforts have addressed the resource aware testing activity such as [3,4]. They focus mainly on optimizing the generation of test cases with the best usage of available resources. To our best knowledge, there is only two visible research works [5,6] being carried out the issue of test component placement with respecting only some resource constraints and some user preferences. Connectivity constraints which are stated with the aim of reducing communications cost over the network between components under test and testers are ignored in these approaches.

In this paper, we propose a resource aware test architecture design phase before executing runtime tests while the system evolves dynamically. Essentially, we have studied the issue of test component placement in a shared execution environment with the SUT. Two main kinds of constraints are considered : resource and connectivity constraints. Hence, the most important question to be tackled in this paper is: How to place test components in the adequate nodes in order to fit these constraints? To solve this problem, we have modeled it using an existing model in the combinatorial optimization area, called *Multiple Multidimensional Knapsack Problem*. In addition, an implementation of a tool based on the Choco solver has been presented. Also, some experiments have been done to evaluate the proposed approach.

The remainder of this paper is organized as follows. We begin by a motivating example in Section 2. A brief description of related work is addressed in section 3. Section 4 overviews the resource aware test architecture design phase and outlines particularly the test component placement over the execution nodes fitting resource and connectivity constraints. In Section 5, we present our test component placement method based on the Knapsack Problem (KP) model. First, we introduce concisely the background of the standard KP and its diverse forms. Next, we illustrate the mathematical modeling of the placement problem using a new KP variant called *Multiple Multidimensional Knapsack Problem.* The realization of this approach is provided in Section 6. Some experiments for execution time measuring of the placement method are conducted in section 7. Finally, Section 8 concludes the paper and draws some future work.

2 Motivating Example

Consider an execution environment made of four nodes ($N1$,$N2$,$N3$ and $N4$) which are offering some free resources as illustrated in table 1. Some software

components ($C1$,$C2$,$C3$,$C4$ and $C5$) are running and distributed among these nodes. We assume that a dynamic reconfiguration occurs and all these components are affected by this modification. Thus, they have to be tested in order to ensure that they still behave as intended.

Table 1. Execution node characteristics

Nodes	Free RAM	CPU Load	Components
$N1$	50 %	20 %	$C1$, $C2$
$N2$	40 %	30 %	$C3$
$N3$	35 %	50%	$C4$
$N4$	60 %	60 %	$C5$

Table 2. Test component characteristics

Testers	Required RAM	CPU usage	CUTs
$T1$	10 %	10 %	$C1$
$T2$	10 %	5 %	$C2$
$T3$	15 %	10 %	$C4$
$T4$	20 %	15 %	$C5$

For this reason, some test components have to be deployed in this shared execution environment. These test components require some computational resources as depicted in table 2. With the aim of not disturbing the running SUT and reducing test burdens, they have to be assigned to the execution nodes efficiently while fitting some resource constraints. In this example, we consider just two kinds of computational resources: RAM and CPU. It is worthy to note that others resources like battery level and hard disk space can be included. Furthermore, we assume for simplicity that memories and processors in execution nodes have approximately the same capacities.

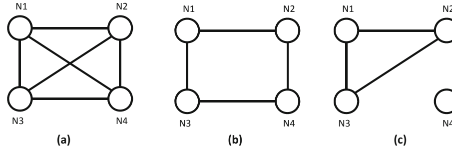

Fig. 1. Execution environment modeling using graphs

Besides resource constraints, connectivity issue in the execution environment has to be also tackled. If all nodes are connected together as a strongly connex graph or even a connex graph (see Figure 1 (a), (b)), test components can be assigned to any node in the environment in order to perform runtime tests. Nevertheless, we are faced sometimes with some connectivity problems[1] that can be arisen due to Firewalls, non-routing networks [7], node mobility, etc. In this case, connectivity constraints have to be considered while assigning test components to nodes. For example, while non routing is performed between the node $N4$ and the rest of the execution environment as depicted in Figure 1 (c) the test component $T4$, which is responsible of testing the component under test (CUT) $C5$, can be hosted only in the node $N4$. It is not allowed to place $T4$ in $N1$, $N2$ or even $N3$ because no route is available to communicate

[1] For instance, wireless communication networks are characterized by frequent and unpredictable changes in connectivity.

with the component under test $C5$. In the same way, the other test components can be assigned to any node except $N4$. This illustrative example is detailed progressively in the following sections of the paper. Placement solutions are also given in section 5.

3 Related Work

Since runtime testing is performed while the system is operational, the business process execution may be affected by testing activities and the system performance may be negatively influenced. Therefore, test isolation techniques (such as SUT duplication, tagging, SUT blocking, etc.) are required to separate testing processes from business processes [8,9,10]. In addition, testing activities have to be resource aware with the aim of minimizing execution nodes burdens while runtime tests are executed. The first issue is out of the scope of this paper. We mainly concentrate on studying the resource aware testing activity [3,4].

Fitting resource and connectivity constraints while testing distributed and adaptable systems at runtime is included in a larger class called *context aware testing* [11,12]. The latter has recently emerged to validate especially new class of software systems which are context aware and adaptive, also known as *Ubiquitous* or *Pervasive* systems [4]. This kind of systems can sense their surrounding environment and adapt their behavior accordingly. In [13], the author generally defines the context as any information that can be used to characterize the situation of an entity (which can be a person, a machine or any object including a service, a software component, or data). He divides context into two main categories: external context (which contains information about the users, their location, time, etc.) and resource context (which describes the available resources on nodes and communication links like memory, CPU load, battery level, bandwidth between two nodes, etc.). Various research efforts have addressed testing activity with considering resource context such as [3,4]. They focus mainly on optimizing the generation of test cases with the best usage of available resources but without studying the placement and the deployment cost of test components.

To our best knowledge, there is only two visible research works related to test component placement under resource constraints and user preferences. In the first work [5], the authors propose a function for distributing a set of test components, which are belonging to a test configuration implemented in TTCN-3 standard[2][14], on different test nodes (computers that are dedicated for test execution). The proposed mathematical function is applied at deployment time separately for each test component in order to assign it to a node where it will be deployed and also executed. It considers two types of parameters when distributing test components in the adequate test nodes: external parameters such as CPU load, memory consumption and internal parameters like the number of components that can be hosted in a specific test node. The second work presented in [6] mainly focuses on the dynamic deployment of test components also

[2] Testing and Test Control Notation version 3 (TTCN-3) standard offers a standardized test notation and an execution platform facilitating the deployment and the execution of test components, `http://www.ttcn-3.org/`

implemented in TTCN-3 standard. It proposes an approach for designing load tests and distributing test components efficiently with considering the available workstation resources. The major problem here is that these approaches do not define explicitly the proposed distributed function used for test components assignment to test nodes. Moreover, they focus mainly on computational resources and they ignore connectivity constraints.

Unlike these approaches, our work aims at defining a novel method for assigning test components to execution nodes in a way fitting both resource and connectivity constraints. This challenging issue has been widely addressed in other research areas. First, an interesting work is presented in [15] that aims to optimize resource allocation and the placement of Java components by using a graph mapping approach. The latter consists in modeling the application by a software graph and the execution environment by a hardware graph. The main purpose here is to map as best as possible the software graph on the hardware graph. Other approaches have studied the placement issue based on constraint programming, which aims to model and solve combinatory problems, such as [16,17]. By extending the *Multiple Knapsack Problem* model, [16] proposes a method for assigning sensors in virtual environments. Hermenier [17] presents a flexible architecture that adapts the placement of virtual machines in grids with response to requirements analysis, resources states and some placement constraints defined by the user. The problem here is similar to *Two-Dimensional Bin Packing* problem. The classical problem consists in packing objects with different volumes into a finite number of bins having a predefined capacity in a way that minimizes the number of bins used. Following the same principle, Hermenier's work aims to minimize the number of nodes in the grid involved in the placement of virtual machines while satisfying resource constraints such as CPU load and memory consumption. Both introduced approaches use the Choco solver [18] in order to solve the placement problem. In the rest of this paper, our proposal that is inspired from the constraint programming based approaches will be highlighted.

4 Resource Aware Test Architecture Design

After a dynamic evolution of the system, runtime testing process is started to validate these changes. Only the affected parts of the system are considered in the testing activity. In Figure 2, two fundamental steps are outlined: *Test Architecture Design* and *Test Component Placement*. They will be detailed in the following subsections.

4.1 Test Architecture Design

This activity consists in defining for each affected component or composition the kind of test component to deploy and the test cases to execute. As illustrated in Figure 3, the elements involved in a distributed test architecture (DTA), their kind and their number depend on the affected parts of the system by a reconfiguration action. In our previous work [2], we defined a *Single Component Tester* (SCT) which is in charge of executing unit tests once a single component

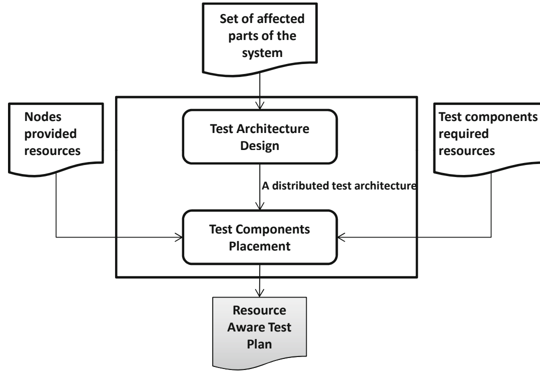

Fig. 2. Resource Aware Test Architecture Design

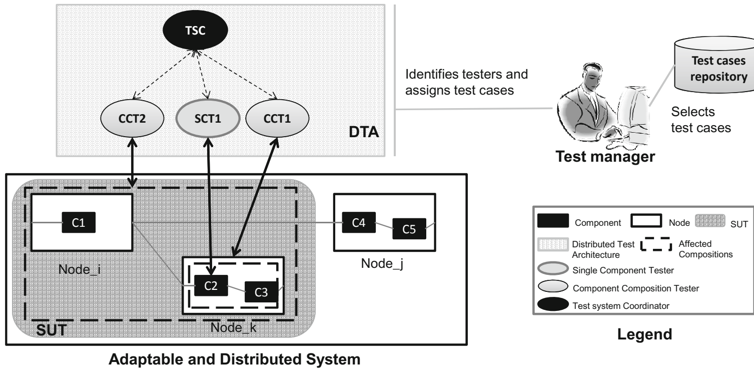

Fig. 3. Overview of a detailed Distributed Test Architecture

has been changed or newly added at runtime. Moreover, *Component Composition Tester* (CCT) is introduced to validate the affected component compositions. These two kinds of testers communicate with a *Test System Coordinator* (TSC) which is charged with generating a global verdict depending on local verdicts of SCTs and CCTs.

In this work, we suppose that for each adaptation process a test manager which is responsible for controlling and managing all the runtime testing processes is introduced. It defines the adequate test architecture and assigns test cases. We also assume that during deployment phase, test cases are available and stored in a repository for further use. They can be also updated or new ones can be added if behavioral adaptations occur.

4.2 Test Component Placement

Once the distributed test architecture is elaborated, we have to assign its constituents to the execution nodes. It is worth noting that in this work we focus on

assigning mainly single components testers to the execution nodes. The placement issue of component composition testers is out the scope of this paper.

Test component placement is more challenging when tests are executed at runtime. In fact, test components may share the same execution environment with the running SUT. This may burden some execution nodes of the SUT and may have a bad impact on the SUT performance. Also, it may sometimes cause malfunctions. To resolve this problem, we concentrate in this work on proposing a new method for adapting the test components deployment at runtime to the resource situation of the execution nodes and also to connectivity constraints.

– **Consideration of Resource Allocation Issue**

We first introduce the considered resources for nodes as well as for test components. For each node in the execution environment, three resources are monitored during SUT execution: the available memory, the provided CPU and the battery level. The value of each resource can be directly captured on each node through the use of internal monitors. These values are measured after the runtime reconfiguration and before starting the testing activity.

For each test component, we introduce the memory size (the memory occupation needed by a test component in execution), CPU load and battery consumption properties. We suppose that these properties values are provided by the test manager. It is also worth noting that some techniques are available in the literature for obtaining the required resources by testers. For example in [5], the authors propose a preliminary test to learn about some required resources such as the amount of memory allocated by a test component, the time needed to execute the test behavior, etc.

– **Consideration of Connectivity Issue**

Regarding the connectivity constraints, we consider that each test component has to find at least one route to communicate with the component under test. As mentioned before, this constraint can be ignored when all nodes are communicating together. In this case, the execution environment is modeled as a connected graph. Recall that in the graph theory, a graph is connected if for every pair of vertices, there is a path in the graph between those vertices. Hence, each test component can be assigned to any node and it can communicate with the node under test either locally or remotely.

In the worst case, whereas some connectivity problems occur [7], the execution environment is considered not connected. In this situation, the graph is obviously decomposed in several *connected components* as we have seen in the illustrative example (see Figure 1 (c)). Therefore, for each test component we have to pinpoint a set of forbidden nodes to avoid when the placement procedure is done. For instance, the set of forbidden nodes for the test component $T1$ contains $N4$. From a technical perspective, either depth-first[3] or breadth-first[4] algorithm can

[3] `http://en.wikipedia.org/wiki/Depth-first_search`
[4] `http://en.wikipedia.org/wiki/Breadth-first_search`

be used to firstly identify the connected components and secondly to compute the forbidden nodes for each test component involved in the test architecture.

We can also associate for each node a profit. While the tester is placed in the same node with the component under test, the profit is maximal because the communications cost over the network will be reduced. It decreases once the tester is placed far from the component under test.

In the rest of this paper, we suppose that the execution environment has been modeled as a connex graph. Even when the obtained graph is disconnected, we have to compute the connected components and apply the same adopted method for placement to these sub-graphs. In the next section, we formalize the placement problem using a variant of the Knapsack Problem under assumptions like: provided resources for each node are accessible and required resources for each test component are available too.

5 Mathematical Modeling of the Test Component Placement

5.1 Background

The *Knapsack Problem* (KP) is a well-studied, $\mathcal{NP}$-hard combinatorial optimization problem. It has been used to model different applications for instance in computer science and financial management. It considers a set of n objects $O = o_1, \ldots, o_n$ and a knapsack of capacity W. Each object o_j has an associated profit p_j and weight w_j. The objective is to find a subset $S \subseteq O$ in such a way that the weight sum over the objects in S does not exceed the knapsack capacity and yields a maximum profit [19,20,21].

The most basic form of *Knapsack Problem* (KP) is formulated as follows:

$$KP = \begin{cases} maximize \quad z = \sum_{j=1}^{n} p_j x_j \\ subject \quad to \quad \sum_{j=1}^{n} w_j x_j \leq W \\ x_j \in \{0,1\} \quad \forall j \in \{1, \cdots, n\} \end{cases}$$

In the literature, we found many variants of this problem. Due to space limitations, we describe in details only the two models used in our context:

The Multidimensional Knapsack Problem (MDKP). is also called Multiply constrained Knapsack Problem or m-dimensional knapsack problem. It can be viewed as a resource allocation model and can be modeled as follows:

$$MDKP = \begin{cases} maximize \quad z = \sum_{j=1}^{n} p_j x_j \\ subject \quad to \quad \sum_{j=1}^{n} w_{ij} x_j \leq c_i \quad \forall i \in \{1, \cdots, m\} \\ x_j \in \{0,1\} \quad \forall j \in \{1, \cdots, n\} \end{cases}$$

Where a set of n items with profits $p_j > 0$ and m resources with capacities $c_i > 0$ are given. Each item j consumes an amount $w_{ij} \geq 0$ from each resource i. The 0-1 decision variables x_j indicate which items are selected. The main purpose is to choose a subset of items with maximum total profit. Selected items must not exceed resource capacities. This is expressed by the knapsack constraints [21]. Obviously, the KP is a special case of the multidimensional knapsack problem with $m = 1$.

The 0-1 Multiple Knapsack Problem (0-1 MKP). is the problem of assigning a subset of n items to m distinct knapsacks having different capacities [22,23]. It is also referenced as the 0-1 integer programming problem or the 0-1 linear programming problem. More formally, a MKP is stated as follows:

$$MKP = \begin{cases} maximize \quad z = \sum_{i=1}^{m} \sum_{j=1}^{n} p_j x_{ij} \\ subject \quad to \quad \sum_{j=1}^{n} w_j x_{ij} \leq W_i \quad \forall i \in \{1, \cdots, m\} \\ \sum_{i=1}^{m} x_{ij} \leq 1 \quad \forall j \in \{1, \cdots, n\} \\ x_{ij} \in \{0, 1\} \quad \forall j \in \{1, \cdots, n\} \quad and \quad \forall i \in \{1, \cdots, m\} \end{cases}$$

5.2 Our Mathematical Modeling

Mathematically, our placement problem can be modeled by merging the two introduced knapsack variants: multidimensional and multiple knapsack problems. The obtained model is called *Multiple Multidimensional Knapsack Problem* (MMKP). It is worthy to note that this new variant of the standard KP has been rarely addressed in the literature except in [24]. We assume that the execution environment consists of m nodes and we have n test components that may be assigned to them. We attempt to find an optimal solution of test component placement not violating resource and connectivity constraints and also maximizing their placement profit. We can formulate this problem using the MMKP variant as follows:

$$MMKP = \begin{cases} maximize \quad \mathcal{Z} = \sum_{i=1}^{n} \sum_{j=1}^{m} p_{ij} x_{ij} \quad (1) \\ subject \quad to \quad \sum_{i=1}^{n} x_{ij} dc_i \leq c_j \quad \forall j \in \{1, \cdots, m\} \quad (2) \\ \sum_{i=1}^{n} x_{ij} dr_i \leq r_j \quad \forall j \in \{1, \cdots, m\} \quad (3) \\ \sum_{i=1}^{n} x_{ij} db_i \leq b_j \quad \forall j \in \{1, \cdots, m\} \quad (4) \\ \sum_{j=1}^{m} x_{ij} = 1 \quad \forall i \in \{1, \cdots, n\} \quad (5) \\ x_{ij} \in \{0, 1\} \quad \forall i \in \{1, \cdots, n\} \quad and \quad \forall j \in \{1, \cdots, m\} \end{cases}$$

The provided resources by the m nodes are given through three vectors: C that contains the provided CPU, R that provides the available RAM and B that contains the battery level of each node.

$$C = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{pmatrix} \quad R = \begin{pmatrix} r_1 \\ r_2 \\ \vdots \\ r_m \end{pmatrix} \quad B = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

In addition, the required resources for each test component are illustrated over three vectors: D_c that carries the required CPU, D_r that contains the required RAM and D_b that contains the required Battery by each tester.

$$D_c = \begin{pmatrix} dc_1 \\ dc_2 \\ \vdots \\ dc_n \end{pmatrix} \quad D_r = \begin{pmatrix} dr_1 \\ dr_2 \\ \vdots \\ dr_n \end{pmatrix} \quad D_b = \begin{pmatrix} db_1 \\ db_2 \\ \vdots \\ db_n \end{pmatrix}$$

Similarly, we define the two dimensional variable, x_{ij} as follows:

$$x_{ij} = \begin{cases} 1 & \text{if} \quad \textit{tester } \text{i} \textit{ is assigned to node } \text{j} \\ 0 & \text{otherwise} \end{cases}$$

We may find a feasible solution of test component placement if the objective function (1) is omitted. Otherwise, an optimal solution is computed that maximizes the placement profit. For doing this, a matrix $\mathcal{P}$ has been introduced which is filled with a profit value of each test component in response to execution nodes. This matrix depends on the length path between the node under test[5] and the node hosting the test component. The profit p_{ij} can be equal to a predefined value $maxP$ if the test component i is assigned to a node j which corresponds to the node under test. It can be equal to $maxP - l$ if the test component i is assigned to a node j reachable from the node under test via a path having the length l. The constraints (2),(3) and (4) ensure that the overall required resources by the testers can not exceed the available resources in each node. They are called knapsack constraints similar to the standard knapsack problem. The equality (5) indicates that all testers have to be assigned to the execution nodes and each of them has to be placed in at most one node.

It is worthy to note that in this work we have chosen the distance between nodes as criteria for filling the matrix $\mathcal{P}$. However, other placement criteria can be used such as bandwidth utilization. Furthermore, this solution is dedicated for a connected network. Even if some connectivity problems occur, we apply the proposed model for each connected component in the network. Thus, we obtain in this case partial placement solution.

[5] The node hosting the component under test

5.3 Illustration

We use the previously studied example in section 2 to illustrate the feasibility of our proposal. In case of connectivity problem as shown in Figure 1 (c), the node $N4$ is forbidden to host testers $T1, T2$ and $T3$. In this case, the placement problem is divided into two sub-problems. In the first one, we consider the connected nodes ($N1, N2$ and $N3$) while searching placement solution for test components $T1, T2$ and $T3$. In the second one, we study the possibility of assigning the test component $T4$ to $N4$. In the following, we detail the first sub-problem as illustrated in Figure 4. First, we introduce for instance the RAM and CPU constraints for the node $N1$ when all nodes in the network are connected:

$$10x_{11} + 10x_{21} + 15x_{31} \leq 50. \tag{1}$$

$$10x_{11} + 5x_{21} + 10x_{31} \leq 20. \tag{2}$$

Next, the objective function is formed as follows with considering that the $maxP$ value is equal to α and $maxP - 1$ is equal to β in this example.

$$\mathcal{Z} = \alpha x_{11} + \beta x_{21} + \beta x_{31} + \alpha x_{12} + \beta x_{22} + \beta x_{32} + \beta x_{13} + \beta x_{23} + \alpha x_{33}. \tag{3}$$

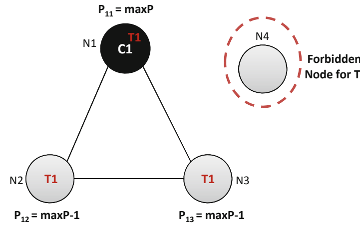

Fig. 4. Illustration Example

The formed MMKP seeks to maximize the equation (3) subject to the constraints such as defined in equations (1) and (2). In the following, we illustrate the derivation of an exact solution of such problem using a well known constraint programming solver called Choco.

6 Realization

To solve our test component placement problem which is modeled as MMKP, we propose a tool illustrated in Figure 5. As inputs, it takes three XML[6] files: nodes provided resources, testers required resources and tester profits. As output, it generates an XML based resource aware test plan which contains for each tester the adequate host to be deployed on. The core of this tool is based on the open source Choco Java library. In the following subsections, we first introduce the Choco library. Next, we demonstrate the mapping between the mathematical formalism to the Choco Java code.

[6] eXtensible Markup Language.

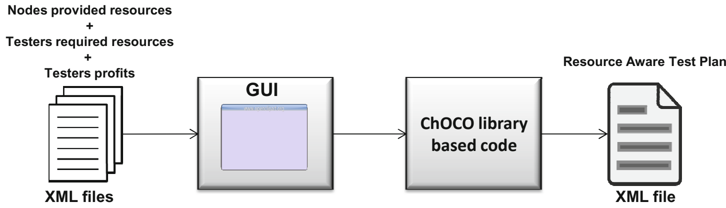

Fig. 5. Architecture of Choco based tool

6.1 Choco Library

Choco is introduced in [18] as a Java library for constraint satisfaction problems (CSP) and constraint programming (CP). It is an open source software which offers a problem modeler and a constraint programming solver. The first one is able to manipulate a large variety of variable types and supports over 70 constraints. The second one can be used in satisfaction mode by computing one solution, all solutions or iterating them. Also, it can be used in an optimization mode (maximization and minimization). We selected this solver because it seems to be one of the most popular within the research community and because it is reliable and stable open source Java solver. In the following, we show how to make use of these two fundamental characteristics of Choco to model and solve our placement problem.

6.2 Modeling and Resolving Our Placement Problem with Choco

To solve the test component placement in execution nodes formulated as MMKP, we use Choco library by defining the variables set of the problem and stating constraints (conditions, properties) which must be satisfied by the solution.

```
1  //Model declaration
2  CPModel model = new CPModel();
3  // Variables declaration
4  IntegerVariable[][] X = new IntegerVariable[n][m];
5  for (int i = 0; i < n; i++) {
6  for (int j = 0; j < m; j++) {
7            X[i][j] = Choco.makeIntVar("X" + i+j, 0, 1);}}
8  //objective variable declaration
9  IntegerVariable Z = Choco.makeIntVar("gain", 1, 1000000,Options.V_OBJECTIVE);
10 //Constraints definition
11 // ...
12 Constraint[] rows = new Constraint[n];
13 for (int i = 0; i < n; i++) {
14    rows[i] = Choco.eq(Choco.sum(X[i]), 1);}
15 model.addConstraints(rows);
16 //Objective function
17 IntegerExpressionVariable []exp1=new IntegerExpressionVariable [n];
18 for (int i = 0; i < n; i++)
19    exp1[i]=Choco.scalar(g[i], X[i]);
20 model.addConstraint(Choco.eq(Choco.sum(exp1),Z));
21 //Solve the problem
22 Solver s = new CPSolver();
23 s.read(model);
24 s.maximize(s.getVar(Z), false);
```

Listing 1. Choco code example

The above Listing 1 presents a brief overview of the model translation from the mathematical representation of our problem to Choco code. In line 7, it shows the declaration of the x_{ij} variable and its domain. Moreover, we display in line 9 the declaration of the objective function that maximize the gain of test component placement. Stating the constraint (5) using the Choco syntax has been illustrated in line 14. To solve the placement problem, two cases exist : obtaining a satisfying solution or an optimal solution. The latter case is illustrated in the line 24.

7 Experimentation

In this section, we present some experiments that are conducted to evaluate the execution time (order of milliseconds) needed for the placement phase. Two cases have been studied in the following subsections. First, we measure the time needed by calculating a satisfying solution. Next, we compute the execution time while optimal solution is required.

All of the experiments were conducted on a PC with Intel Core 2 Duo CPU and 2 GB of main memory having as operating system Microsoft vista. The number of nodes is equal to the number of testers in all the experiments that we have done. Also, we have to note that each experiment is carried out five times to derive the precise average execution time of the placement phase.

7.1 Computation of a Feasible Solution

The graph of Figure 6 shows the average execution time required by the Choco solver to compute a satisfying solution. Analysis of the results indicates that the average time required for assigning test components to execution nodes increases with the increase in number of test components and nodes. The proposed solver may resolve this $\mathcal{NP}$-hard problem in a reasonable amount of time while the number of test components and nodes does not exceed some dozens. Such solution can be sufficient especially when the affected parts of the system to validate after dynamic reconfiguration are not important also when the execution environment is not large.

7.2 Computation of an Optimal Solution

Recall that in this case we search for an exact solution that maximizes the placement profits by assigning test components to the adequate execution nodes. By computing such solution, we aim to reduce the communication cost over the network between the test components and the components under test. As illustrated in Figure 7, the calculation of the optimal solution takes a significant time especially when the number of test components and nodes increases. We have to note that this computation technique can be opted when the dynamic changes are not frequently done. Thus, we have enough time to validate them. Otherwise, it can be enhanced by the use of some predefined heuristics in Choco library.

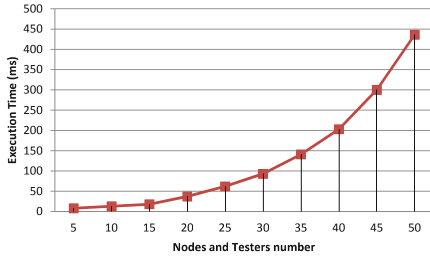

Fig. 6. Execution time needed for computing a feasible solution

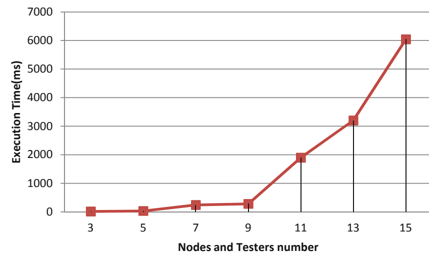

Fig. 7. Execution time needed for computing an optimal solution

8 Conclusion

In this paper, we have studied the runtime testing of adaptable and distributed systems after the occurrence of dynamic changes. This resource intense testing method is often performed in a resource constrained execution environment. For this reason, defining efficiently the distributed test architecture and the assignment of its components to the execution nodes can be a useful solution for either respecting resource constraints or reducing the cost of testing activity.

To do this, we have proposed a new approach for resource aware test architecture design of adaptable and distributed systems. Our main contribution in this work consists in proposing a method for test component placement in the execution nodes while respecting resource and connectivity constraints. This $\mathcal{NP}$-hard problem has been formulated as a multiple multidimensional knapsack problem. We have also implemented a tool facilitating the resolution of our problem using the Choco Java library.

As future work, we will enhance the proposed solution by adding other resource constraints such as network bandwidth or by associating different weights with the considered resources. Moreover, it is obvious that for large scale systems or systems having hard realtime timing constraints, the proposed method is not suitable. In this case, it might take a lot of time to find the exact solution. Therefore, we investigate effort in enhancing the proposed technique using heuristics.

References

1. Brenner, D., Atkinson, C., Malaka, R., Merdes, M., Paech, B., Suliman, D.: Reducing verification effort in component-based software engineering through built-in testing. Information Systems Frontiers 9(2-3), 151–162 (2007)
2. Lahami, M., Krichen, M., Jmaiel, M.: A Distributed Test Architecture for Adaptable and Distributed Real-Time Systems. In: Journées Nationales IDM, CAL, et du GDR GPL, Lille, France (June 2011)
3. Zhang, X., Shan, H., Qian, J.: Resource-Aware Test Suite Optimization. In: Proceedings of the 2009 Ninth International Conference on Quality Software, QSIC 2009, pp. 341–346. IEEE Computer Society, Washington, DC (2009)

4. Merdes, M., Malaka, R., Suliman, D., Paech, B., Brenner, D., Atkinson, C.: Ubiquitous RATs: how resource-aware run-time tests can improve ubiquitous software systems. In: SEM 2006: Proceedings of the 6th International Workshop on Software Engineering and Middleware, pp. 55–62. ACM, New York (2006)
5. Din, G., Tolea, S., Schieferdecker, I.: Distributed Load Tests with TTCN-3. In: TestCom 2006: Proceedings of International Conference for Testing of Communicating Systems, 18th IFIP TC6/WG6.1, pp. 177–196 (May 2006)
6. Csorba, M., Eottevenyi, D., Palugyai, S.: Experimenting with Dynamic Test Component Deployment in TTCN-3. In: 3rd International Conference on Testbeds and Research Infrastructure for the Development of Networks and Communities, TridentCom 2007, pp. 1–8 (May 2007)
7. Maassen, J., Bal, H.E.: Smartsockets: solving the connectivity problems in grid computing. In: Proceedings of the 16th International Symposium on High Performance Distributed Computing, HPDC 2007, pp. 1–10. ACM, New York (2007)
8. Piel, É., González-Sanchez, A., Groß, H.G.: Automating Integration Testing of Large-Scale Publish/Subscribe Systems. In: Hinze, A., Buchmann, A.P. (eds.) Principles and Applications of Distributed Event-Based Systems, pp. 140–163. IGI Global (2010)
9. Gonzalez, A., Piel, E., Gross, H.G., Glandrup, M.: Testing Challenges of Maritime Safety and Security Systems-of-Systems. In: Proceedings of the Testing: Academic & Industrial Conference - Practice and Research Techniques, pp. 35–39. IEEE Computer Society, Washington, DC (2008)
10. King, T.M., Allen, A.A., Cruz, R., Clarke, P.J.: Safe Runtime Validation of Behavioral Adaptations in Autonomic Software. In: Calero, J.M.A., Yang, L.T., Mármol, F.G., García-Villalba, L.J., Li, X.A., Wang, Y. (eds.) ATC 2011. LNCS, vol. 6906, pp. 31–46. Springer, Heidelberg (2011)
11. Mei, L.: A context-aware orchestrating and choreographic test framework for service-oriented applications. In: ICSE Companion, pp. 371–374. IEEE (2009)
12. Flores, A., Augusto, J.C., Polo, M., Varea, M.: Towards context-aware testing for semantic interoperability on PvC environments. In: SMC (2), pp. 1136–1141. IEEE (2004)
13. Rodriguez, I.B.: Dynamic Software Architecture Management for Collaborative Communicating Systems. PhD thesis, University of Sfax & University of Toulouse (March 2011)
14. ETSI ES 201 873-1 (V3.1.1): Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language (2005)
15. Portigliatti, V., Philippe, L.: Java Components with constraints and preferences in automatic administration of execution and placement. In: PDP, pp. 266–273. IEEE Computer Society (2003)
16. Pizzocaro, D., Chalmers, S., Preece, A.: Sensor assignment in virtual environments using constraint programming. In: Ellis, R., Allen, T., Petridis, M. (eds.) The Twenty-seventh SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence Applications and Innovations in Intelligent Systems XV: Proceedings of AI 2007. Winner of Best Poster Presentation, AI 2007, pp. 333–338. Springer (2007)
17. Hermenier, F., Lorca, X., Cambazard, H., Menaud, J.M., Jussien, N.: Reconfiguration dynamique du placement dans les grilles de calculs dirigée par des objectifs. In: 6ième Conférence Francophone sur les Systèmes d'Exploitation (CFSE 2006), Fribourg, Switzerland (2008)

18. Jussien, N., Rochart, G., Lorca, X.: Choco: an Open Source Java Constraint Programming Library. In: CPAIOR 2008 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP 2008), pp. 1–10, Paris, France (2008)
19. Cotta, C., Troya, J.: A hybrid genetic algorithm for the 0-1 multiple knapsack problem. Artificial Neural Nets and Genetic Algorithms 3, 251–255 (1998)
20. PospÍchal, P., Schwarz, J., Jaroš, J.: Parallel Genetic Algorithm Solving 0/1 Knapsack Problem Running on the GPU. In: 16th International Conference on Soft Computing, MENDEL 2010, pp. 64–70. Brno University of Technology (2010)
21. Puchinger, J., Raidl, G.R., Pferschy, U.: The Core Concept for the Multidimensional Knapsack Problem. In: Gottlieb, J., Raidl, G.R. (eds.) EvoCOP 2006. LNCS, vol. 3906, pp. 195–208. Springer, Heidelberg (2006)
22. Martello, S., Toth, P.: Solution of the zero-one multiple knapsack problem. European Journal of Operational Research 4(4), 276–283 (1980)
23. Jansen, K.: Parameterized approximation scheme for the multiple knapsack problem. In: Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics, pp. 665–674 (2009)
24. Song, Y., Zhang, C., Fang, Y.: Multiple multidimensional knapsack problem and its applications in cognitive radio networks. In: Military Communications Conference, MILCOM 2008, pp. 1–7. IEEE (November 2008)