ICCP2018 Companion Paper: SMT Encoding

Mădălina Eraşcu
Department of Computer Science, West University of Timişoara
Institute e-Austria Timişoara,
Blvd. V. Pârvan 4, Timişoara, Romania;
E-mail: madalina.erascu@e-uvt.ro

June 28, 2018

Abstract

In this paper we present the ideas behind the SMT encoding used in ICCP2018 paper.

The problem to be solved is to find a mapping (of application components to VMs) which: (1) satisfies the constraints induced by the interactions between components; (2) satisfies the hardware requirements of all components; and (3) minimize the purchasing price.

To describe the problem in a more formal way, let us consider a set of N interacting components, $\{C_1, C_2, \ldots, C_N\}$ to be assigned to a set of M virtual machines, $\{V_1, V_2, \ldots, V_M\}$. Each component C_i is characterized by a set of requirements concerning the hardware resources, e.g. number of CPUs (HC_i), memory (HM_i) - expressed in MB, storage (HS_i) - expressed in MB. The problem can be formulated as an integer programming problem, aiming to find the values of a set of binary variables, $a_{ik} \in \{0,1\}$ for $i = \overline{1,N}$, $k = \overline{1,M}$ which are interpreted as follows:

$$a_{ik} = \begin{cases} 1 & \text{if } C_i \text{ is assigned to } V_k \\ 0 & \text{if } C_i \text{ is not assigned to } V_k \end{cases} \tag{1}$$

The overall search space is $\{0,1\}^{M\times N}$ but the set of feasible solutions is smaller, being characterized by the constraints described in the following.

In the SMT formalization, we introduced integer variables for V_i, $i = \overline{1,M}$. The initial value for M is computed based on the constraints and represents the maximum number of VMs needed for deployment.

Example 1 `(declare-fun VM1 () Int)`

The values of a are seen as a matrix, each element being an integer variable. For example Ci_VMj is the element (i,j) in the matrix a. It is declared similar to Example 1, for example

Example 2 `(declare-fun C2_VM3 () Int)`

We also have a vector VM called the usage vector which is defined as:

$$(1)vm_k = \begin{cases} 1 & \text{if there is no component assigned to } VM_k \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

The introduction of the vector *vmType* in the implementation was necessary in order to transform the objective function into a linear one. Each element of this vector representing a VM index can have one of the types of VMs from available configurations from Cloud Providers (CPs). Assume that $M = 11$. Then we have:

```
(declare-fun VM1Type () Int)
...
(declare-fun VM11Type () Int)
```

meaning that each of the 11 machines have a certain type.

Now assume there are 40 available offers from CPs. Then the type of $VM1$ can be 1 or 2 or ... 40; of $VM2$ the same, and so on. This is specified as:

```
(assert (or (= VM1Type 1) (= VM1Type 2) ... (= VM1Type 40)))
...
(assert (or (= VM11Type 1) (= VM11Type 2) ... (= VM11Type 40)))
```

We have two main types of constraints: *hardware constraints* which refer to the hardware characteristics of the VM types offered by the Cloud Provider and *interaction constraints* that describe the interactions between components.

Hardware Constraints These constraints are of inequality type and specify that the amount of resources required by the components assigned to a VM does not overpass the VM characteristics, i.e.: are related to components hardware and have to be correlated to VMs offers from cloud providers:

$$\sum_{i=1}^{N} a_{ik} H_i^{res} \leq H_k^{res_{prov}}, \quad k = \overline{1, M} \tag{3}$$

where H^{res} is the required amount of resource (we considered in this paper only CPU, Memory and Storage capacity) and $H^{res_{prov}}$ is the corresponding characteristic of a VM included in an existing Cloud Provider offer.

Therefore, we have the following declarations:

```
(declare-fun PriceProv1 () Int)
...
(declare-fun PriceProv11 () Int)
(declare-fun StorageProv1 () Int)
...
(declare-fun StorageProv11 () Int)
(declare-fun MemProv1 () Int)
...
(declare-fun MemProv11 () Int)
(declare-fun ProcProv1 () Int)
...
(declare-fun ProcProv11 () Int)
```

We also specified that the price of a VM contributes to the final price only if the machine is leased. Therefore, if the machine is not occupied, then the price is 0. Hence, we have:

```
(assert (=> (= VM1 0) (= PriceProv1 0)))
...
(assert (=> (= VM11 0) (= PriceProv11 0)))
```

We also encoded a VM offers as logical formulas. We have:

```
(assert (=>
  (and (= VM1 1) (= VM1Type 1))
  (and (= PriceProv1 8403) (= ProcProv1 64) (= MemProv1 976000) (= StorageProv1 1000))))
(assert (=>
  (and (= VM2 1) (= VM2Type 1))
  (and (= PriceProv2 8403) (= ProcProv2 64) (= MemProv2 976000) (= StorageProv2 1000))))
...
(assert (=>
  (and (= VM1 1) (= VM1Type 2))
  (and (= PriceProv1 9152) (= ProcProv1 64) (= MemProv1 488000) (= StorageProv1 8000))))
(assert (=>
  (and (= VM2 1) (= VM2Type 2))
  (and (= PriceProv2 9152) (= ProcProv2 64) (= MemProv2 488000) (= StorageProv2 8000))))
...
```

Relations of type (3) are written as, for example:

```
(assert (<= (+ 0 (* C1_VM9 500) (* C2_VM9 500) (* C3_VM9 2000) (* C4_VM9 2000)
                 (* C5_VM9 3000) (* C6_VM9 1000) (* C7_VM9 500) (* C8_VM9 2000)
                 (* C9_VM9 1000) (* C10_VM9 2000))
         StorageProv9))

(assert (<= (+ 0 (* C1_VM10 500) (* C2_VM10 500) (* C3_VM10 2000) (* C4_VM10 2000)
                 (* C5_VM10 3000) (* C6_VM10 1000) (* C7_VM10 500) (* C8_VM10 2000)
                 (* C9_VM10 1000) (* C10_VM10 2000))
        StorageProv10))
...
```

Interaction Constraints These constraints correspond to various types of interactions identified in the case studies:

1. some components cannot be assigned to the same VM (*conflict*);
2. some components should be assigned to the same VM (*collocation*);
3. some components cannot be simultaneously deployed, not even on the same VM (*exclusive deployment*);
4. some components require functionalities offered by other components inducing constraints on the number of deployed instances (*require-provide*)

Besides the constraints expressing interactions between components, there are constraints referring to the number of deployed instances for a given component:

- some components should be deployed on all VMs (*full deployment*);
- the number of instances of a given component should check some lower and/or upper bound constraints (*deployment with bounded number of instances*).

0.1 Constraints Description

Conflicts This case corresponds to situations when there are conflictual components which cannot be deployed on the same VM. For example, in the case of the *Secure Web Container* application, we have the constraint that the Balancer component cannot be deployed on the same machine where any of components Apache or Nginx reside. In the general case we consider that all conflicts between components are encoded in a matrix, R, defined by: $R_{ij} = 1$ if C_i and C_j are conflictual components and $R_{ij} = 0$ otherwise. Thus these constraints can be described as a set of linear inequalities:

$$a_{ik} + a_{jk} \leq 1, \quad k = \overline{1, M}, \quad \text{for all } (i,j) \text{ such that } R_{ij} = 1.$$

Collocation This means that once a component is deployed on a VM an instance of the other component should be also deployed on the same VM. For example, in case of Oryx2 application, the components HDFS.DataNode and Spark.Worker must the collocated. In the general case, the collocation dependencies can be stored in a matrix D (i.e. $D_{ij} = 1$ if C_i and C_j should be collocated and $D_{ij} = 0$ otherwise) and the constraints can be described as a set of equalities:

$$a_{ik} = a_{jk}, \quad k = \overline{1, M}, \quad \text{for all } (i,j) \text{ such that } D_{ij} = 1$$

Exclusive deployment There are cases when two, or more, components cannot be simultaneously deployed. For instance, in the case of the Wordpress application, only one type of Balancer must be deployed even if there are two types of the Balancer components (HTTPLoadBalancer, DNSLoadBalancer). Such a constraint can be described as:

$$H(\sum_{k=1}^{M} a_{ik}) + H(\sum_{k=1}^{M} a_{jk}) = 1, \text{ for all } (i,j) \text{ such that } E_{ij} = 1$$

where E is the matrix describing the exclusive deployment relation (i.e. $E_{ij} = 1$ if C_i and C_j should be exclusively deployed and $E_{ij} = 0$ otherwise) and H is a Heaviside-like function ($H(u) = 1$ if $u > 0$ and $H(u) = 0$ if $u = 0$).

Require-Provide A special case of interaction between components is when one component requires some functionalities offered by other components as for instance in the case of Wordpress case study where a Wordpress component requires at least three instances of MySQL and one MySQL instance can serve at most 2 Wordpress instances. Such an interaction induces constraints on the number of instances corresponding to the interacting components as follows: *(1)* C_i requires (consumes) at least n_{ij} instances of C_j and *(2)* C_j can serve (provides) at most n_{ji} instances of C_i. This can be written as:

$$n_{ij}\sum_{k=1}^{M} a_{ik} \le n_{ji}\sum_{k=1}^{M} a_{jk}, \quad n_{ij}, n_{ji} \in \mathbb{N}$$

Another related case appearing in the *Secure Web Container* case study is when for each set of n instances of component C_j a new instance of C_i should be deployed. This can be described as:

$$0 < n\sum_{k=1}^{M} a_{ik} - \sum_{k=1}^{M} a_{jk} \le n, \quad n \in \mathbb{N}$$

Full Deployment As *Secure Web Container* and *Oryx2* case studies illustrate, there can be also cases when a component C_i must be deployed on all VMs (except on those which would induce conflicts). This can be expressed as:

$$\sum_{k=1}^{M}(a_{ik} + H(\sum_{j,R_{ij}=1} a_{jk})) = M$$

where R is the conflicts matrix and H is the Heaviside-like function defined above.

Deployment with bounded number of instances There are situations when the number of instances corresponding to a set of deployed components, $\overline{C}$, should be equal, greater or less than some values. For instance, in the case of Secure Web Container application, the total amount of instances of the Apache and Nginx components must be at least 3. These types of constraints can be described as follows:

$$\sum_{k=1}^{M} a_{ik} \ \langle \text{op} \rangle \ n, \quad \forall i \in \overline{C}, \quad \langle \text{op} \rangle \in \{=, \le, \ge\}, \quad n \in \mathbb{N}$$

Finally the optimization function is:

```
(+ 0 PriceProv1 PriceProv2 PriceProv3 PriceProv4 PriceProv5 PriceProv6
     PriceProv7 PriceProv8 PriceProv9 PriceProv10 PriceProv11)
```