ISBN 978-981-14-4787-7
Proceedings of 2020 the 10th International Workshop on Computer Science and Engineering (WCSE 2020)
Yangon (Rangoon), Myanmar (Burma), February 26- February 28, 2020, pp. 146-150
doi: 10.18178/wcse.2020.02.025

A CSP-based Approach to Design a Subnet Solving a Network Construction Exercise for Beginners

Yuichiro Tateiwa[1+] and Yoshifumi Hisanaga[1]

[1]Nagoya Institute of Technology, Japan

Abstract. When creating exercise problems for network construction, teachers confirm whether the problems are solvable, and then they create a correct answer. However, the work is troublesome and may result in some mistakes such as creating unsolvable problems and incorrect answers. This paper proposes a simulator that computes communications in exercise problems, and CSP formulas for solving exercise problems by analyzing the simulator with symbolic execution. In experiments, In experiments, solving the formulas with CSP solver Z3 found correct answers and unsolvable problems.

Keywords: Zeroconf, CSP, SMT, Z3, network construction, e-learning

1. Introduction

It is important to increase the number of network engineers who administer computer networks. These network engineers can develop the infrastructure to develop a ubiquitous network society and to provide new services for it. The experience of constructing a basic network is useful not only to the network administrators but also to the network application programmers and the network system designers.

When creating exercise problems for network construction, teachers confirm whether the problems are solvable, and then they create a correct answer. However, the work is troublesome and may result in some mistakes such as creating unsolvable problems and incorrect answers.

Let us consider a tool that generates network settings sequentially and executes a network simulator with each of the settings. If the tool confirms that one of the network behaviors satisfies all the requirements of the exercise problem, it outputs the setting as one of the correct answers. However, finding a correct answer requires a significant amount of time because there is a large search space that includes combinations of setting values such as Internet Protocol addresses (IP addresses) and subnet masks.

Symbolic execution [1] is an execution method that executes a program with symbolic values as inputs rather than concrete values. Interpreting a program by propagating symbols can analyze the relationships between the input variables and the inner variables, along with the relationships between the input variables and the branch conditions for an execution path. After finding inner variables and an execution path for satisfying all the requirements of an exercise problem in the simulator, the relationships between them and input variables that store a network setting can be analyzed. The relationships are helpful to reduce the search space because of meaning constraints of network settings.

A Constraint Satisfaction Problem (CSP) is formulated by a set of variables, domain of the variables, and constraints of the variables. If CSP solvers find a set of concrete values that satisfy the formulas, the solvers output the set; otherwise, the solvers notify there are no values satisfying the formulas.

This paper proposes a CSP-based approach to find an example of correct network settings. This method consists of the following two parts:

+ Corresponding author. Tel.: +81527355450
E-mail address: tateiwa@nitech.ac.jp

1) A simulator to calculate communications that are specified in requirements of the exercise problems

2) CSP formulas consisting of network settings as the set of variables, original rules of each parameter as the domains, and the relationships based on the symbolic execution as the constraints

2. Related Work

Zeroconf [2] is a set of technologies that assign an IP address to a new device that is connected to a network. Zeroconf also resolves host names to IP addresses and detects network services in the network. IPv4 Link-local addressing [3] is a technology used in Zeroconf for IP addressing. A host with the technology can search for an unused IPv4 address (169.254/16) in a subnet including itself. The Dynamic Host Configuration Protocol (DHCP) [4] searches for an IP address that is unused in a network that is connected to the DHCP server and is included in the range of the server settings. However, the exercise problems also require design of Media Access Control address (MAC address) assignments and cable connections, which are not supported by the above technologies. The technologies cannot determine whether there is a solution to the exercise problems.

3. Proposed Approach

3.1. Preliminaries

The function *Value(s, e)* in a step *s* and an expression *e* returns the output of *e* just before starting *s*. The function *Value(s, v)* in a step *s* and a variable *v* returns the value of *v* just before starting *s*. The sequence element X_i returns the *i*-th element in *X*. The operator *.(dot)* accesses the value of a member variable in a tuple.

3.2. Network Settings

Table 1 denotes the network setting items and their data structure used in the exercises. The students can set values just to the underlined variables. Hosts equip one Ethernet port named ep1 and switching hubs equip five Ethernet ports named ep1-ep5. It is assumed that communication data can be exchanged between a host and a switching hub.

3.3. Exercise Problems

Exercise problems consist of **communication examples** and a **setting requirement**.

A communication example is an example of Internet Control Message Protocol (ICMP) echo communications, which should be established in correct networks. Let us consider that a device *src* sends an ICMP echo request packet with a destination IP address *dip*. The data structure of a **send setting** is a tuple *(src, dip)*. The data structure of a communication example is a 3-tuple *(snd, OW, HW)* for which *OW* and *HW* are a sequence of devices which are sorted in arrival order of the ICMP echo request and its reply on send setting *snd*, respectively. *OW* and *HW* can be referred to as communication routes.

A setting requirement consists of devices that must be installed into networks, values that must be set in devices, and Ethernet cables that must be connected to the assigned devices, and have the same data structure as that of items in Table 1. All the items must be assigned concrete values, unless underlined items are assigned "*" (denoting arbitrary values), to which students assign concrete values in exercises.

In actual exercise problems, communication examples and a setting requirement are expressed using natural language and figures. Fig. 1 shows an example of exercise problems based on Fig. 2. The upper terms in the squares denote the device type and the lower terms represent the device identifier.

3.4. Communication Simulator

This study proposes a communication simulator for computing communication routes of ICMP echo in Fig. 3 and Table 2. The function *Out* works for transmission of ICMP echo data in a host. The function *Fwd* implements reception and transmission of the data in a switching hub. The function *In* realizes reception and response of the data in a host. The parameters *nd*, *ep*, *pl*, *dip*, *dm*, *sip*, and *sm* of the functions denote a device identifier, an Ethernet port name, a payload, a destination IP address, a destination MAC address, a source IP

address, and a source MAC address, respectively. When establishing ICMP echo communication that satisfies *OW* and *HW*, statements in the simulator are executed as follows:

1. When OW_1 sends an ICMP echo Request *req*, steps 1-8 and 37 are executed. *Value(8, dev2)* is equal to OW_2.
2. When OW_2 receives and sends *req*, steps 9-11 and 36 are executed. *Value(11, dev2)* is equal to OW_3.
3. When OW_3 receives *req*, steps 12-16 and 35 are executed.
4. When HW_1 (i.e., OW_3) sends the ICMP echo reply *rep* (i.e., the response to *req*), steps 17-24, 34, and 35 are executed. *Value(24, dev2)* is equal to HW_2.
5. When HW_2 receives and sends *rep*, steps 25-27 and 33 are executed. *Value(27, dev2)* is equal to HW_3.
6. When HW_3 receives *rep*, steps 28-32 are executed.

Table 1: Network Setting Items and The Corresponding Data Structure.

Name		**Data structure**
Network		Tuple (Set of devices *Devices*, Set of cables *Cables*)
Device	Host	Tuple (identifier *id*, IP address *ip*, subnet mask *mask*, MAC address *mac*, routing table entries *re1*, *re2*, *re3*, ARP cache entries *ae1*, *ae2*, *ae3*)
	Switching hub	identifier *id*
Cable		Set {(device identifier *dev*, Ethernet port name *ep*), (device identifier *dev*, Ethernet port name *ep*)}
Routing table entry		Tuple (destination IP address *ip*, destination network address *nwaddr*, next hop IP address *nh*, *nwaddr*'s subnet mask *mask*, sender Ethernet port name *ep*)
Arp cache entry		Tuple (IP address *ip*, MAC address *mac*)

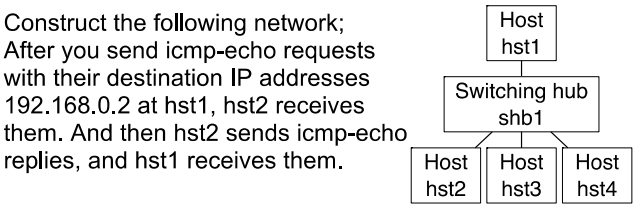

Fig. 1: Example of an Exercise Problem.

```
SR = ({hst1, hst2, hst3, shb1}, {cbl1, cbl2, cbl3, cbl4})
CE = {(snd, OW, HW)}
     snd = (hst1, 192.168.0.2)      OW = <hst1, shb1, hst2>
hst1 = (hst1, *, *, *, re1, re2, re3, ae1, ae2, ae3)
     re1 = (*, *, *, *, *)          ae1 = (*, *)  shb1 = shb1
cbl1 = (hst1, *, shb1, *)
```

Fig. 2: Part of Components of The Communication Example.

Code	Step No.			
def Out(dev, pl, dip)				
(ip, ep) = L3Rtng(dev, dip)	1	17		
if(ep == 'ep1')	2	18		
sip = SIP(dev, ep)	3	19		
if(sip != '')	4	20		
sm = SMac(dev, ep)	5	21		
dm = DMac(dev, ep, ip)	6	22		
if(dm != '')	7	23		
(dev2, ep2) = Next(dev, ep)	8	24		
Fwd(dev2, ep2, pl, dip, dm, sip, sm)			34	37
end				
end				
end				
end				
def Fwd(dev, ep, pl, dip, dm, sip, sm)				
ep2 = L2Rtng(dev, ep, dm)				
if(ep2 != '')	9	25		
(dev2, ep3) = Next(dev, ep2)	10	26		
In(dev2, ep3, pl, dip, dm, sip, sm)	11	27		
end			33	36
end				
def In(dev, ep, pl, dip, dm, sip, sm)				
chkmac = ChkDMac(dev, ep, dm)				
if(chkmac)	12	28		
chkip = ChkDIP(dev, dip)	13	29		
if(chkip)	14	30		
if(pl == 'REQUEST')	15	31		
Out(dev, 'REPLY', sip)	16	32		35
end				
end				
end				
end				

Fig. 3: Communication Simulator

Table 2: Functions Used in The Simulator.

Name	Description
ChkDIP (dev, dip)	If *Obj(dev).ip=dip* is satisfied, this function returns *true*; otherwise, it returns *false*.
ChkDMac (dev, ep, dm)	If *ep='ep1' ∧ dm=mac* is satisfied, this function returns *true*; otherwise, it returns *false*.
DMac (dev, dip, ep)	If *Obj(dev).ae1.ip=dip* is satisfied, this function returns *Obj(dev).ae1.mac*; else, if *Obj(dev).ae2.ip=dip* is satisfied, this function returns *Obj(dev).ae2.mac*; else, if *Obj(dev).ae3.ip=dip* is satisfied, this function returns *Obj(dev).ae3.mac*; else if there is an *ne* for which *Obj(ne.dev).ip=dip ∧ ne.dev≠dev* is satisfied in *ne ∈Neigh(nx.dev)*, *nx=Next(dev, ep)*, this function returns *Obj(ne.dev).mac; otherwise*, this function returns an empty string.
L2Rtng (dev, ep, dm)	If there is an *ne* for which *Obj(ne.dev).mac=dm* is satisfied in *ne ∈Neigh(dev)*, this function returns *Next(ne.dev, ne.ep).ep* for the first found *ne;* otherwise, this function returns an empty string.
L3Rtng (dev, dip)	If there is an *r* for which *r.ip* matches *dip* by the longest prefix match or for which *r.nw* and *r.mask* match *dip* by the longest prefix match in *r ∈{Obj(dev).re1, Obj(dev).re2, Obj(dev).re3}*, this function returns a tuple *(r.nh, r.ep)* (*r.nh≠0.0.0.0* is satisfied), a tuple *(dip, r.ep)* (*r.nh=0.0.0.0* is satisfied), or a tuple *('', '')* (otherwise).
Neigh(dev)	This function returns a set *{nx | nx≠('',''), nx=Next(dev, ep), ep ∈NI(dev)}*.
Next (dev, ep)	If there is a *c* in *{(dev, ep), c} ∈nw.Cables*, this function returns *c*; otherwise, this function returns *('', '')*.
NI(dev)	This function returns a set of Ethernet port names equipped on *dev*. If *dev* is a host, this function returns a set {'ep1'}; else if *dev* is a switching hub, this function returns a set {'ep1', 'ep2', 'ep3', 'ep4', 'ep5'}.
Obj(dev)	If there is a *obj* for which *id=obj.id* in *obj ∈nw.Devices*, this function returns *obj*.
SIP(dev, ep)	If *ep='ep1'* is satisfied, this function returns *Obj(dev).ip*; otherwise, it returns an empty string.
SMac (dev, ep)	If *ep='ep1'* is satisfied, this function returns *Obj(dev).mac*; otherwise, it returns an empty string.

It can be determined whether an answer a*ns* satisfies a communication example *ce* with the following steps:

1. Set up the global variable *nw* of the simulator based on *ans.*
2. Execute the function *Out(ce.snd.src, 'REQUEST', ce.snd.dip)*.
3. If the simulator acts by following the six steps described above, then *ans* satisfies *ce*.

3.5. CSP Formulation

In order to find network settings *nw* that satisfies both setting requirements *sr* and a communication example *ce*, this paper formulates exercise problems as CSPs.

- **Variables:** Empty variables in *nw*, which correspond to the variables whose values are assigned as "*" in *sr*.
- **Domains:** These are shown in Table 3. The variable *dev* in Cable stores an integer identifying hosts and switching hubs, which are assigned with a unique integer that ranges from 1 to the number of devices. The variable *ep* in Cable stores an integer that corresponds to a name of Ethernet port; the integer *i* corresponds to Ethernet port 'ep*i*'.
- **Constraints:** *Eq. 1 ∧ Eq. 2 ∧ Eq. 3* using the following expressions

In order to execute the simulator with *snd*, the following constraints are required.

$$Value(1, nd) = ce.snd.src \wedge Value(1, pl) = 'REQUEST' \wedge Value(1, dip) = ce.snd.dst \quad (1)$$

In order for the simulator to perform steps 1-37, the following constraints are required.

$$\begin{aligned} &Value(2, ep == 'ep1') = Value(4, sip != '') = Value(7, dm != '') = Value(10, ep2 != '') \\ &= Value(13, chkmac) = Value(15, chkip) = Value(16, pl=='REQUEST') = Value(18, ep != '') \\ &= Value(20, sip != '') = Value(23, dm != '') = Value(26, ep2 != '') = Value(29, chkmac) \\ &= Value(31, chkip) = true \wedge Value(32, pl=='REQUEST') = false \end{aligned} \quad (2)$$

In order that the value of variables storing reception devices in the simulator satisfies *ce.OW* and *ce.HW*, the following constraints are required.

$$Value(8, dev2)=ce.OW_2 \wedge Value(11, dev2) = ce.OW_3 \wedge Value(24, dev2) = ce.HW_2 \wedge Value(27, dev2) = ce.HW_3 \quad (3)$$

Table 3: Domains of The Variables.

Variables		Domains
Host	*ip*	[0, 4294967295]
	mask	[0, 4294967295]
	mac	[0, 281474976710655]
Cable	*dev*	[1, size of Devices]
	ep	[1, 5]
Routing table entry	*ip*	[0, 4294967295]
	nwaddr	[0, 4294967295]
	nh	[0, 4294967295]
	mask	[0, 4294967295]
	ep	1
Arp cache entry	*ip*	[0, 4294967295]
	mac	[0, 281474976710655]

4. Experiments

This study implemented a prototype of the CSP formulas with the SMT solver Z3 4.4.1 [5]. The prototype evaluated the formulas with the exercise problem in Fig. 1 and three types of network settings on a PC that featured a 2.93 GHz CPU and a 4 GB main memory.

No. 1 in Table 4 shows the prototype took network settings that had no values and then outputted concrete values that were regarded as correct. No. 2 indicates the prototype took an example of the correct settings and then reported that the settings satisfied the constraints; we agreed with the opinion. No. 3 suggests that the prototype took incorrect settings and then notified that the settings did not satisfy the constraints; we agreed with the opinion. While the prototype ran, the execution times were measured manually. Each execution time was less than one second.

Table 4: Evaluation of The Results.

No.	Type of settings	Output by prototype	Opinion of authors	Execution time (sec.)
1	Empty	Concrete settings	Correct	< 1.0
2	Correct	Satisfactory	Agree	< 1.0
3	Incorrect	Unsatisfactory	Agree	< 1.0

5. Conclusion

A simulator that computes the communications in the exercise problems was proposed in this study. This paper also describes the CSP formulas for solving the exercise problems by analyzing the simulator with symbolic execution. The experiments show that the results for solving the formulas with the CSP solver Z3 are as per our estimations.

In our future work we will focus on expanding this method to deal with larger networks, including routers, and developing a tool for generating CSP formulas based on the exercise problems.

6. References

[1] James C. King, “Symbolic execution and program testing,” Communications of the ACM, Vol. 19, No. 7, pp.385-394, July 1976.

[2] Zero Configuration Networking (Zeroconf), http://www.zeroconf.org, accessed Dec. 16, 2019.

[3] S. Cheshire, B. Aboba, E. Guttman, “Dynamic Configuration of IPv4 Link-Local Addresses,” RFC 3927, May 2005.

[4] R. Droms, “Dynamic Host Configuration Protocol, ” RFC 2131, March 1997.

[5] Z3, https://github.com/Z3Prover/z3, accessed Dec. 20, 2019.