Distributed Programming II course

Politecnico di Torino - AA 2015/16

Teacher: Riccardo Sisto
Teacher assistant: Matteo Virgilio

Report of special project 3:

Porting to Java of Python code for reachability verification in VNF chains

Indice

Capitolo 1

Requirements

This assignment consists of the following parts:

Part a) Porting to Java of a set of Python classes developed at UC Berkeley for performing reachability verification on chains of VNFs. The source code of the Python classes is available at https://github.com/apanda/modeling. These classes generate models in the form of Boolean expressions, to be fed as input to a Boolean satisfiability checker (Z3). In order to generate Boolean expressions, the Python classes use the Z3 Python API. The corresponding Java classes have to use the equivalent Java API available in Z3. Some of the Python classes deal with the various types of VNFs (end-hosts, firewall, and other), while others deal with the model of the entire network. An explanation of how the Python classes operate will be given to the student who has been assigned the project.

Part b) Design and implementation of a Java web service that can perform VNF chain verification through the Java classes developed at the previous point. The web service must be able to receive, in a proper format:

- the chain that connects two end-hosts (source and destination nodes)
- the configuration required by each function that belongs to the above chain.

A documentation of the XML format and of the web service, a client for testing the service, and an ant script that automates compilation and running of the various programs and tests must also be provided as part of the solution.

Capitolo 2

Development

2.1 Part a)

First of all I installed the z3 library, paying attention to install the 32-bit version which was the same bit version of Python in my PC. Once I installed and configured the Z3 library in my Windows 8 environment, equipped with Java 7 and Python, I started to learn the structure of the Python project, which was composed by the one developed at UC Berkley and some specific test classes implemented at Politecnico di Torino.
The name of the library was *verigraph*, so in this document I will refer to it and to the developed java version as *verigraph* and *j-verigraph*, respectively.
The specific goals of my work were:

- Porting of the core part of the "model checking for networks" library, from Python to Java, with a focus on reachability verification.

- Porting of custom tests and VNFs done at Politecnico.

The difficult part of this phase of my work is linked to the Z3 Boolean satisfiability checker library. In fact, though this library provides API for the most common programming languages including Java, for some reason there is no documentation for every specific language, so the only way to learn how to traduce Z3 operations was a java file with an example of usage of the various API.
In particular, problems arose when I found that there are many inconsistencies between the nomenclatures of the methods in the Python API and Java API, and that the way to use those API sometimes is quite different.
About that last thing, one thing I had to add in *j-veriigraph* was the "Context" object which according to Java Z3 API is a mandatory object to be passed to every Java object that uses Z3 methods, i.e. our VNF. I will discuss some of the other choices I had to make due to Z3 Java API while presenting the specific implemented classes.

Other choices I mad are linked to the programming languages itself, and are about:

- Python lambda functions: I decided not to traduce them using the Java 8 lambda functions in order to have compatibility with previous java versions.
- Python variable number of arguments: the only thing to do in Java seems to be having a variable number of arguments of the same type, so I traduced the arguments of the functions which had **args* and ***kwargs* with *Object[]...*
- Python dynamic variable names: this feature doesn't exist in Java, so when needed I created HashMaps with the name of the created variable as key and the object as value. In this way there was a structure to be populated dynamically with different variable names to be called like they are actual fields of a class.

The result of this phase is the following.
The *j-verigraph* project includes 4 packages, which represent the subfolders of *verigraph*:

j-verigraph	***verigraph***
mcnet.components mcnet.netobjs	mcnet/components
tests	tests
tests.examples	tests/examples

The added package "mcnet.netobjs" simply includes the classes implementing VNFs which originally were in the folder "mcnet/components". This has been done in order to have more clarity, separating VNF from the classes implementing the generic model of the entire network.

2.1.1 mcnet.components

In this package there are the classes regarding the generic model of a network and the way to perform the reachability verification in that context. The Java-Python classes correspondence:

j-verigraph	*verigraph*
Checker.java	checker.py
Core.java	core.py
NetContext.java	context.py
Network.java	network.py
NetworkObject.java	
DataIsolationResult.java	
IsolationResult.java	
Result.java	
Tuple.java	

Checker.java

This class implements various checks for specific properties in the network.
I couldn't implement the methods "AssertionsToHTML" and "ModelToHTML" because in Java Z3 API there is no correspondence with the methods *z3.in_html_mode()* and *z3.set_html_mode()* defined in Python Z3 API.

Core.java

This is the main component, the core of everything in the library.
The original *verigraph* file contained also the "NetworkObject" class, which I choose to put away in another file.

NetContext.java

Contains basic fields and other things required for model checking.
The original file "context.py" was renamed in "NetContext.java" because there is a conflicting class "Context" that is used in every Z3 Java application.
The method *failurePredicate* contained an attribute "failed" for which I found no matching in Java, and furthermore this method was never used in the whole library, so I didn't implement it in *j-verigraph*.

Network.java

It's the model for a network, encompasses routing and wiring.

NetworkObject.java

This is the abstract class of a generic VNF, which must be inherited by every specific VNF.
The class "NetworkObject" was originally defined in the "core.py" file.

DataIsolationResult.java, IsolationResult.java, Result.java

In *verigraph* these data structures were implemented as return classes in "checker.py" and their aim is to structure the responses to the various types of check requests.

Tuple.java

Finally the class "Tuple" is simply a utility to make a generic couple of objects with different types in Java (HashMap was firstly used, but there was the non-duplicate key limitation).

2.1.2 mcnet.netobjs

Here we have the specific VNF used in our implemented tests. These are a subset of all the existing VNF in *verigraph*, and are just the needed ones for the tests implemented at Politecnico di Torino.
From the functional point of view those Java classes are exactly equal to the original Python ones, and the names are quite self-explanatory. The correspondence is:

j-verigraph	***verigraph***
AclFirewall.java	aclfirewall.py
DumbNode.java	dumb_node.py
EndHost.java	endhost.py
PolitoAntispam.java	PolitoAntispam.py
PolitoCache.java	PolitoCache.py
PolitoErrFunction.java	PolitoErrFunction.py
PolitoMailClient.java	PolitoMailClient.py
PolitoMailServer.java	PolitoMailServer.py
PolitoNat.java	PolitoNat.py
PolitoNF.java	PolitoNF.py
PolitoWebClient.java	PolitoWebClient.py
PolitoWebServer.java	PolitoWebServer.py

2.1.3 tests

In this package there are the mains to start the reachability tests. In particular only the custom tests implemented at Politecnico di Torino were ported.
The correspondence is the following:

j-verigraph	*verigraph*
Polito_AntispamTest.java	polito_antispam_test.py
Polito_CacheFwTest.java	polito_cache_fw_test.py
Polito_CacheNatFwTest.java	polito_cache_nat_fw_test.py
Polito_CacheTest.java	polito_cache_test.py
Polito_DoubleFwTest.java	polito_double_fw_test.py
Polito_SimpleTest.java	polito_simple_test.py

2.1.4 tests.examples

Here there are the network configurations of the tests already mentioned, thus the classes represent the specific network models used for the tests, in which we configure and link together the various VNF. The implemented Java classes with the corresponding Python classes are the following:

j-verigraph	*verigraph*
PolitoAntispamTest.java	PolitoAntispamTest.py
PolitoCacheFwTest.java	PolitoCacheFwTest.py
PolitoCacheNatFwTest.java	PolitoCacheNatFwTest.py
PolitoCacheTest.java	PolitoCacheTest.py
PolitoDoubleFwTest.java	PolitoDoubleFwTest.py
SimpleFwTest.java	SimpleFwTest.py

2.1.5 constraints summary

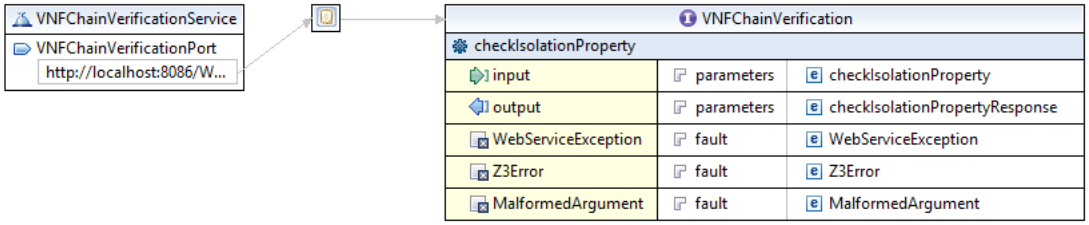
Here is a table with a more readable version of each Z3 boolean constraint used to define and configure the VNF, the network models of the test examples and to check the isolation/reachability property of two chosen nodes.

Class - method	Constraints explaination
AclFirewall	send(fw, n_0, p, t_0) -> (∃ n_1,t_1 : (recv(n_1, fw, p, t_1) && t_1 < t_0 && !acl_func(p.src,p.dest))
	acl_func(a_0,a_1) == or(foreach ip1,ip2 in acl_map ((a_0 == ip1 && a_1 == ip2)||(a_0 == ip2 && a_1 == ip1)))
EndHost	send(node, n_0, p, t_0) -> nodeHasAddr(node,p.src)
	send(node, n_0, p, t_0) -> p.origin == node
	send(node, n_0, p, t_0) -> p.orig_body == p.body
	recv(n_0, node, p, t_0) -> nodeHasAddr(node,p.dest)
PolitoAntispam	if(isInBlackList(ef_0) == or(for bl in blacklist (ef_0==bl)) ? true : false
	isInblackList(ef_0) == false
	send(politoAntispam, n_0, p, t_0) && p.proto(POP3_RESP) -> (∃ n_1,t_1 : (recv(n_1, politoAntispam, p, t_1) && t_1 < t_0)) && !isInBlackList(p.emailFrom)
	send(politoAntispam, n_0, p, t_0) && p.proto(POP3_REQ) -> (∃ n_1,t_1 : (recv(n_1, politoAntispam, p, t_1) && t_1 < t_0))
	send(politoAntispam, politoErrFunction, p, t_0) -> (∃ n_1,t_1 : (recv(n_1, politoAntispam, p, t_1) && t_1 < t_0 && p.emailFrom ==1))
	send(politoAntispam, n_0, p, t_0) -> p.proto == POP_REQ || p.protpo == POP_RESP
	send(politoAntispam, n_0, p, t_0) -> nodeHasAddr(politoAntispam,p.src)
PolitoCache	if(isInternalNode(a_0) == or(listadeinodiinterni) ? True : false
	isInCache(u_0,t_0), ∃ t_1, t_2, p_1, p_2, n_1,n_2 : (t_1< t_2 < t_0 && recv(n_1, politoCache, p_1, t_1) && recv(n_2, politoCache, p_2, t_2))) && p_1.proto == HTTP_REQ && p_2.proto == HTTP_RESP && isInternalNode(n_1) && !isInternalNode(n_2) && p_1.url == u_0 && p_2.url == u_0)
	send(politoCache, n_0, p, t_0) && !isInternalNode(n_0) -> (∃ t_1,n_1 : (t_1 < t_0 && recv(n_1, politoCache, p, t_1) && p.proto == HTTP_REQ && !isInCache(p.url,t_0))
	send(politoCache, n_0, p, t_0) && isInternalNode(n_0) -> (∃ p_1,t_1 : (t_1 < t_0 && recv(n_0, politoCache, p_1, t_1) && p_1.proto == HTTP_REQ && p.proto == HTTP_RESP && p_1.url == p.url && p.src == p_1.dest && p.dest==p_1.src && isInCache(p.url,t_0))
PolitoMailClient	send(politoMailClient, n_0, p, t_0) -> nodeHasAddr(politoMailClient,p.src)
	send(politoMailClient, n_0, p, t_0) -> p.origin == politoMailClient
	send(politoMailClient, n_0, p, t_0) -> p.orig_body == p.body
	recv(n_0, politoMailClient, p, t_0) -> nodeHasAddr(politoMailClient,p.dest)
	send(politoMailClient, n_0, p, t_0) -> p.proto == POP3_REQ
	send(politoMailClient, n_0, p, t_0) -> p.dest == ip_mailServer
PolitoMailServer	send(politoMailServer, n_0, p, t_0) -> nodeHasAddr(politoMailServer,p.src)
	send(politoMailServer, n_0, p, t_0) -> p.origin == politoMailServer
	send(politoMailServer, n_0, p, t_0) -> p.orig_body == p.body
	recv(n_0, politoMailServer, p, t_0) -> nodeHasAddr(politoMailServer,p.dest)
	send(politoMailServer, n_0, p, t_0) -> p.proto == POP3_RESP && p.emailFrom == 1
	send(politoMailServer, n_0, p, t_0) -> (∃ p_1,t_1 : (t_1 < t_0 && recv(n_0, politoMailServer, p_1, t_1) && p_0.proto == POP3_RESP && p_1.proto == POP3_REQ && p_0.dest == p_1.src)
PolitoErrFunction	send(politoErrFunction, n_0, p, t_0) -> 1 == 2
PolitoNat	send(nat, x, p_0, t_0) && !private_addr_func(p_0.dest) -> p_0.src == ip_politoNat && (∃ y, p_1 e t_1 | (recv(y, nat, p_1, t_1) && t_1 < t_0 && private_addr_func(p1.src) && p_1.origin == p_0.origin && <...stesso per dest,orig_body,body,seq,proto,emailFrom,url,options...>)
	send(nat, x, p_0, t_0) && private_addr_func(p_0.dest) -> !private_addr_func(p_0.src) && (∃ y, p_1 e t_1 | (recv(y, nat, p_1, t_1) && t_1 < t_0 && !private_addr_func(p1.src) && p_1.dest == ip_politoNat && p_1.origin == p_0.origin && <...stesso per src,orig_body,body,seq,proto,emailFrom,url,options...>
PolitoNat -setInternalAddress	private_addr_func(n_0) == or(n_0==n foreach internal address)
PolitoNF	myFunction(a_0,a_1) == ((a_0 == ipA && a_1 == ipB) || (a_0 == ipB && a_1 == ipA))
	send(politoNF, n_0, p, t_0) -> (∃ n_1,t_1 : (t_1 < t_0 && recv(n_1, politoNF, p, t_1) && myFunction(p.src,p.dest))
PolitoWebClient	send(politoWebClient, n_0, p, t_0) -> nodeHasAddr(politoWebClient,p.src)
	send(politoWebClient, n_0, p, t_0) -> p.origin == politoWebClient
	send(politoWebClient, n_0, p, t_0) -> p.orig_body == p.body
	recv(n_0, politoWebClient, p, t_0) -> nodeHasAddr(politoWebClient,p.dest)
	send(politoWebClient, n_0, p, t_0) -> p.proto == HTTP_REQ
	send(politoWebClient, n_0, p, t_0) -> p.dest == ipServer
PolitoWebServer	send(politoWebServer, n_0, p, t_0) -> nodeHasAddr(politoWebServer,p.src)
	send(politoWebServer, n_0, p, t_0) -> p.origin == politoWebServer
	send(politoWebServer, n_0, p, t_0) -> p.orig_body == p.body
	recv(n_0, politoWebServer, p, t_0) -> nodeHasAddr(politoWebServer,p.dest)
	send(politoWebServer, n_0, p, t_0) -> (∃ p_1,t_1 : (t_1 < t_0 && recv(n_0, politoWebServer, p_1, t_1) && p_0.proto == HTTP_RESP && p_1.proto == HTTP_REQ && p_0.dest == p_1.src && p_0.src == p_1.dest && p_0.url == p_1.url)

Class - method	Constraints explaination
NetContext	send(n_0, n_1, p_0, t_0) -> n_0 != n_1
	recv(n_0, n_1, p_0, t_0) -> n_0 != n_1
	send(n_0, n_1, p, t_0) -> p.src != p.dest
	recv(n_0, n_1, p, t_0) -> p.src != p.dest
	recv(n_0, n_1, p, t_0) -> send(n_0, n_1, p, t_1) && t_1 < t_0
	send(n_0, n_1, p, t_0) -> p.src_port > 0 && p.dest_port < MAX_PORT
	recv(n_0, n_1, p, t_0) -> p.src_port > 0 && p.dest_port < MAX_PORT
	recv(n_0, n_1, p, t_0) -> t_0 > 0
	send(n_0, n_1, p, t_0) -> t_0 > 0
Checker - IsolationProperty	recv(n_0,destNode,p0,t_0)
	send(srcNode,n_1,p1,t_1)
	nodeHasAddr(srcNode,p1.srcAddr)
	p1.origin == srcNode
	nodeHasAddr(destNode,p1.destAddr)
	p1.origin == p0.origin
	nodeHasAddr(destNode, p0.destAddr)
Checker - IsolationFlowProperty	recv(n_0,destNode,p,t_0)
	send(srcNode,n_1,p1,t_1)
	nodeHasAddr(srcNode,p.srcAddr)
	p.origin == srcNode
	(∄ p_2, n_2, t_2 : (send(destNode,n_2,p_2,t_2) && p_2.srcAddr == p. destAddr && p_2.srcPort == p.destPort && p_2.destPort == p.srcPort && p_2.destt == p.src && t_2 < t_0)
Checker - NodeTraversalProperty	recv(n_0,destNode,p,t_0)
	send(srcNode,n_1,p1,t_1)
	nodeHasAddr(srcNode,p.srcAddr)
	p.origin == srcNode
	(∄ n_2, t_2 : (recv(n_2,node,p,t_2) && t_2 < t_0)
	(∄ n_2, t_2 : (send(node,n_2,p,t_2) && t_2 < t_0)
Checker - LinkTraversalProperty	recv(n_0,destNode,p,t_0)
	send(srcNode,n_1,p1,t_1)
	nodeHasAddr(srcNode,p.srcAddr)
	p.origin == srcNode
	∃ t_1, t_2 : send(linkNode0,linkNode1,p,t_1) && recv(linkNode0,linkNode1,p,t_2) && t_1 < t_0 && t_2 < t_0
Checker - DataIsolationPropertyCore	recv(n_0,destNode,p,t)
	p.origin == srcNode
	for each constraint(n -> constraint)
Checker - DataIsolationProperty	recv(n_0,destNode,p,t)
	p.origin == srcNode
Network -setAddrMapping	addrToNode(addr) = node
	nodeHasAddr(node, a_0) == (a_0 = addr)
Network -setAddrMappingMulti	addrToNode(foreach ad in addr) = node
	nodeHasAddr(node, a_0) == Or(foreach ad in addr (a_0 == ad))
Network - saneSend	send(node, n_0, p, t_0) -> not(nodeHasAddr(node, p.dest))
Network - setGateway	send(node, n_0, p_0, t_0) -> n_0 = gateway
Network - compositionPolicy	foreach rtAddr,rtNode in rt(send(node, n_0, p_0, t_0) && Or(foreach rtAddr in rt destAddrPredicate(p_0,rtAddr)) -> n_0 == rtNode)
Network - compositionPolicyShunt	foreach rtAddr,rtNode in rt(send(node, n_0, p_0, t_0) && Or(foreach rtAddr in rt destAddrPredicate(p_0,rtAddr)) -> n_0 == rtNode)

2.2 Part b)

To implement the JAX-WS Web Service, first of all I created a *wsdl* file to describe it.



The server part now contains the *j-verigraph* library, a "VNFChainVerificationImpl.java" class which implements the reachability check service and a "VNFChainVerificationServer.java" class to publish the service.
On the other hand the client provides the same examples of network models used before, in the following classes:

clients
WS_Polito_AntispamTest.java
WS_Polito_CacheFwTest.java
WS_Polito_CacheNatFwTest.java
WS_Polito_CacheTest.java
WS_Polito_DoubleFwTest.java
WS_Polito_SimpleTest.java

The only difference is that now the client is no more aware of the *j-verigraph* library, but it uses the stub classes generated from the *wsdl*. The structure of the input that the server needs in order to provide the service is discussed in the next section.

2.2.1 XML format

I created an xml schema to represent and validate the information that the client must provide to the server in order to make the request about the reachability of two network nodes. The schema structure represented in pseudo-code is the following:

```
<checkIsolationProperty>
    <SourceNodeName/>
    <DestinationNodeName/>
    <VNF>
        <Name/>
        <IPs/>
        <VNFType_and_configuration>
        <RoutingTable>
            <Entry>
                <IP/>
                <Name/>
            <Entry>
        </RoutingTable>
    </VNF>
    <VNF>
        <Name/>
        <IPs/>
        <VNFType_and_configuration>
        <RoutingTable>
            <Entry>
                <IP/>
                <Name/>
            <Entry>
        </RoutingTable>
    </VNF>
    ...
</checkIsolationProperty>
```

Each VNF element is a network object with its configuration, and the set of those elements represent the VNF chain. Actually the structure of the input permits not only to have VNF chains, but also more complex graphs. In fact, the source and destination elements could be every element of a network model, and every VNF has a routing table which describes the routes and somehow the topology of the network. In that way the order of the VNF in the list is not so much relevant and the start and end VNF in the list are not necessarily the nodes for whom we want to check the reachability.
Each VNF element has a Name, one or more IP addresses, an element which specifies VNF type-specific information and a routing table with its configured entries.
The source and destination elements are references to 2 specific VNF described in the chain and their purpose is to indicate which are the nodes for which we want to know the reachability(non-isolation) property. When the server reaches those information creates and configure the actual *j-verigraph* network objects needed and attaches them to create the network model.

2.2.2 input validation and error handling

I introduced a server-side validation handler which manages the input and ensures that is valid. This was done using a XML handler chain configuration file linked to the SIB class through an annotation.

As you can see in the *wsdl* figure, for the error handling part there are three types of errors defined as faults in the *wsdl*:

- **Z3Error:** this fault corresponds to the *Z3Exception*, which is thrown when there is an exception in the usage of Z3 Library
- **MalformedArgument**: this fault is used to inform the client when its input is not well formed.
- **WebServiceException**: when there is an error in server's remote operation.

The MalformedArgument exception can be seen as a redundancy since there is a "ValidationHandler" class that manages the server's input validation, but it has been introduced in order to improve the robustness of the server, that can face different types of clients.

Capitolo 3

Configuration and execution

The first thing to do to use the *j-verigraph* library and the web service is to install Z3 and put the path of "com.microsoft.z3.jar" and the libraries it refers to as environment variable, i.e. is enough to add the project subfolder "build" to the PATH environment variable.
Regarding the part a) of the project, each class of the package *tests* has a main to run the specific request of reachability with various network models. Is quite easy to compile and run one of them, as well as it's easy to create similar tests using the implemented VNF, for example creating different network models.
To run the web service there are two ant scripts called "build.xml", one for the server and one for the client.
The first is situated in "service" folder and it does the proper operations to run the server: it configures the *wsdl* file with the endpoint address of the server, then it compiles the *wsdl* and it copies wsdl, schema, and chain file to META-INF directory, and finally it runs the server.
The client's ant script is situated in 'client" folder and it imports the *wsdl* file, then it compiles the *wsdl* and finally it runs the client.
The client classes are reflecting the network models implemented in *j-verigraph*'s *tests* package, and to select one of them, the name of the chosen one can be passed as an argument to the script, preceded by the command *-Dtestclient=*.