A Modelling Notation for Grid Computing

Yih-Jiun Lee
DSSE research group
Electronics and Computer Science
University of Southampton, UK
y.lee@ecs.soton.ac.uk

Peter Henderson
DSSE research group
Electronics and Computer Science
University of Southampton, UK
p.henderson@ecs.soton.ac.uk

Abstract

Script *is a modelling language designed for Grid computing. This modelling language can be used to model and check the correctness of global-size grid-based applications. Script models the abstract of applications. It provides simple structure to define the model and basic reductions to describe the actions. It uses assertions for the initial state and conditions, and intends to achieve the final state defined as goal(s), so the user can check the correctness. A graphic based model checker is also provided for* Script *to animate, check and verify the models. In this paper, the structure of* Script *model is introduced and scenarios will be modelled to show how Script models are used.*

1. Introduction

Script is a modelling language designed for grid computing. The aim of *Script* is to model the abstract of grid applications. In addition, *Script* would like to be used for modelling how a grid-based application functions rather than how a single process performs. More specifically, a user of *Script* would like to understand and check the correctness of a grid application by modelling the work-flows which happen within the whole environment.

This paper is organized as follows. Section 2 briefs the structure of a *Script* model and the statements (assertions and reductions) used in Script. Section 3 provides several grid based applications and framework to show the capability of *Script* and how it can be used in the simple and complicated scenarios. Section 4 introduces the verification tool. Finally, the last section will conclude the Script.

2. Script Notation

Considering the following scenario, Alice is a scientist working in a virtual domain which contains five different servers. With the valid authorization, Alice has the right to access these five servers, but she usually keeps her files on server1 and the dataset is located on server2. When Alice needs to run an executable process which needs dataset as an input parameter, Alice needs to send the process to server2 as submitting a job. This job needs to be executed on her behalf, so that the process can be authorized to access the dataset on server2. It is a significant case in grid. In most situations, the tasks that a user is used to do are moving files around and executing jobs on servers.

Therefor, files in grid system can be sorted into three groups: data file, process and script. A data file is the file which only contains data and is not executable or at least not executable in the model. A process is a pre-defined and executable file which might need some data files as input(s) and might generate another file as an output. The existence of the input file(s) is the executing precondition of the process and the existence of the output file is the post-condition. A script is an executable and customize-able file in the system. The scripts are also the skeleton of the model. It defines the movement and operation of every object in the model. It is usually considered as a workflow definition.

2.1. Model Assertion

When a script model is built, the first part is a set of assertions. Three kinds of assertions might be defined: initial state assertion, precondition assertion and post-condition assertion. Firstly, an initial state instruction defines the initial state of the model, such as the location of files. For instance, file1 on server1 can be written as “assert file1 on server1” to indicate the initial location of file1.

An assertion can also be used to define the conditions of processes. It might be the necessary input files or the possible generated file. They are what is called precondition and post-condition assertions. For instance, “assert process1 reads file1” indicates that process1 can be executed only if there is file1 on the same server. It is the precondition of the execution. It is possible that more than one

file will be used during the process execution. Therefore, the "reads" assertion of one process might take place many times and the relationship between those assertions is an "AND".

Conversely, the post-condition is defined by"writes". Statement "assert process1 writes file2" defines that a new file "file2" will be generated by process1. A process might produce either file1 or file2, so the "writes" assert might also appear twice or more. But, eventually only one file can be created. This design is especially for the case of verification, so that a verifier can output the message regarding the acceptance or refusal. Hence, the relation between the "writes" assertions of one process is an "OR".

For instance, an authentication process might take a personal credential and a request as input parameters and generate one message: accept or refusal. It can be written in the following:

```
assert authentication reads credential
assert authentication reads request
assert authentication writes accept
assert authentication writes reject
```

The file location assertion is only used for the files not shown in the model. It means that all the files except scripts have to be defined in the assertions. Scripts have to be defined inside the server in the model. Other relative reductions are discussed later.

2.2. Model Final State Assertion

Final state assertions are also goals assertions, which are the appearance of file on a particular location, written as "final file1 on server1". Only one goal can be reached in a model. In the earlier authentication, the goal can be "final accept on Server1". If there are more than one goal defined in a model, only one of them can be achieved. Hence, the relation is an "OR".

2.3. Reductions

Script provides operational reductions and supplement reductions. Operational reductions are used to define the fundamental functionality of the model and supplement reductions complete the complicated situation for the users.

2.3.1 Operational Reductions

Script provides two groups of basic statements. The first one is the transportation group, which is used for describing the movement of the objects. Only one reduction in the transportation group is *push*. It takes two parameters which are a file name and a server name, such as "push file1 to server1". Considering the shared directory might be used, "push alias/file1 to server2" can be used to replace. It is used to model the movement of the objects.

The operation group commands the execution of objects. When a script file is used as a job describer or controller, another script or process might be activated/enabled by the use of reduction "exec .. on ..". This reduction is to activate a name-given process on a particular location. To specify the detail location of the process, "exec Alice/file1 on server1" can also be used. However, a process might not be able to execute straightforward. To control the execution condition, a process can only be executed if its precondition(s) can be satisfied. They are defined in the "assert.. reads.." assertions. Conversely, an output file may be generated by the process as a post-condition, defined in the assertion of "assert..writes..".

"Wait" is a temporary blocking reduction in *Script*. It can be used when a script cannot process its next statement until the previous process has finished. The evidence of the finish might be an appearance of a file. For instance, the following script s1 shows the push statement cannot be processed unless file1 appears. Without the "wait" command, file1 might be tried to send before it has been generated.

```
Script s1{
  . . .;
  wait file1;
  push file1 to server2; . . .
```

Another usage of "wait" is as a precondition of a script, such as Script s2. "File1" is seen as the precondition of script s2. Hence, in the model, s2 should be blocked until file1 shows up.

```
Script s2{
  wait file1;
  [..Block A..]
}
```

2.3.2 Supplemental Statement

supplemental statements are used for more complicated situations., including a naming method and a conditional statement.

Naming is a method for changing an object's name after its movement. It is usually used to indicate the path of the object or the creator of the object. There is an optional field in "push", called "as", such as "push file1 to server1 as input-file". It is often used to integrate the name of precondition. For example, the input file of "process1" is "p1.input", but the input file that user uses is "user.data". "push user.data to server1 as p1.input" ensures that process1 can correctly find out the data file by given it a new name. Another related naming method is a "wild word", "me". It represents the current node. For instance, "exec file1 on me" indicates the activation of file1 on the current server,

if file1 is on the server. “Me” is also the name for renaming. Suppose, the current server is Server1, and then “push file1 to Server2 as me/file1” will copy file1 from Server1 to Server2 and rename it to “Server1/file1”. This design is especially for the shared memory in Grid computing.

The conditional statement is “choice”, including internal choice and external choice. An internal choice is similar to “if .. else ..” or “if .. else if.. else if... else”. It is shown earlier that a process might generate one file from several possibilities. Hence, it is also possible that with different outputs, different processes should be invoked. This choice is made referring to the existence of a specific file in the system, called internal choice. The option in internal choice usually begins with “wait”. For instance, in the following scenario: if file1 shows up, blockA performs; if file 2 appears, blockB should execute. However, only one process block chosen is depending on which file shows up.

```
Choice {
    {wait file1; {block-A}}
    {wait file2; {block-B}}
}
```

2.4. The BNF of Script

Table 1. shows the structure of a *Script* model. The exclamation mark “!” indicates the following content is repeated. For instance, “! ServerSet” in Table 1. means ServerSet can be constructed by different repeated ServerSet. One thing to mention is all servers are at the same level of execution. So are the scripts. The concurrent execution is implicit. However, the reductions separated by “;” have to be run in sequence.

Table 1. The BNF of Script model

Container	:=	Contents
Model	:=	AssertSet Final-Assert ServerSet end
AssertSet	:=	! AssertSet | assert file on servername; | assert file reads a-file; | assert file writes a-file; | nil
Final-Assert	:=	final file on servername

ServerSet	:=	! ServerSet
		| Server servername {
		! ScriptSet
		}

ScriptSet	:=	!ScriptSet
		| Script scriptname {
		! Statement;
		}
		| nil

Statement	:=	! Statement | nil | push file to servername; | push file to servername as alias; | exec file on servername; | wait file; | ChoiceSet

ChoiceSet	:=	Choice {
		! { option }
		}

option	:=	Statement

3. Script Model

This section shows the use of *Script* in grid applications to show that *Script* is not only designed for grid applications, but also suitable for security framework of Grid.

3.1. A Workflow Submission Model

A workflow contains the information to execute a task. Moreover, a workflow can also define the information to enable other workflows. In this scenario, we would like to show a more complicated model, shown in Fig 1. The “dash” block means the block is moved following the “dash” arrow.

Supposed *w1, w2, w3* are workflows, which have to be defined as script in the model, on server *A*. There are two data files, *d1* and *d2*, on server *B*. Process *P*, which might be a service or a function provided on server *C* and server *D*, will need to consume a data file *d* and then generate a new file *r*. *Q* is another process on server *B*. The objective of Q is taking two data files, *C.r* and *D.r* and combing them into one file *r*. Finally, the goal of the model is to prove that file *r* is on server *A*.

The main portion of model is defined as follows:

```
Server A {
    Script w1 {
        push w2 to B;
        push w3 to C;
        push w3 to D;
        exec w2 on B;
    }
    Script w2 {
        push d1 to C as d;
        push d2 to D as d;
        exec w3 to C;
        exec w3 to D;
        wait C.r;
```

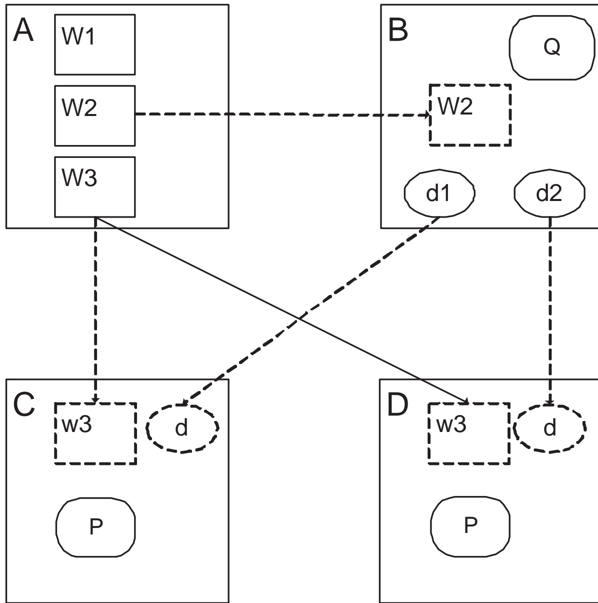

Figure 1. Initial state of Workflows submission model

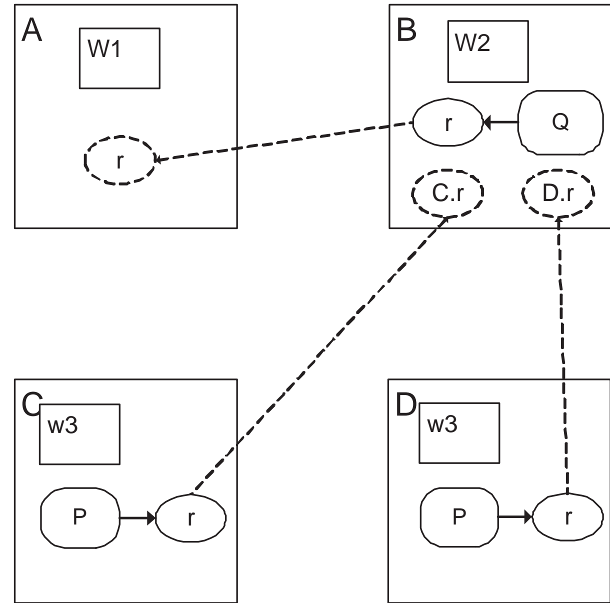

Figure 2. Final state of Workflows submission model

```
        wait D.r;
        exec Q on me;
        wait r;
        push r to A;
    }
    Script w3{
        wait d;
        exec P on me;
        wait r;
        push r to B as me.r;
    }
}
```

Because "exec Q on me" is executed after *w2* has been moved to B, "me" will represent server *B*. However, *w3* was sent to server *C* and server *D*, so the "push r to B as me.r" causes the new name of *r* will depend on which server the file source is. By this naming method, the model can distinguish the ambiguous files. The final state is shown in Fig 2. The bolder arrow shows the flow that the data files are consumed by the processes, shown in rounded rectangle, or the data files are generated by the processes.

3.2. A CAS of Globus Project Model

This section shows the *Script* model of the Community Authorization Service (CAS) [5, 2]. CAS is a virtual organization (VO) solution for grid, developed by the Globus project. The main object of CAS is providing an authorization framework.

The use of CAS the be described as follows [1]. Three roles in the system are requester who is a resource requester and a member in the virtual organization who has a CAS server enabled, a resource or a group of resources, which also belong to the VO, a CAS server which is responsible for the VO and trusted by both the resource requesters and providers. However, the resource requester and resource provider might not know each other or their access rights. In order to request a resource, the user must start from acquiring a certificate containing the allowance from the CAS server. Firstly, he passes his X.509 certificate which might be a newly generated proxy certificate to the CAS server. The CAS server verifies the received certificate and establishes the authorization and resource by referring to the local policies which might be delegated by different resources [4]. A new restricted proxy certificate with the information of the authorization, the requester and the issuer will be created by the CAS server and sent to the user. Therefor, the user can then use the new proxy certificate to access the account of the VO but his access is restricted based on the statement on the certificate.

In the model, "Token" represents the certificate as a proof of user's membership, such as the first proxy certificate in the earlier paragraph. "Request" is a request from the user to the CAS server. They are sent to the CAS server by the "main" script. CapGen is a process of the CAS server, which verify the user identities and requests to generate restricted proxy certificates (capresult) regarding local policies and user authentication information. The first two

"wait" statements (line.3 and 4) in script "CapReq" ensure the script is blocked until those two files show up. However, CapGen cannot be executed without them, because the preconditions are defined in the assertions. They can be omitted.

```
assert token on User;
assert request on User;
assert CapGen on CAS;
assert CapGen reads token;
assert CapGen reads request;
assert CapGen writes capresult;
Server User {
  Script main{
1.  push token to CAS;
2.  push request to CAS;
  }
}
Server CAS {
  Script CapReq{
3.  wait token;
4.  wait request;
5.  exec CapGen on me;
6.  wait capresult;
7.  push capresult to User
          as CAS.capresult;
   }
}
```

When "capresult" is returned to the user, it is renamed to "CAS.capresult" to represent its issuer. Then script "requestResource" can be unblocked (at line 8). It is sent to the resource along with an access request (ResourceRequest). Same to "CapReq", the wait in line 11 can be omitted. "VerifyRequest", a process on the resource, needs the certificate and request to make an authorization decision, either accept or reject. Both of them will be sent back to the user as "Resource.Reply", which is the goal to achieve.

However, two different files might be generated and which one to generate is an implicit external decision made by the operator. But the corresponding operations are chosen by the system as an internal choice, defined in the "Choice" statement at line 14. This internal choice is made according to which file (accept or reject) is generated. If "accept" shows up, it has to be returned to the user and renamed. Same happens if "reject" is generated.

```
assert ResourceRequest on User;
assert VerifyRequest on Resource;
assert VerifyRequest
     reads ResourceRequest;
assert VerifyRequest
     reads CAS.capresult;
assert VerifyRequest
     writes accept;
assert VerifyRequest
     writes reject;

final Resource.Reply on User;

Server User {
 Script requestResource {
8.  wait CAS.capresult;
9.  push CAS.capresult to Resource;
10. push ResourceRequest to Resource;
} }
Server Resource {
  Script ResReq{
11. wait CAS.capresult;
12. wait ResourceRequest;
13. exec VerifyRequest on me;
14. Choice {
      {wait accept;
       push accept to User as
          Resource.Reply;}
      {wait reject;
       push reject to User as
          Resource.Reply;}
    }
  }
}
```

4. The Verification Tool

The verification tool of *Script* provides a user friendly model checker. A user starts the tool by loading a text-based model. Once loaded, the user has to verify the layout of model by "checking", shown in the dashed ellipse in Figure 3. It is because a text-format model is difficult to be correct formatted. "Checking" the format earlier can reduce the cost of error generation. If an error has been found, an error log will be generated. Otherwise, the user clicks "initial", under "check" to generate the initial state.

The column-"available choices" is a list of current available activities. Because the servers and scripts are designed to run concurrently, the sequence of the execution is artificially decided. When an activity is chosen from the left column, the system moves to the next state according to the execution of the activity, which is also appended at the end of the "history" column on the right side. In the meantime, the previous state is also "pushed" into a state for stepping back in case.

Retreating to a previous state can be performed by the operation of "Undo". It is designed to find out the existence of deadlock caused by execution in different sequences. A successful Script model should only have one activities available at a time if two activities are relevant. However, if

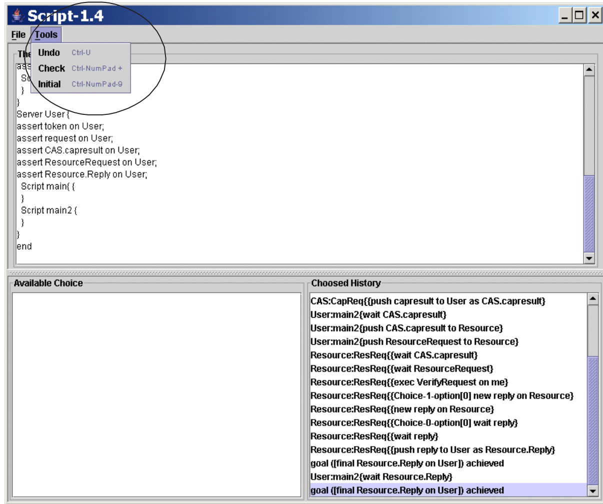

Figure 3. Verification tools:load a model

two or more activities are enabled at the same time, the user should try different sequences of execution in order to find out the potential error(s). A correct model should end with an empty "available choices" column and a line showing "goal achieved.." at the end of "history" column. If either is missing, the model is incorrect.

5. Future Plan and Conclusions

The future focus of Script is considering *Script* as a workflow language. Script-i [3] is a web-services based grid middleware having similar syntax of *Script*.Hence, the user can easily transform a script in a model into a Script-i program manually.

Meanwhile,as other modelling languages, *Script* models can be verified by its verification tool. It provides the functionalities of checking layout, initiating the model, and executing the model step by step. It is also possible to step back the state of execution if an error is found or different combination of executions has to be examined. Thus, the correctness of models can be verified and provide a better usage of Script.

Script is a modelling language for grid computing. It is designed to model grid applications and related security frameworks. Firstly, by using the simple reductions, a *Script* model can successfully simulate the actions happening in the system. A *Script* model is easily composed of assertions, servers and scripts. The preconditions and post conditions are clear and understandable. Secondly, *Script* is also suitable to model the security solutions of grid computing. This is particular important because modelling security model is often difficult to get right. Finally, Script has proved that it can be used to model different grid computing applications and the models are legible and tiny. It is a good choice for a new grid user getting to know the grid operations.

References

[1] S. Canon, S. Chan, D. Olson, C. Tull, and V. Welch. Using cas to manage role-based vo sub-groups. In *Computing in High Energy and Nuclear Physics, CHEP03*, CA, USA, Apr. 2003.

[2] I. Foster, C. Kesselman, L. Pearlman, S. Tuecke, and V. Welch. The community authorization service: Status and future. In *Computing in High Energy and Nuclear Physics, CHEP03*, CA, USA, Apr. 2003.

[3] Y. Lee. A security solution for web-services based virtual organizations. In *Proceedings of the Information Resources Management Association International Conference (IRMA 2005): Managing Modern Organizations with Information Technology*, San Diego, USA, May 2005.

[4] M. Lorch, B. Cowles, R. Baker, and L. Gommans. Authorizaiton framework document, Mar. 2004.

[5] L. Pearlman, C. Kesselman, V. Welch, I. Foster, and S. Tuecke. The community authorization service: Status and future. In *CHEP03, Computing in High Energy and Nuclear Physics, 2003 Conference Proceedings*, San Diego, CA, USA, Mar. 2003.