
Project DMC Documentation

Release 3.0

Project DMC

July 16, 2016

1	Topics	3
1.1	Project DMC	3
1.2	DOME	28
2	Contribute	129

Note: This documentation is a work in progress. Topics marked with a TODO stub-icon are placeholders that have not been written yet. You can track the status of these topics through our public documentation [issue tracker](#).

Welcome to the DMC Developer Documentation. Users looking for instructions should refer to the Project DMC Wiki instead.

1.1 Project DMC

1.1.1 Introduction

This section contains information about the Digital Manufacturing Commons web platform.

1.1.2 Topics

Deployment

First, create a development machine either from your local machine or from a seed AWS image. Then choose to deploy a stack either from the DMC repos or from your own code base.

Topics

Deployment Process First, create a development machine either from your local machine or from a seed AWS image. Then choose to deploy a stack either from the DMC repos or from your own code base.

Topics

Creating a Development Machine from a Local Machine

- *Clone the DMC dev repo*
- *Set Up AWS keys*
- *Edit terraform.tfvars*
- *Launch dev machine*
- *Launch dev instance*
- *SSH into dev instance*
- *Optional: switch to a better shell*

Clone the DMC dev repo

```
$ git clone https://bitbucket.org/DigitalMfgCommons/dmcdeploy_dev.git
```

Go to the sub folder:

```
$ cd dmcdeploy_dev
```

Switch the branch to devmakina:

```
$ git checkout devmakina
```

Set Up AWS keys Make the keys directory if not there:

```
$ mkdir ~/keys
```

Create and download the AWS .pem key files you wish to use for your deployments:

- Key for the seed machine
- Key for dev machine
- Key for any other machines dev may launch

In the AWS EC2 console, under keypairs, create as many key pairs as needed and download them locally. By convention they are stored in ~/keys

If you are deploying from an AWS seed machine then you must upload to this seed machine the keys that you would like it to use when deploying subsequent machines.

Upload a .pem file to the seed machine. From your local machine (the machine you use to connect to AWS, not the AWS seed or dev machines):

```
$ scp -i seed.machine.key.pem remotekey.pem ec2-user@ip.of.seed.machine:/home/ec2-user/keys
```

(Note: “seedmachinekey.pem” is the name of the key from creating the instance in AWS and “remotekey” is a name you choose for the file and will use in the terraform.tfvars file below.)

Confirm that the remote .pem is on your dev machine by finding its name listed with the following command entered in the seed machine:

```
ls ~/keys
```

Edit terraform.tfvars

```
$ cd dmcdeploy_dev/  
$ nano terraform.tfvars
```

- Input your aws credentials on the first two lines
- Edit key_name to the name of the remote key you just uploaded (without the .pem extension)
- Edit the key_full_path to "~/keys/remote_key_name.pem"

Note: Make sure that the keys you have entered match the region where you wish to deploy to. Keys are only valid in the region they were created.

- Edit the stackPrefix
- **To launch the alpha release:**
 - change release = “alpha”
- **To launch latest nightly:**
 - change release = “hot”

- Edit the stackPrefix to something meaningful; this will be the prefix label for your new instances in AWS
- Edit the aws_region ('us-east-1' or 'us-west-2') as appropriate

Launch dev machine

- Uncomment the dev.tf file if needed by removing the comment characters on the first and last line (nano dev.tf)
- Comment out any other machines that may be there

Install Terraform if you have not already done so:

```
$ cd /tmp
$ wget https://releases.hashicorp.com/terraform/0.6.8/terraform_0.6.8_linux_amd64.zip
$ mkdir ~/tf
$ unzip terraform_0.6.8_linux_amd64.zip -d ~/tf
```

Make sure the path variables are properly set:

```
$ echo "export PATH=$PATH:/home/ec2-user/tf" >> ~/.bashrc
```

Reload your bashrc:

```
$ source ~/.bashrc
```

Verify install with:

```
$ terraform -v
```

Launch dev instance To create the plan in a file called “a” and check you are launching the infrastructure you think you are:

```
$ terraform plan -out a
```

To launch that infrastructure from the plan created in the previous step:

```
$ terraform apply a
```

Note: If something goes wrong, execute `$ terraform destroy`. Then fix your configuration or terraform.tfvars file before repeating the terraform plan and terraform apply steps described above.

Note: At this point your local machine or the Seed Machine will have deployed a Dev Machine on AWS. You will then use the Dev Machine to do further work.

SSH into dev instance To find out what the IP address of the dev machine is:

```
$ terraform show
```

Read it from the line

```
aws_instance.dev:
...
...
public_ip = 52.23.214.212
...
...
```

Then:

```
$ cd ~/keys
$ chmod 400 <keyname>.pem      # in case amazon complains that key restrictions are too open
$ ssh -i "key_used_in_terraformvars.pem" ec2-user@ip.of.dev.machine
```

Accept the key fingerprint:

```
$ yes
```

Note: At this point you are logged into the Dev machine you have just created. The seed machine/ local laptop is no longer needed.

To speed up your development you may now want to create a image of the dev machine in whatever region you want and launch that and use it for development thereby not needing to repeat many steps listed above.

In the future a dev machine may be simply running and multiple developers will only need to ssh into it and not recreate it.

As part of the security infrastructure developers interact with the DMC from the Dev Machines.

We want to be transparent with the tools and technology we use which is why we document and expose our processes instead of simply handing off opaque development environments.

The reason for using a seed machine is that this machine controls the setup of the dev machines. As the DMC is a rapidly evolving project the dev machines will need to be updated periodically. The seed machines can be used to give developers a personalized dev machine without those developers caring for/about the dev ops that created their instances.

Optional: switch to a better shell

```
$ zsh
$ export PATH=$PATH:/home/ec2-user/tf
$ alias tf='terraform'
```

Creating a Development Machine from a Seed Machine

- *Option 1: Use a prebuilt seed image*
- *Option 2: Create your own seed machine*

Option 1: Use a prebuilt seed image Go to the EC2 Dashboard and launch a custom AMI image using the steps below.

For west:

West_Seed_1203: ami-45a6b424 << linux machines

West_Seed_w_1203: coming soon << windows machines

For east:

East_Seed_1203: ami-34f3be5e << linux machine

East_Seed_w_1203: coming soon << windows machines

Steps to launch from custom image:

- EC2 Console > click launch instance.
- My AMI's > search for appropriate machine.
- Resize as appropriate (M4.large).
- Tag your instance with a memorable name.
- Choose the key-pair you wish and launch.

This instance will have the same configuration that you would get in Option 2 below.

Set up your private key on the machine from which you will access the Amazon instance. For PC users, PuTTY is a good tool to access the Amazon instance. If you are on a corporate network behind a firewall, consult with local administrators to determine how to access AWS instances. Configure PuTTY to use your private key by specifying the file in the PuTTY Configuration under Connection -> SSH -> Auth -> Private key file for authentication.

Log in to the new instance with ssh (putty). The username is ec2-user.

After logging in you can verify the configuration with:

```
$ terraform -v
$ git --version
```

Update your machine:

```
$ sudo yum update -y
```

You can now follow the same instructions described in Creating a Development Machine from a Local Machine to create your personal dev machine that you will use to launch your stack or further develop.

Option 2: Create your own seed machine

1. Create an AWS Linux machine following the AWS instructions. This machine can be in any region you want and of any size you choose.
2. SSH into the created machine.
3. Install the needed tools:

```
$ sudo yum install -y git
```

Install Terraform:

```
$ cd /tmp
$ wget https://releases.hashicorp.com/terraform/0.6.8/terraform_0.6.8_linux_amd64.zip
$ mkdir ~/tf
$ unzip terraform_0.6.8_linux_amd64.zip -d ~/tf
```

4. Make sure the path variables are properly set:

```
$ echo "export PATH=$PATH:/home/ec2-user/tf" >> ~/.bashrc
```

5. Reload your bashrc:

```
$ source ~/.bashrc
```

6. Verify install with:

```
$ terraform -v
```

You can now follow the same instructions described in [Creating a Development Machine from a Local Machine](#) to create your personal dev machine that you will use to launch your stack or further develop.

Deploying the Stack from the DMC Repositories The stack that this process deploys reflects the data found in the Digital Manufacturing repositories found at: <https://bitbucket.org/DigitalMfgCommons/>

The purpose of this is to demonstrate the deployment of the stack from our stable repositories. It focuses on only the infrastructure deployment segment of the software cycle. This process is not applicable if you wish to alter the code as it will ALWAYS deploy from the DMC repositories. If you wish to make code changes and see them reflected in your deployment, see [Deploying the Stack from a Local Code Base](#) instead.

1. Pick dev or deploy branch (here we choose dev to demonstrate):

```
$ cd dmcdeploy_dev/
```

We deploy just the front end to prove it works.

2. Configure:

```
$ nano front.tf
```

- remove the first and last line comments
- remove the dependency line: `depends_on = ["aws_instance.rest"]`
- remove the setting of rest environment variable: `echo 'export Restip=${aws_instance.rest.private_ip}' >> ~/.bashrc`

3. Edit terraform.tfvars

- Input your aws credentials on the first two lines
- Upload a key you wish to deploy with to this dev machine

```
$ scp -i mykey.pem somefile.txt ec2-user@my.ec2.id.amazonaws.com:/home/ec2-user/keys
```

- Edit key_name to the name of the remote key you just uploaded (no .pem)
- Edit the key_full_path to " ~/keys/key_name.pem"
- Edit the stackPrefix
- **To launch the alpha release:**
 - change release = "alpha"
- **To launch latest nightly:**
 - change release = "hot"

4. Create the stack from the official repos of the DMC

```
$ terraform plan -out a
```

Confirm that terraform will create the infrastructure that you want, then run:

```
$ terraform apply a
```

5. To prove that you successfully launched the front end process, run:

```
$ terraform show
```

From the listing, determine the public IP of the `aws_instance.front` process. Open that IP in any browser. You should see the DMC user interface.

6. To destroy the infrastructure:

```
$ terraform destroy
```

Deploying the Stack from a Local Code Base Create a Dev Machine that allows you to edit the code and deploy your changes.

This process demonstrates mutable infrastructure. The purpose of this method is to show how one can deploy and develop for the DMC.

The production infrastructure builds are automated using Bamboo and integrated into a code delivery pipeline. We will expose more of our build and deployment tools in the near future.

This demonstrates how one may deploy manually and create infrastructure.

1. Go into the `dmcdeploy_dev` folder:

```
$ cd ~/dmcdeploy_dev
```

2. Switch branches to the `devmakina` branch:

```
$ git checkout devmakina
```

3. Pull in dev repos as needed (here we demonstrate the front end repo):

```
$ cd ~
$ git clone https://bitbucket.org/DigitalMfgCommons/frontend.git
```

4. Make your code changes

We will change the website footer to have a "):"

```
$ cd /home/ec2-user/frontend/app/templates/common/footer
$ nano footer-tpl.html
```

- Change a Lorem Ipsum line to :)
- Save and exit

5. Rebuild the app

Install the build tools on your dev system:

```
$ cd ~/dmcdeploy_dev
$ nano build_tools.sh
```

- Each infrastructure component has a different build process
- Edit the `build_tools.sh` to only install the tools you need to save time and space

```
$ ./build_tools.sh
```

FRONT END BUILD PROCESS

In the FrontEnd repository, run the commands below.

```
cd ~/frontend
```

INITIAL TOOLING INSTALL (only to be done once)

```
$ npm install
$ bower install      # answer 'n' to question when installing
```

Make your code changes, if you haven't done so already.

TO BUILD

```
$ gulp build
```

The output of this process can be found in the `dist` folder

Commit your changes to the frontend locally:

```
$ cd ~/frontend
```

Forcibly add the `dist` folder to the commit

```
$ git add dist -f
$ git add -A
$ git commit -m "your message"
```

6. Deploy Infrastructure that reflects your changes

```
$ cd ~/dmcdeploy_dev
```

Will only deploy front end to prove it works

```
$ nano front.tf
```

- Remove the first and last line comments
- Remove the dependency line
- Remove the setting of `rest env val`

7. Start the git daemon on the dev machine

```
$ cd ~
$ git daemon --base-path=. --export-all --reuseaddr --informative-errors --verbose &
```

You should see “Ready to rumble” if all went well

8. Edit `terraform.tfvars`

```
$ cd dmcdeploy_dev/
$ nano terraform.tfvars
```

- Input your AWS credentials on the first two lines
- Upload a key you wish to deploy with to this dev machine

```
(quick shortcut env | grep MYip -- will show you your public ip )
```

From the machine that has the keyfile you wish to use:

```
$ scp -i mykey.pem somefile.txt ec2-user@my.ec2.id.amazonaws.com:/home/ec2-user/keys
```

Verify the key is there

```
$ ls ~/keys/en
```

- Edit key_name to the name of the remote key you just uploaded (without the .pem extension)
- Edit the key_full_path to " ~/keys/key_name.pem"
- Edit the stackPrefix
- **To launch the alpha release**
 - edit release = "alpha"
- **To launch latest nightly**
 - edit release = "hot"

9. Deploy new infrastructure

Comment out (Add /* to first line and */ to last line) dev.tf if needed as will not be deploying dev machine

```
$ nano dev.tf
```

Run 'terraform plan' and confirm that you are actually deploying the infrastructure that you intend:

```
$ terraform plan
```

To actually deploy the infrastructure, run:

```
$ ./deployFromDev.sh
```

10. To prove that you successfully launched the front end process, run:

```
$ terraform show
```

From the listing, determine the public IP of the aws_instance.front process. Open that IP in any browser. You should see the DMC user interface.

11. To destroy the infrastructure

```
$ terraform destroy
```

Once you have deployed your infrastructure further changes to your stack need only be built and committed to your dev machine. To deploy those changes you ssh into the stack and execute updateMe.sh. This will update the code-base without having to relaunch the infrastructure.

Installing Terraform You will need to install Terraform on your seed machine in order to run through the new deployment process.

Graphical User Interface Go to <https://terraform.io/downloads.html> and select the package corresponding to your OS.

Command Line

```
$ wget https://releases.hashicorp.com/terraform/0.6.6/terraform_0.6.6_linux_amd64.zip
$ unzip terraform_0.6.6_linux_amd64.zip -d whatever.directory.you.wish
```

Add the directory where the unzipped contents are located to your PATH variable.

- In Unix-based systems, this can be edited by opening ~/.bashrc using your favorite editor, and adding the line PATH=\$PATH:<filepath>.

- In Windows, this is done by opening up Windows Explorer, right clicking on This PC, and selecting Properties -> Advanced System Settings -> Environment Variables. Select PATH and choose edit. Append the directory to the end of the path variable WITHOUT spaces.

Reload or open a new terminal so the update to your PATH is recognized.

Last, verify your install was successful:

```
$ terraform -v
```

Setting Up Amazon Credentials To create a PEM file to login into your Amazon servers, please follow the steps in the following page: <http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-key-pairs.html#having-ec2-create-your-key-pair>

To create your public and private access keys:

- Visit the following page: <https://console.aws.amazon.com/iam/home?#users>
- Find your Amazon login username. Select it and under the User Actions Dropdown at the top of the users list, select Manage Access Keys. Here you can create your access keys, both public and private.

Warning: You can always visit the above link to get your public key, but this is the ONLY time you will be able to view your private key, so save it somewhere secure.

You will need these for several processes.

Please refer to the following link on Amazon access key management: http://docs.aws.amazon.com/IAM/latest/UserGuide/id_credentials_access-keys.html

Stack Machines

Topics

DMC ActiveMQ

- *Purpose*
- *Access & Testing*
- *Machine Type*
- *Depends On*
- *Dependents*
- *Ingress Rules*
- *Egress Rules*
- *Ingress / Egress Testing Status*
- *Required Vars*

Purpose Holds the results of DOME execution until the DMCFrontEnd is ready to receive/display them.

Access & Testing To access the activeMq manager visit `ip.of.machine:8161/admin/`

To test that activeMq is running:

```
netstat -nl|grep 61616"
```


Machine Type

Machine Type m4.large

AMI base image

east

west

Installed software:

- Apache ActiveMQ ver. 5.10

Depends On No other instances.

Dependents

- ForgeFrontEnd
- DOMESingleServer

Ingress Rules Connections are from DOMEServer and ForgeFrontEnd

- Port 61616, TCP and UDP – port on which ActiveMQ listens on for DOME input
- Port 61613, TCP, for connecting to STOMP.
- Port 8161, TCP, for web access.
- Port 22, TCP, for SSH, Bastion only.
- Port 8, TCP for ???

Egress Rules

- Assume same for now.

Ingress / Egress Testing Status

- Not tested.

Required Vars

- activeMqRoot
- activeMqRootPass
- activeMqUser
- activeMqUserPass

These variables are set in a configuration file called `jetty-realm.properties`.

DMC DB

- *Purpose*
- *Postgres*
 - *Machine Type*
 - *Depends On*
 - *Dependents*
 - *Ingress Rules*
 - *Egress Rules*
 - *Ingress / Egress Testing Status*
 - *Required Vars*
 - *Solr Search Integration*
- *Oracle*

Purpose Maintains information about the system in a Postgres/Oracle DB.

Postgres To access the database:

1. Switch to root after logging into the DB server, then switch to the postgres user using `su postgres`.
2. Use the command `psql` to enter the Postgres command line. From here, you can navigate into the gforge database and view the database.
3. Alternatively, enter the REST machine via ssh, and enter the command `$ curl http://localhost:8080/rest/<mapping>`. This should return a JSON response if functioning.
4. If it does not, ensure the WAR file was unpackaged in `/var/lib/tomcat7/webapps` and that Tomcat is running. Otherwise, inspect `catalina.out` for errors.

To check if postgres is running:

```
netstat -ln | grep 5432
```

(This assumes you have configured postgres to use the default port of 5432)

Machine Type

Machine Type m4.xlarge

AMI base image

east

west

Installed software:

- PostgreSQL 9.1

Depends On No other instances.

Dependents

Note: These are based on a CloudFormation template. Some of these dependencies may no longer be true.

- SolrServer

- REST Services
- Shibboleth
- Kerberos

Ingress Rules

- Open port 5432 to SOLR, Rest Services, Kerberos/Shibboleth. (This also used to be needed for ForgeFrontEnd. Perhaps that is no longer the case, because the new deployment does not have a direct connection between the DB and the FrontEnd.)
- Open port 22 for SSH, Bastion only.

Egress Rules

- Open port 5432 to SOLR, Rest Service, Kerberos/Shibboleth.
- Ports 8000-8100 seem to be needed (perhaps specifically 8069).
- Port 22 for SSH, Bastion only.

Ingress / Egress Testing Status

- Tested by setting ports manually.
- Ingress rules tested using CloudFormation.

Required Vars

- Rest Server public IP
- Solr public IP Use Solr Server Private IP by adding entry to `/var/lib/pgsql/9.1/data/pg_hba.conf`
- Example line added to `pg_hba.conf`

```
host all gforge 172.31.34.148/32 trust
```

- kerberos/shibboleth public IP
- userPass
- adminPass
- DBPort (optional, otherwise use 5432)

Solr Search Integration To enable Solr to pull data from the database configure the database to allow the SolR hshot to connect to the database.

For postgres: Add Solr Server Private IP to `/var/lib/pgsql/9.1/data/pg_hba.conf`

Example line added to `pg_hba.conf`

```
host all gforge 172.31.34.148/32 trust
```

Oracle Oracle support is still in development. Please check back later for updated information.

DMC DOME	<ul style="list-style-type: none">• <i>Purpose</i>• <i>Access & Testing</i>• <i>Machine Type</i>• <i>Depends On</i>• <i>Dependents</i>• <i>Ingress Rules</i>• <i>Egress Rules</i>• <i>Ingress / Egress Testing Status</i>• <i>Required Environment Variables</i>
-----------------	--

Purpose The DOME Server allows for simulation execution.

Access & Testing To test go to `ipofdomeserver:8080/DOMEApiServicesV7/`

You should see a web page with the following list:

- **Folders:**

- **Fracture-Mechanics**

- * Alpha
 - * AppliedLoad
 - * BetaFactor
 - * BEVXOverP
 - * CrackLength
 - * DeltaK
 - * Kmax
 - * NormalCompliance
 - * SigXCrack
 - * SimulatedLoad
 - * Ux
 - * UxCompliance
 - * X0OverWp2p8
 - * X0OverWp8p1

- File-Utilities
 - Mathematics
 - Physics

Machine Type

Machine Type m4.large

AMI base image

east

west

Installed software:

- Dome Server ver. 7
- Tomcat Server ver. 7
- Java – OpenJDK Runtime Build 1.8.0_65 b17, OpenJDK 64-bit Server VM build 25.65-b01, mixed mode
- Git ver. 2.4.3 (Unix)

Depends On

- ActiveMQServer

Dependents

- ForgeFrontEnd

Ingress Rules Connections are from ActiveMQServer and ForgeFrontEnd

- Port 8080, TCP.
- Port 7795, TCP.
- Port 22, TCP, for SSH, Bastion only.
- Open Ports– 61616 to talk to DMCActiveMq ??
- Open Ports – 80 to receive from DMCFrontEnd ??

Egress Rules

- Assume same for now.

Ingress / Egress Testing Status

- Not tested.

Required Environment Variables

- ActiveMqDns

DMC Front End

- *Purpose*
- *Access & Testing*
 - *Automatic Deployment*
 - *Manual Deployment*
- *Machine Type*
- *Depends On*
- *Dependents*
- *Ingress Rules*
- *Egress Rules*
- *Ingress / Egress Testing Status*
- *Required Environment Variables*

Purpose Present the DMC customer facing website to users.

Access & Testing

Automatic Deployment

Code repository:	https://bitbucket.org/DigitalMfgCommons/frontend.git
Build process:	Compile (see instructions below)
Build tools:	Bamboo
Built artifacts:	
Tests:	
Test tools:	Bamboo
Hosted at:	AWS S3 bucket
Deployed by:	Terraform
Deployed at:	AWS

Manual Deployment

Code repository:	https://bitbucket.org/DigitalMfgCommons/frontend.git
Build process:	Compile (see instructions below)
Build tools:	gulp build
Built artifacts:	/dist
Tests:	
Test tools:	
Hosted at:	/dist
Deployed by:	Terraform
Deployed at:	AWS

To compile:

First, install nodejs, npm, gulp, and bower:

```
$ yum install nodejs npm --enablerepo=epel
$ npm install -g gulp bower
```

Then, in the FrontEnd repository, run:

```
$ npm install
$ bower install
$ gulp build
$ cp -r dist/* <WEBROOT>/www/.
```

Machine Type

Machine Type m4.large

AMI base image

east

west

Installed software:

- Apache2 ver. ?
- php5 ver. ?
- git ver. ?

Depends On

- GITServer
- SolrServer
- DBBackEnd (perhaps not any longer)
- DMCRestServices
- Shibboleth
- Kerberos
- ActiveMQServer
- DMCDOMEServer

Dependents None.

Ingress Rules

- Port 443, tcp, GIT
- Port 80, tcp, Shibboleth (and the Internet)
- Port 22, tcp, SSH, Bastion only.

Egress Rules

- Assume same for now.

Ingress / Egress Testing Status

- Not tested.

Required Environment Variables

- DomeServerDns
- ActiveMqDns
- RestServiceDns

DMC Git

- *Purpose*
- *Access & Testing*
- *Machine Type*
- *Depends On*
- *Dependents*
- *Ingress Rules*
- *Egress Rules*
- *Ingress / Egress Testing Status*
- *Required Environment Variables*

Purpose Git server available for DMC usage.

Access & Testing Send git clone the demo repo found on the git server using

```
ssh-agent bash -c 'ssh-add keytosshintogitserver.pem; git clone ec2-user@ipofgitserver:/tmp/test/my-p
```

Machine Type

Machine Type m4.large

AMI base image

east ami-12663b7a – this is RHEL 7 (64)

west

Installed software:

- Git ver. latest (1.9.1)

Depends On None.

Dependents

- DMC Front End

Ingress Rules

- Port 9418, tcp
- Port 443, tcp for ForgeFrontEnd
- Port 22, tcp, SSH, Bastion only.

Egress Rules

- Assume same for now.

Ingress / Egress Testing Status

- Not tested.

Required Environment Variables None.

DMC Kerberos

- *Purpose*
- *Access & Testing*
- *Machine Type*
- *Depends On*
- *Dependents*
- *Ingress Rules*
- *Egress Rules*
- *Ingress / Egress Testing Status*
- *Required Environment Variables*

Purpose

Access & Testing

Machine Type

- Machine Type
- AMI base image
- east
- west

Installed software:

- Git ver. latest (1.9.1)

Depends On None.

Dependents

- Shibboleth
- ForgeFrontEnd

Ingress Rules

- Port 9418, tcp
- Port 443, tcp for ForgeFrontEnd
- Port 22, tcp, SSH, Bastion only.

Egress Rules

- Assume same for now.

Ingress / Egress Testing Status

- Not tested.

Required Environment Variables None.

DMC Load Balancer

- *Purpose*
- *Access & Testing*
- *Machine Type*
- *Required Configuration*
- *Connected To*
- *Ingress Rules*
- *Egress Rules*
- *Ingress / Egress Testing Status*

Purpose Load balance requests to the DMCFrontEnd.

Access & Testing Open the Load Balancer DNS location and it should point you to the front page.

Machine Type

Machine Type AWS Load Balancer

Installed software:

- Git ver. latest (1.9.1)

Required Configuration

- DMCFrontEnd machines

Connected To

- DMC Front End

Ingress Rules

Egress Rules

Ingress / Egress Testing Status

DMC REST Services

- *Purpose*
- *Access & Testing*
- *Machine Type*
- *Connected to*
- *Required Environment Variables*
- *Ingress Rules*
- *Egress Rules*
- *Ingress / Egress Testing Status*

Purpose Present the DMC Front End a REST interface to connect to the DMC DB.

Access & Testing To test, open `ipofrest:8080/rest/services/test` and you should receive the following JSON:

```
{
  "id":111,
  "title":"Test title",
  "description":"Test Description",
  "owner":null,
  "releaseDate":null,
  "tags":null,
  "specifications":"/services/111/specifications",
  "featureImage":null,
}
```

```

"currentStatus":null,
"serviceType":null
}

```

Machine Type

Machine Type m4.large

AMI base image

east ami-12663b7a – this is RHEL 7 (64)

east ami-18bac672 with Tomcat, Java, and Git

west

Installed software:

- Java ver. 1.8 – openJDK runtime 1.8.0_65-b17, 64-bit server VM build 25.65_b01 mixed mode
- Tomcat ver. 7
- Git ver. latest (2.4.3)

Connected to

- DMCFrontEnd
- DMCDomeServer

Required Environment Variables

- DomeDBDns

Ingress Rules

Egress Rules

Ingress / Egress Testing Status

DMC Shibboleth

- *Purpose*
- *Access & Testing*
- *Machine Type*
- *Depends On*
- *Dependents*
- *Ingress Rules*
- *Egress Rules*
- *Ingress / Egress Testing Status*

Purpose

Access & Testing

Machine Type

Depends On

- DBBackEnd
- Kerberos

Dependents

- ForgeFrontEnd

Ingress Rules

- Port 8443, tcp
- Port 80, tcp, ForgeFrontEnd
- Port 22, tcp, SSH, Bastion only.

Egress Rules

- Assume same for now.

Ingress / Egress Testing Status

- Not tested.

DMC Solr

- *Purpose*
- *Access & Testing*
- *Machine Type*
- *Depends On*
- *Dependents*
- *Ingress Rules*
- *Egress Rules*
- *Ingress / Egress Testing Status*
- *Required Environment Variables*
- *Database Integration*

Purpose Provides search for the DMC.

Access & Testing To verify that the server is running correctly, follow these steps:

- Verify the administrative web interface is up at <http://54.173.29.102:8983/solr/#/> (Substitute the IP address of your Solr instance instead of 54.173.29.102)
- Go to http://54.173.29.102:8983/solr/#/gforge_components/query
- Replace the q input with the following and press execute query: component_name:Physics
- Alternatively, copy this string into a browser: http://54.173.29.102:8983/solr/gforge_components/select?q=component_name:Physics

Output:

```
{
  "responseHeader":{
    "status":0,
    "QTime":7,
    "params":{
      "q":"component_name:Physics",
      "indent":"true",
      "wt":"json"}},
  "response":{"numFound":1,"start":0,"docs":[
    {
      "group_id":"6",
      "component_name":"Physics",
      "id":"2",
      "group_name":"Low Heat Loss Transformer",
      "unix_group_name":"lowheatlosstran",
      "is_public":false,
      "_version_":1518308270913093632}}
  ]}
}
```

- Go to http://54.173.29.102:8983/solr/#/gforge_projects/query
- Replace the q input with the following and press execute query: `%3A`
- Alternatively, copy this string into a browser: http://54.173.29.102:8983/solr/gforge_projects/select?q=%3A&wt=json&indent=true

Output: (no projects defined in database, so no results)

```
{
  "responseHeader":{
    "status":0,
    "QTime":0,
    "params":{
      "q":"*:*",
      "indent":"true",
      "wt":"json"}},
  "response":{"numFound":0,"start":0,"docs":[]
  }
}
```

- Go to http://54.173.29.102:8983/solr/#/gforge_services/query
- Replace the q input with the following and press execute query: `group_name:Low*`
- Alternatively, copy this string into a browser: http://54.173.29.102:8983/solr/gforge_services/select?q=group_name:Low*&wt=json

Output:

```
{
  "responseHeader":{
    "status":0,
    "QTime":11,
    "params":{
      "q":"group_name:Low*",
      "indent":"true",
      "wt":"json"}},
  "response":{"numFound":1,"start":0,"docs":[
    {
      "interface_data":{"interFace":{"version":1,"modelId":"995f865e-d90a-1004-8438-64281c"},
      "interface_name":"velocity interface",
      "id":"1",
      "server_id":"1",
      "cem_id":"2",
      "group_id":"6",
      "component_name":"Physics",
      "id":"2",
      "group_name":"Low Heat Loss Transformer",
      "unix_group_name":"lowheatlosstran",
      "is_public":false,
      "_version_":1518308270913093632}}
  ]}
}
```

```
"user_id": "102",
"server_url": "http://ec2-52-88-73-23.us-west-2.compute.amazonaws.com:8080/DOMEApiServicesV7/",
"group_id": "6",
"group_name": "Low Heat Loss Transformer",
"unix_group_name": "lowheatlosstran",
"is_public": false,
"_version_": 1518308270970765312}]
}}
```

- Go to http://54.173.29.102:8983/solr/#/gforge_users/query
- Replace the q input with the following and press execute query: user_name:berlier
- Alternatively, copy this string into a browser: http://54.173.29.102:8983/solr/gforge_users/select?q=user_name:berlier&wt=json&

Output:

```
{
  "responseHeader": {
    "status": 0,
    "QTime": 1,
    "params": {
      "q": "user_name:berlier",
      "indent": "true",
      "wt": "json"
    }
  },
  "response": {
    "numFound": 1, "start": 0, "docs": [
      {
        "user_name": "berlier",
        "id": "103",
        "realname": "berlier test",
        "_version_": 1518308898953494528
      }
    ]
  }
}
```

- Go to http://54.173.29.102:8983/solr/#/gforge_wiki/query
- Replace the q input with the following and press execute query: %3A
- Alternatively, copy this string into a browser: http://54.173.29.102:8983/solr/gforge_wikiselect?q=%3A&wt=json&indent=true

Output: (no wiki pages in database, so no results)

```
{
  "responseHeader": {
    "status": 0,
    "QTime": 0,
    "params": {
      "q": "*:*",
      "indent": "true",
      "wt": "json"
    }
  },
  "response": {
    "numFound": 0, "start": 0, "docs": []
  }
}
```

Machine Type

Machine Type m4.large

AMI base image

east

west

Installed software:

Solr software installed: 5.3.1.

The deployment script for Solr leverages the installation script provided by the Solr distribution. For a good description of the Solr installation process see <https://cwiki.apache.org/confluence/display/solr/Taking+Solr+to+Production>

The Solr script performs the following:

- Creates a user 'solr'
- Installs the Solr software in /opt/solr
- Configures Solr data (home) directory to: /var/solr/data
- Installs a startup script

The deployment script performs these steps:

- Downloads gforge Solr configuration files from <https://bitbucket.org/DigitalMfgCommons/dmcsolr.git>
- Downloads Solr distribution from <http://archive.apache.org/dist/lucene/solr/5.3.1/solr-5.3.1.tgz>
- Initiates the Solr installation script
- Unpacks the gforge Solr configuration files and copies them to /var/solr/data
- Restarts Solr

Depends On

- DBBackend
- DMCDomeServer (?)

Dependents

- DMCFrontEnd

Ingress Rules

- Port 8983, tcp.
- Port 22 for open SSH, Bastion only.

Egress Rules

- Assume same for now.

Ingress / Egress Testing Status Tested setting ports manually.

Ingress successfully tested using CloudFormation.

Required Environment Variables

- solrDbDns = DomeDbDns

Database Integration To enable Solr to pull data from the database configure the database to allow the Solr host to connect to the database.

For postgres: Add Solr Server Private IP to `/var/lib/pgsql/9.1/data/pg_hba.conf`

Example line added to `pg_hba.conf`: `host all gforge 172.31.34.148/32 trust`

Development

Topics

DMC REST Services Development Pull down the latest code from git:

```
git clone https://<username>@bitbucket.org/DigitalMfgCommons/restservices.git
```

Make any changes to the source code in the `src/main/java` branch. Tests are contained within `src/test/java`.

Navigate to the directory of the `pom.xml` file. This is the build file that Maven uses. Make sure to include any dependencies not explicitly used by Java here (`org.json`, `springboot-io`, etc.)

Once you are happy with your code, you have some choices:

1. You can start a local instance of your application with an embedded Tomcat container using `mvn spring-boot:run`. This lives inside the shell until you kill the process. This process requires no more configuration.
2. You can run your tests with `mvn integration-test -P <profile listed in pom.xml>`. This will again start an embedded Tomcat container and run your tests against it. The Tomcat application will be terminated upon the end of your tests.

Results will be in the `target/folder`.

3. You can create a WAR file by running the command `mvn package`. If successful, once complete, you will have a WAR file following your naming convention (set in `pom.xml`) in the `target/folder`.

1.2 DOME

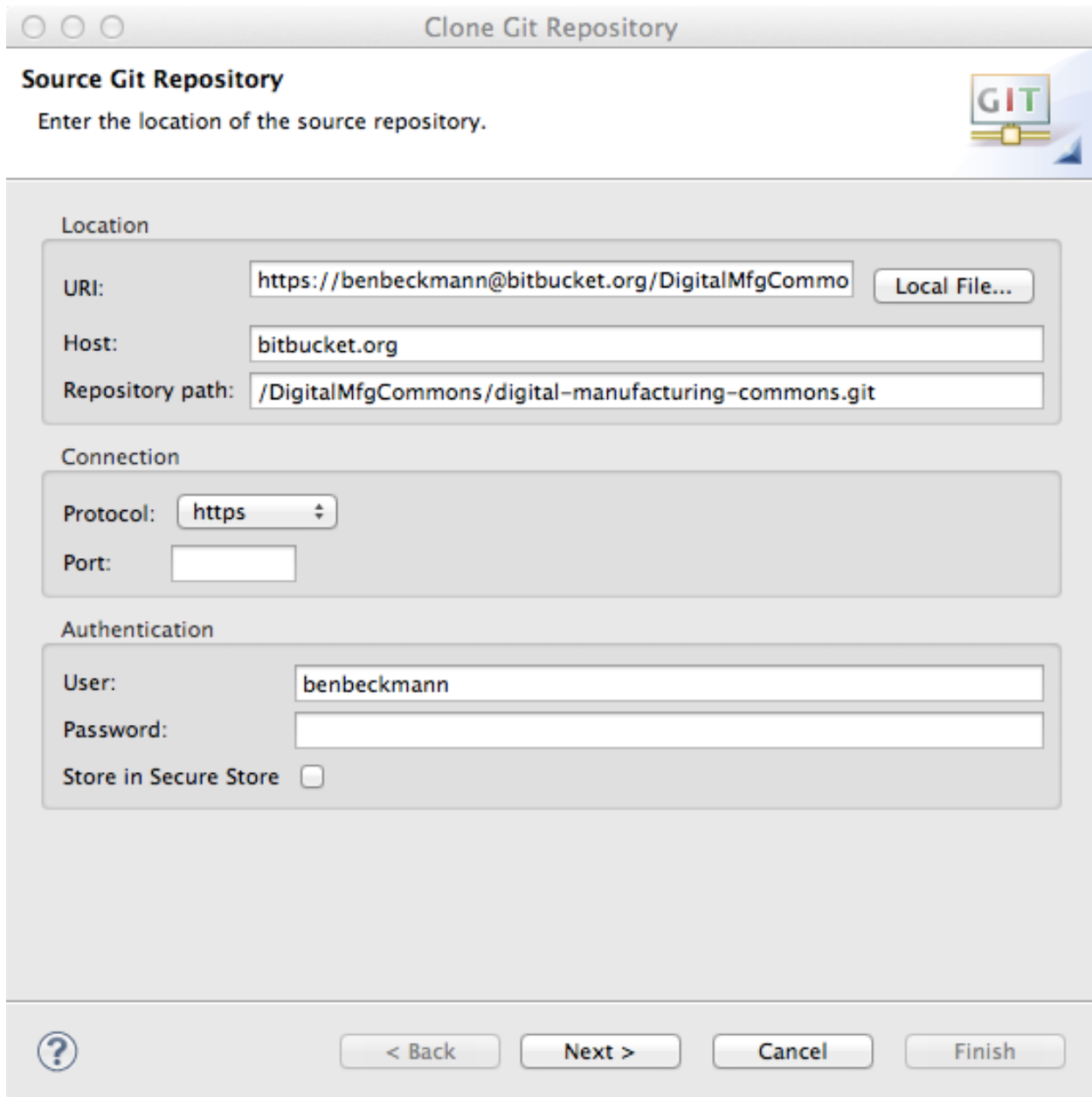
Note: This documentation is a work in progress. Topics marked with a TODO stub-icon are placeholders that have not been written yet. You can track the status of these topics through our public documentation [issue tracker](#).

1.2.1 Topics

DOME Installation

Warning: These instructions, intended for developers, point to the latest working version of DOME. Users interested in the latest stable version can find instructions [on the wiki](#).

This page contains basic instructions for installing DOME. For more details, please see the subpages.



The image shows a 'Clone Git Repository' dialog box with a title bar containing three window control buttons. The main title is 'Clone Git Repository'. Below it, the section 'Source Git Repository' is followed by the instruction 'Enter the location of the source repository.' and a small Git logo. The dialog is divided into three sections: 'Location', 'Connection', and 'Authentication'. The 'Location' section contains fields for 'URI' (https://benbeckmann@bitbucket.org/DigitalMfgCommo), 'Host' (bitbucket.org), and 'Repository path' (/DigitalMfgCommons/digital-manufacturing-commons.git), along with a 'Local File...' button. The 'Connection' section has a 'Protocol' dropdown set to 'https' and an empty 'Port' field. The 'Authentication' section includes 'User' (benbeckmann), an empty 'Password' field, and a 'Store in Secure Store' checkbox. At the bottom, there is a help icon, and navigation buttons: '< Back', 'Next >', 'Cancel', and 'Finish'.

Clone Git Repository

Source Git Repository
Enter the location of the source repository.

Location

URI:

Host:

Repository path:

Connection

Protocol:

Port:

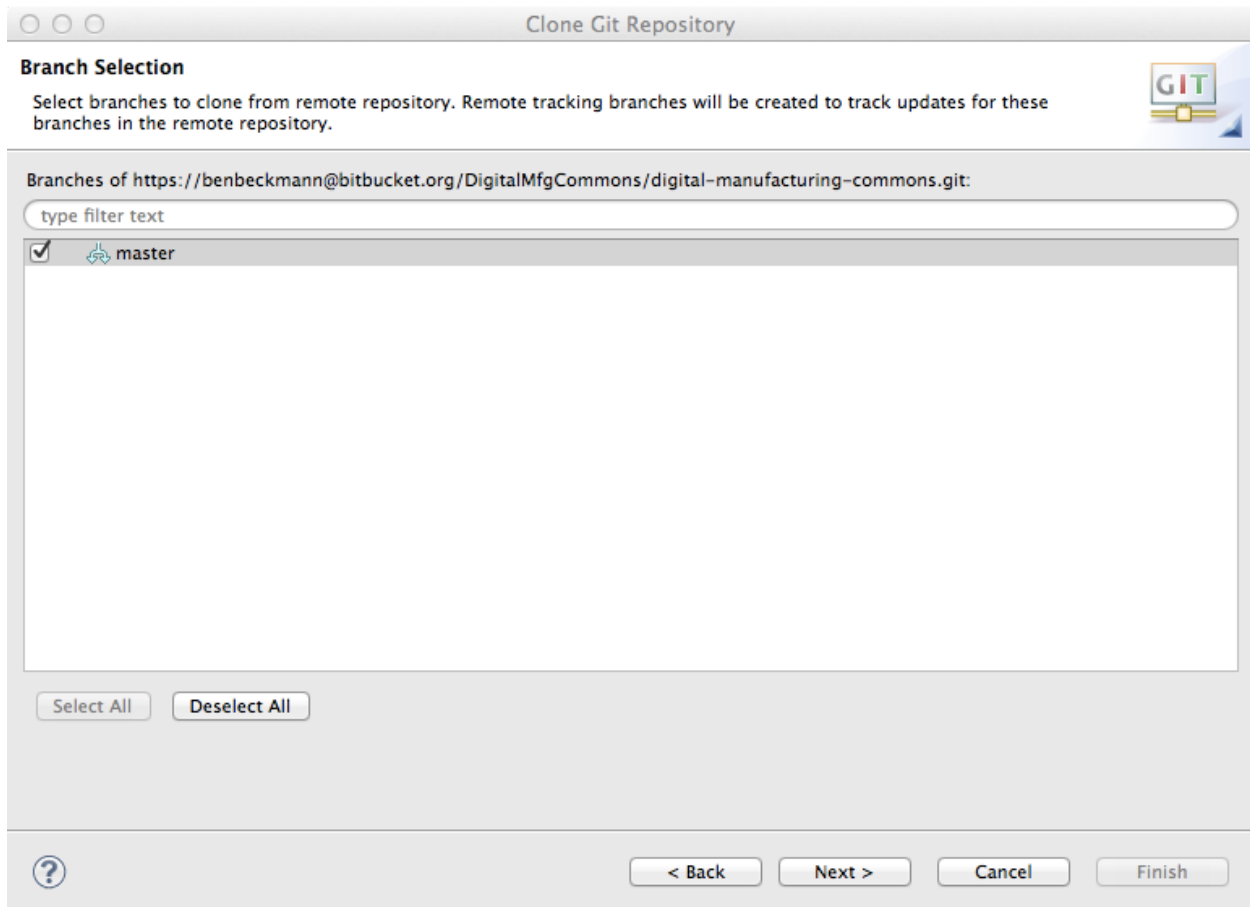
Authentication

User:

Password:

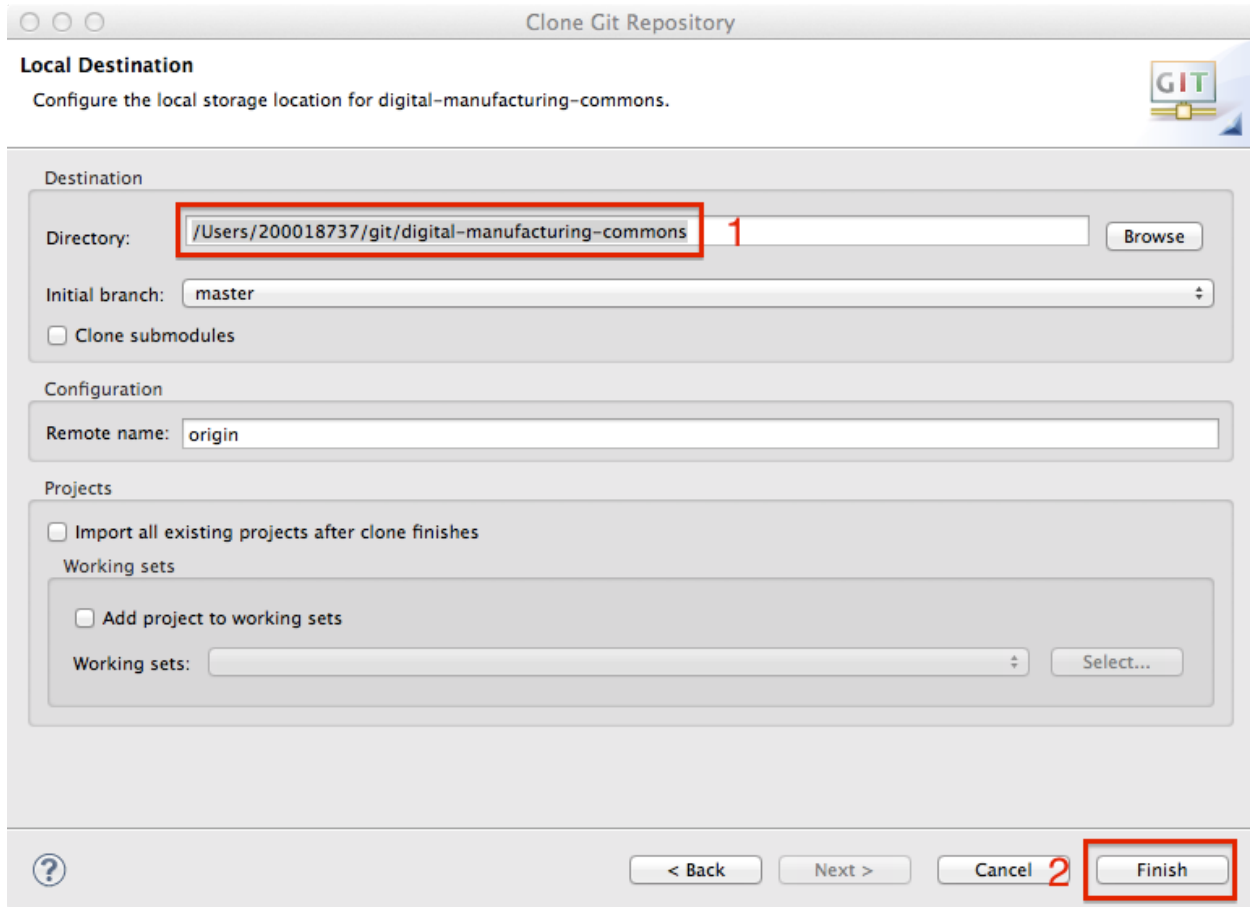
Store in Secure Store ☐

Step 3: Select the master branch to clone



Step 4: (1) Choose the local destination and then (2) click “Finish”

The git repository will now be cloned.



How to check-out and run DOME code This article outlines how to check out DOME source code from DMC repository, to compile and to run the code.

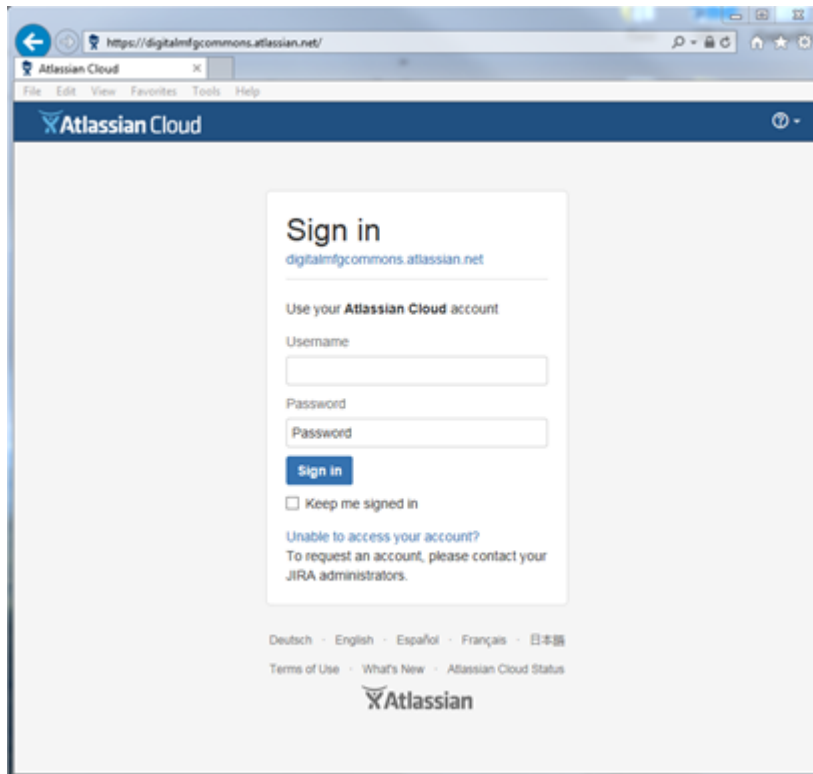
Prerequisites:

- User accounts already created in Digital Commons Cloud collaboration tool (atlassina.net) and source management tool repository (<https://www.bitbucket.org>) before the following operations.
- The following tools need to be installed: JDK (Java Development Kit), Git (<http://git-scm.com/download/>), Ant (<http://ant.apache.org/>), optionally Eclipse (<http://www.eclipse.org/home/index.php>).

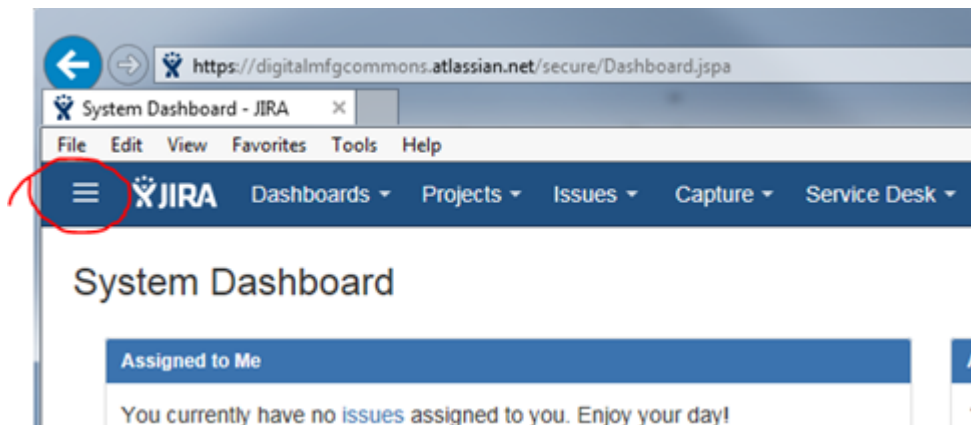
Step-by-step guide

1. Login to Atlassian Cloud.

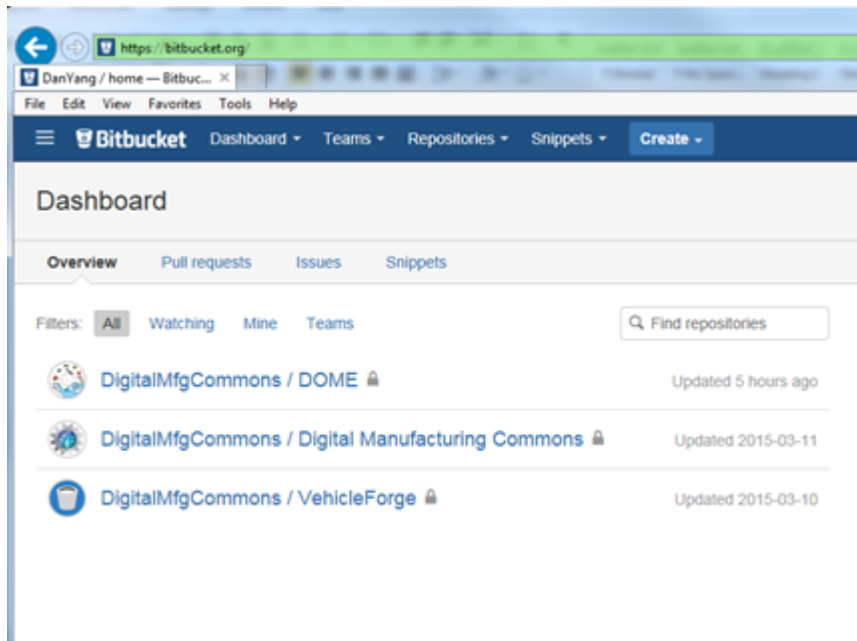
From a web browser, type the following URL: (<https://digitalmfgcommons.atlassian.net/>) When a login screen appears, type your atlassian.net user name and password:



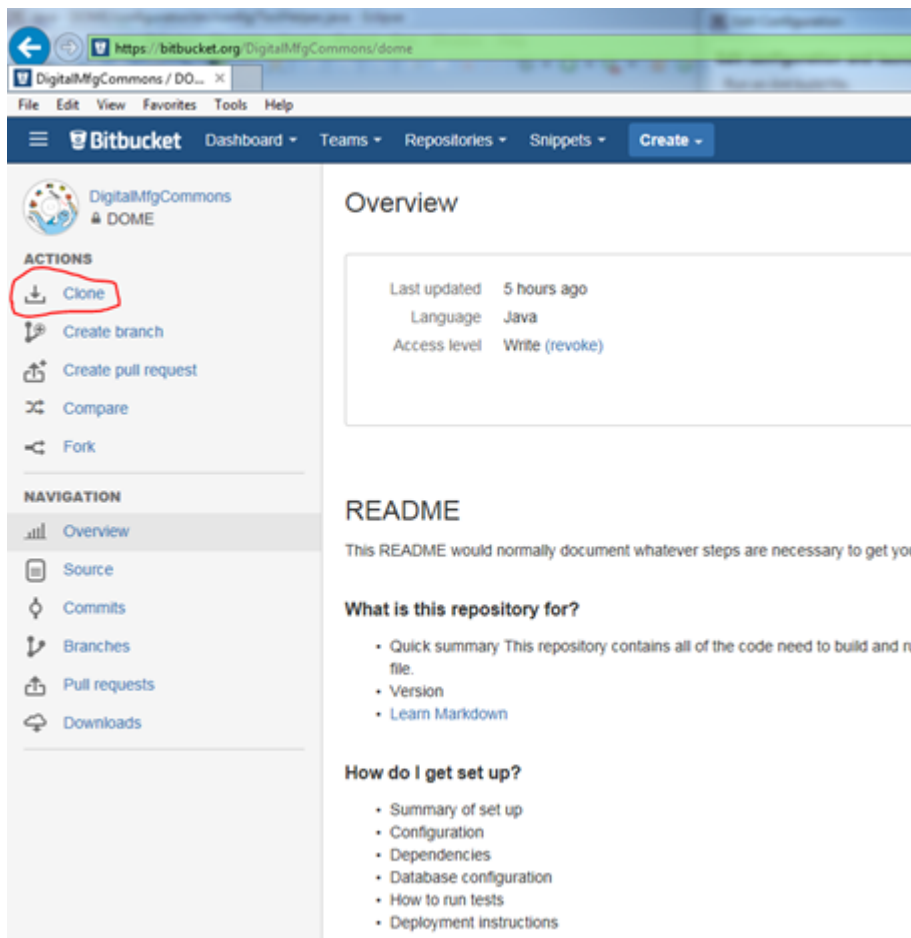
2. Go to BitBucket in Atlassian (<https://bitbucket.org/DigitalMfgCommons/dome>) After login, click the option list on the top left corner to go into BitBucket. Note: if you have not sign in to BitBucket, and remember the user name and password on the machine, you will be ask for BitBucket user name and password.



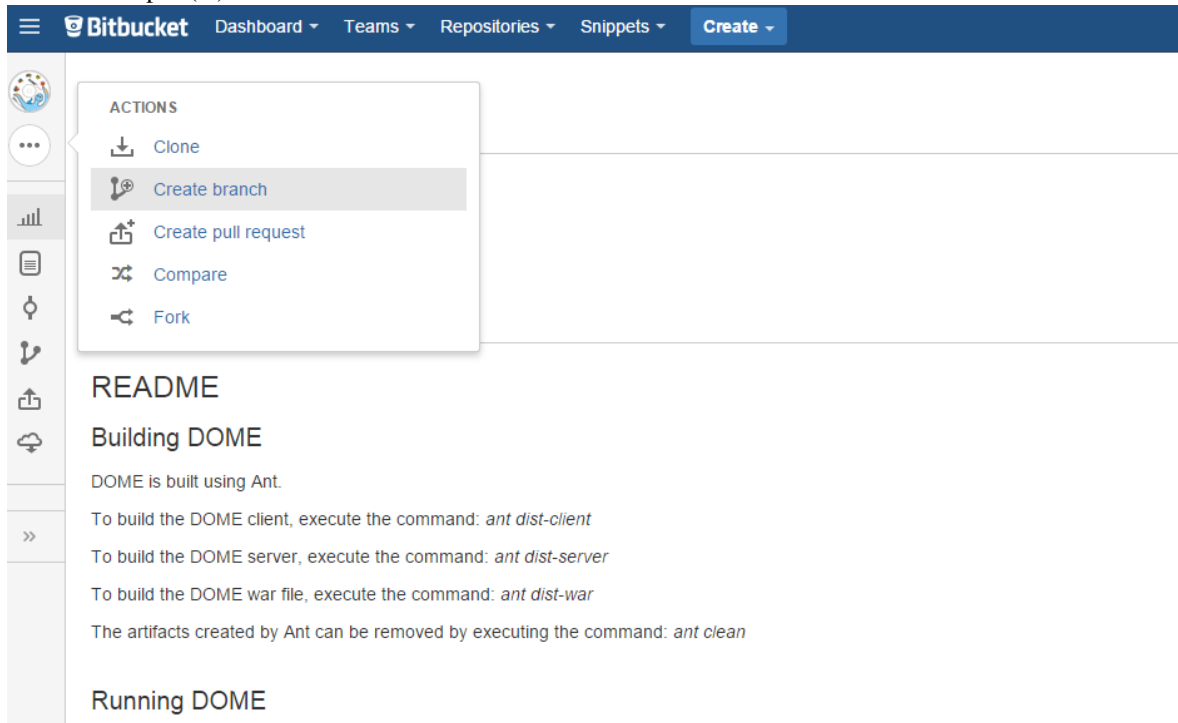
After go into BitBucket, you can see all the available projects listed similar to below:



There maybe more projects listed for individual users, but you should have “DigitalMfgCommons / DOME” for the DOME source code repository. Click on it, the following page will shown with Clone listed on the left as one of the Actions:



Note: If you do not see the Actions panel, your configuration may have Clone in the fly-out action menu from the ellipse (...) as shown here:

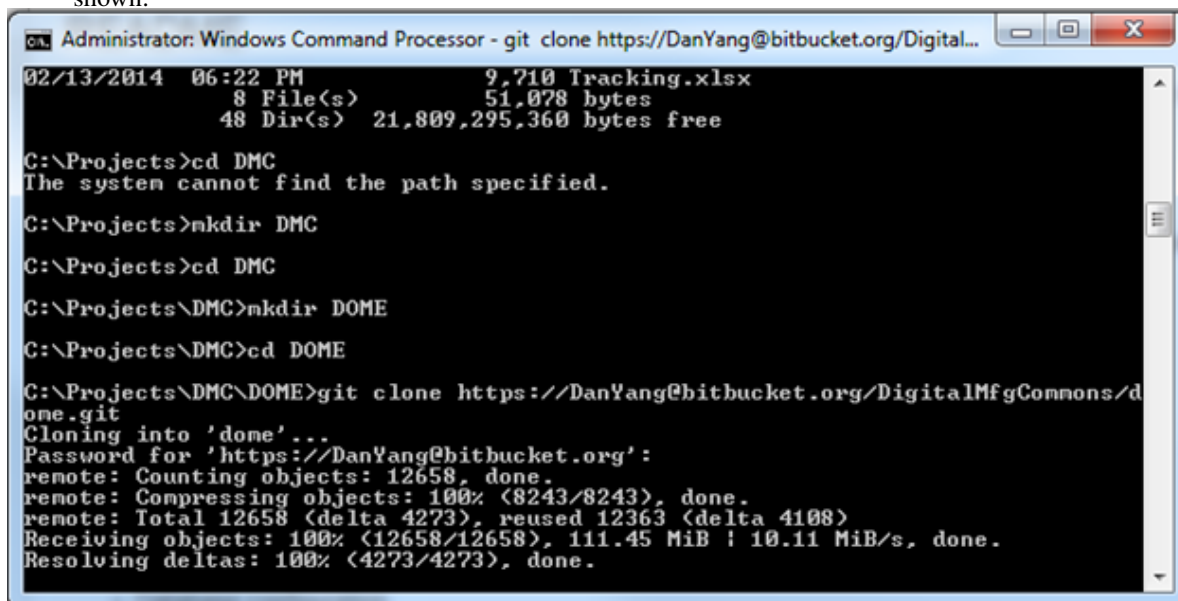


Click “Clone”, the system will pop up a small page generate Git script needed to check out the source code from BitBucket, for example:

```
git clone https://BitBucketName@bitbucket.org/DigitalMfgCommons/dome.git
```

Hit Ctrl+C to copy the string into buffer.

3. If you use Windows system, open a command line window, create a directory that you would like to check out the project. Type or paste (from C:icon from top left of the window) the git script you copied from the previous step to checked out code from BitBucket. When the checkout process finishes, a message similar to below will shown:



Note 1: You can perform same operations if you use Linux system.

Note 2: You need to install Git before this step. Please refer to the prerequisites.

Note 3: If you access the internet from within a firewall through proxy server, you need to obtain proxy server and port from your network system administrator, and use the following two steps to set up proxy server for Git so that it can check out code from internet. On Windows system, use system tool to add environment variable `http_proxy` (for example, refer to: <http://best-windows.vlaurie.com/environment-variables.html>):

```
http_proxy=GRC_Proxy_Server:port
```

After the variable is defined, use the following command to define proxy server for git:

```
git config --global http.proxy %name of the proxy%
```






























To get proxy server and port, you can either ask system administrator, or go to Internet Explorer, then go to Tool-> Internet Options -> Connections -> Lan Setting. From there, you can find the proxy server and port. Or you can open the proxy script to find the main proxy server. You can use similar commands to set up proxy server for git on Linux.

Note 4: Use your Bitbucket username before @bitbucket.org.

Note 5: Make sure you have permission to access Bitbucket before do git clone command. Now enter the dome directory

```
cd dome
```

4. After check out, you should see directory structure similar to that listed below with an Ant project build file `build.xml` included:

Name	Date modified	Type	Size
 .settings	3/26/2015 12:08 PM	File folder	
 bin	3/26/2015 3:22 PM	File folder	
 bld	3/26/2015 3:24 PM	File folder	
 configurator	3/26/2015 12:08 PM	File folder	
 dist	3/26/2015 3:26 PM	File folder	
 dlls	3/26/2015 12:08 PM	File folder	
 documentation	3/26/2015 12:08 PM	File folder	
 groovy	3/26/2015 12:08 PM	File folder	
 idea	3/26/2015 12:08 PM	File folder	
 lib	3/26/2015 12:08 PM	File folder	
 models	3/26/2015 12:08 PM	File folder	
 out	3/26/2015 12:09 PM	File folder	
 prototypes	3/26/2015 12:09 PM	File folder	
 python	3/26/2015 12:09 PM	File folder	
 scripts	3/26/2015 12:09 PM	File folder	
 src	3/26/2015 12:09 PM	File folder	
 test	3/26/2015 12:09 PM	File folder	
 WebContent	3/26/2015 12:08 PM	File folder	
 .classpath	3/26/2015 12:08 PM	CLASSPATH File	5 KB
 .gitignore	3/26/2015 12:08 PM	Text Document	1 KB
 .gitmodules	3/26/2015 12:08 PM	Text Document	1 KB
 .project	3/26/2015 12:08 PM	PROJECT File	1 KB
 bookmarks.xml	3/26/2015 12:08 PM	XML Document	1 KB
 build.xml	3/26/2015 12:08 PM	XML Document	33 KB
 DOME3.iml	3/26/2015 12:08 PM	IML File	7 KB
 DOME3.ipr	3/26/2015 12:08 PM	IPR File	16 KB
 DOME3.iws	3/26/2015 12:08 PM	IWS File	120 KB
 README.md	3/26/2015 12:08 PM	MD File	1 KB
 unittest.properties	3/26/2015 12:09 PM	PROPERTIES File	1 KB

5. (Option 1) You can use Ant tool command line window to build the DOME client and server. If you run ant in a command-line window, it should look something like the following in Windows:

```

C:\dmc\dome>ant run-server
Buildfile: C:\dmc\dome\build.xml

-init:
[mkdir] Created dir: C:\dmc\dome\bld\client
[mkdir] Created dir: C:\dmc\dome\bld\server
[mkdir] Created dir: C:\dmc\dome\bld\dome3
[mkdir] Created dir: C:\dmc\dome\bld\war
[mkdir] Created dir: C:\dmc\dome\bld\test
[copy] Copying 26 files to C:\dmc\dome\bld\test

compile-all-dome:
[javac] Compiling 1634 source files to C:\dmc\dome\bld\dome3

```

You need to use the following command to run Ant target: * To clean any old compiled code or residuals:

```
ant clean
```

- To compile and package for distribution (note: this step may take a while, wait until system outputs ant build successful) :

```
ant dist
```

- To run DOME server (after the server is running, leave the command line window open):

```
ant run-server
```

- To run DOME client:

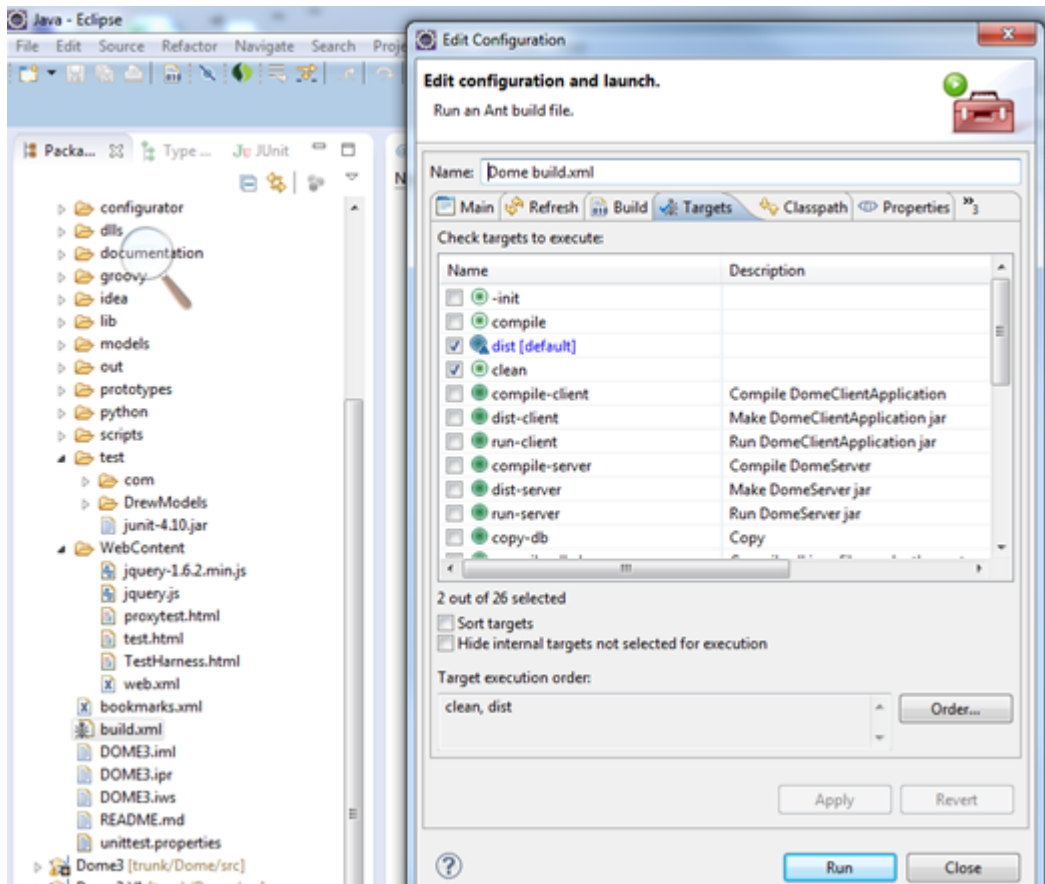
```
ant run-client
```

6. (Option 2) Build/run DOME server/client in Eclipse environment. See Development Environment Setup.
7. (Option 3) Build/run DOME server/client in Eclipse environment - git repository already cloned. To create a project using the checked out code and Ant build file, use Eclipse main menu: File -> New -> Project -> Java Project. Uncheck the Use default location and browse to the directory where you cloned the repository.

dome/installation/_static/howto-checkout/step7-new-project.jpg

Attention: Earlier versions suggested importing from Existing Ant Buildfile, but this does not work as well because the classpath and run directory end up in your workspace instead of the code directory.

After the project is imported to to Eclipse, if you have Ant tool in Eclipse environment, you can see that the build.xml in DOME project is marked as an Ant file. With that, right click build.xml, and choose Run As -> Ant Build..., all the available targets are listed on the right panel. These will get saved as “External Tools Configurations”.



You can use this to build distributable packages, as well as to run the DOME server and client as explained below:

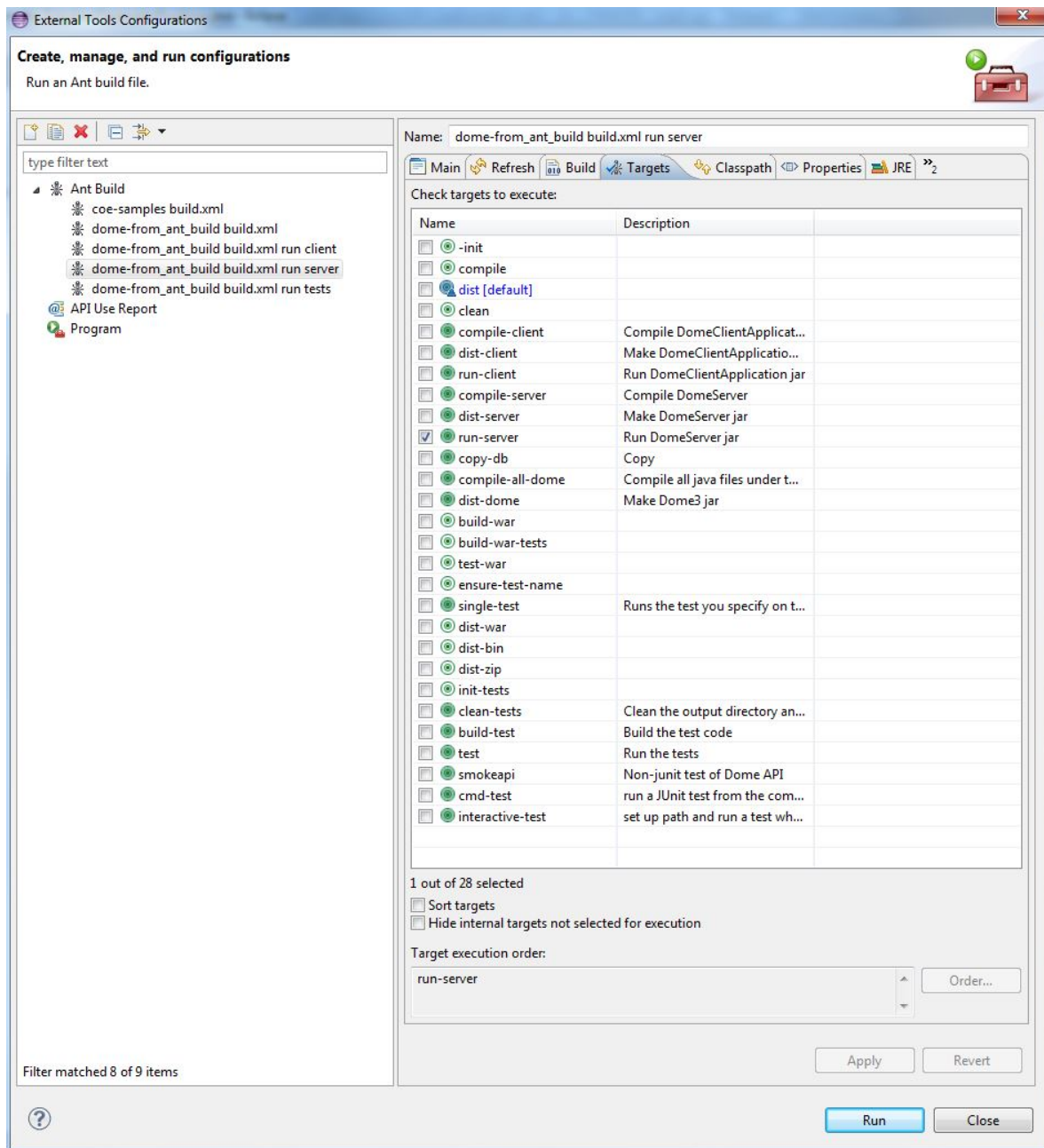
- clean – clean and initialization of the system
- dist – compile and build distribution packages for both client and server
- run-server – run DOME server project. From console you can see the following information: .. image:: _static/howto-checkout/step7-run-server.png
- run-client – after you run client the following message will appear in Eclipse console, and the DOME interface will appear. From here on, you can start to learn DOME model with the existing examples, or you can build your own models. .. image:: _static/howto-checkout/step7-run-client.png

Attention: You will need multiple Ant configurations to run both client and server from the same Eclipse workspace since both will remain running. A best practice is create at least 3 or 4 configurations:

- build - has clean and dist
- run-client
- run-server
- test

The copyright notice might appear behind your Eclipse window - if nothing seems to be happening after you start the client, minimize your Eclipse window. Make sure you have started the server before trying to login.

Example External Tools Configurations:



Quick Start

Clone [the DOME repository](#) using command-line git or your favorite GUI:

```
sudo apt-get -y install git
cd ~
git clone https://<username>@bitbucket.org/DigitalMfgCommons/dome.git
```

Install ant

```
# check whether ant is already installed on your system (likely)
ant -version
```

```
# install ant if necessary
sudo apt-get install ant
```

Build DOME

```
cd digital-manufacturing-commons/DOME      # Navigate to the directory containing build.xml
ant dist-client                            # build client
ant dist-server                            # build server

# Other options:
ant dist-war                               # build DOME war file
ant clean                                  # remove artifacts created by ant
```

Software Architecture

Dome Tools

The purpose of developing integrated simulations is to allow a wide variety of individuals, ranging from scientists to designers and policy-makers, to explore the characteristics of different alternatives. Tools are special models that run other models for different analysis purposes. There are a number of different analysis tools that have been wrapped as plugin models so that they can interface with DOME models.

Access Privileges

DOME3 supports the definition of users (username and password) and the assignment of users to groups while in server mode. Groups are collections of users that will be treated identically from an access permission standpoint. For example, all individual users that are a member of the CADlab group are eligible to log into the CADlab server. During deployment, user and group access to individual model interfaces is assigned. Thus, a given user may not be allowed to see all of the interfaces to a model that are deployed on a server. Additionally, users and groups are assigned to playspaces, thereby defining those who are allowed to participate in a collaborative work area. Even if a user has permission to enter a playspace, they will only be able to see model interfaces within according to access privileges assigned to the individual interfaces.

Integration Projects

Integration Projects are a special type of model. They may contain other models or projects as computation resources, along with integration models that define interactions between resources. Integration models are similar to DOME models, with the exception that they may utilize parameters from the project resources and they may not be used independent of the project. Thus, projects are used to build integrated simulations. Since projects are a type of model they may have interfaces and user defined documentation.

Interfaces

All types of models are executed through interfaces. Interfaces provide views that define which entities will be observable by users of the model at run-time. Model entities that are not in an interface are private and consequently will not be available to users. Model builders can add parameters that are in a model to an interface. Further, all interfaces automatically contain filters to identify interface parameters that are independent inputs to and results from the model. Interfaces also contain a model view that allows anything within a model to be exposed to users in a read-only format. Multiple interfaces can be associated with models to create different interfaces for different model users. The ability to define multiple interfaces for a model increases the reusability of a model. The car example below provides a simple illustration of a model with multiple interfaces.

Car model (a flat list of entities) Door length parameter Overall length parameter Wheel base parameter Length relationship: Overall Length = 3 x Door Length Car Context (a hierarchical visualization structure defined by the model builder) Car Dimensions Context Door length parameter Overall length parameter Wheel base parameter Car Door Design Context Door length parameter Relationships Filtered Context Length relationship Door Designer Interface Inputs Filtered Context (parameter values that can be changed by a user) Door length parameter Results Filtered Context Overall length parameter Body Designer Interface Inputs Filtered Context (parameter values that can be changed by a user) Wheel base parameter Car Dimensions Context (parameters viewable by a user) Door length parameter Overall length parameter Wheel base parameter

Access privileges defined during deployment determine what interfaces can be seen by different model users. All interfaces have support for builder-defined documentation.

Playspaces

Users may collaborate, sharing the same model and data, through a playspace. When models or projects are deployed onto a DOME server, they are automatically available for individual use (i.e., every user of the model works with a separate or different instance of the model). Additionally, collaborative playspaces or work areas can be created so that these deployed models can be used collaboratively. Then, when different users run the same model through a common playspace they will share the same instance of the model. Access privileges may be defined to determine which users are allowed to enter a playspace when the playspace is deployed.

Models

Models are DOME objects that provide executable simulation capabilities. Fundamentally, there are three different kinds of models: DOME-native; plugins; and integration projects. From a use standpoint they look the same, but they differ in how their internal computation is performed. DOME-native models are completely implemented in the DOME environment and during execution all computation is performed within DOME. DOME models are most commonly used within integration projects to define relationships between other model interfaces. Plugin models are models where the underlying computation is performed by an external software application, such as a spreadsheet, CAD system, commercial simulation tools, analysis tools, optimization tools, or models written in a programming language. Plugin models wrap the non-native computation with a DOME model and interfaces. An integration project is a special form of model that is used to connect or integrate other models. Integration projects have standard model interfaces

Dome Native Models DOME-native models are completely implemented in the DOME environment and during execution all of the simulation computation is performed within DOME. Simulations are created by defining DOME parameters within the model. These parameters can then be linked using mathematical DOME relationships. When changes to the value of parameters are made, the DOME server in which the model resides will solve the model's relationship network and determine which relationships need to be executed and in what order.

Plugin Models Plugin models are any type of model where the executable simulation capabilities are provided by an external software application, such as a spreadsheet, CAD system, commercial simulation tools, analysis tools, optimization tools, or custom models written in a programming language. Several software programs are supported within DOME as plugin models. The analysis tools available within DOME are implemented as plugin models DOME models wrap the external simulation with DOME parameters that are connected to data elements within the external simulation. The two figures below provide a conceptual illustration of how DOME interacts with software possessing structured application program interfaces (APIs) and batch mode applications.

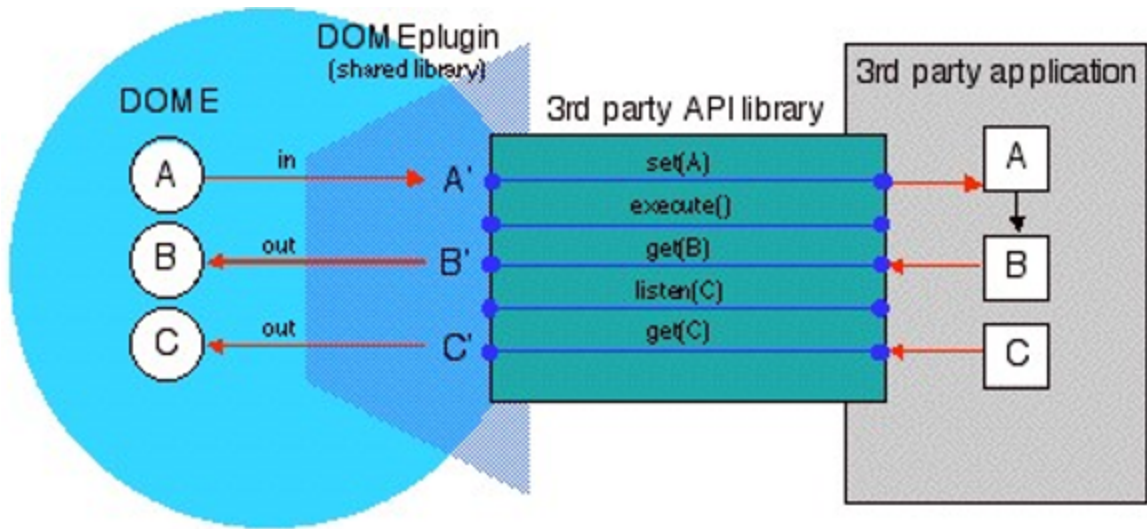


Figure 1: Connection of DOME parameters to software with a structured API library. This type of communication typically adds a few milliseconds (less than 5) per execution of the simulation in the external application.

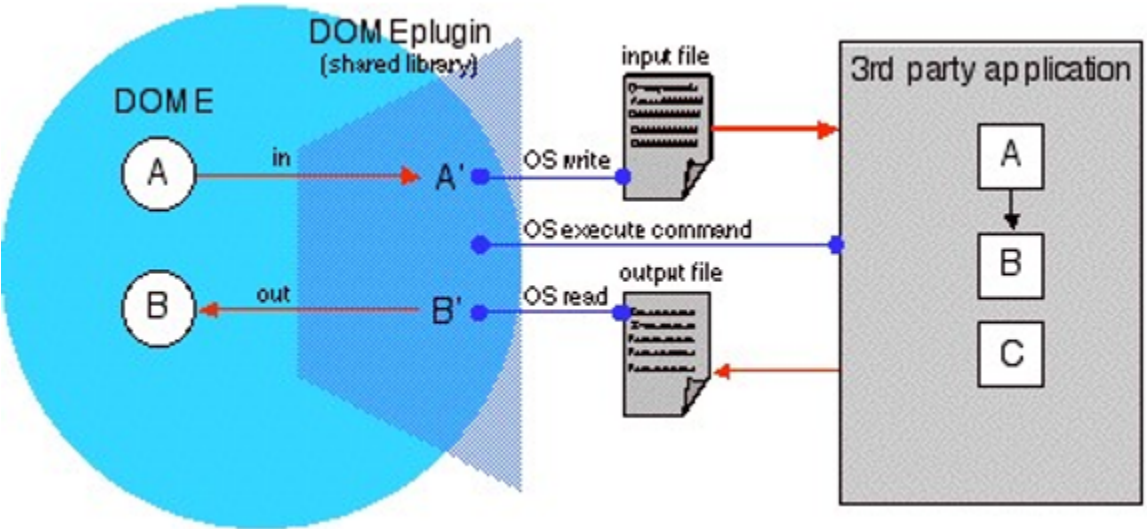


Figure 2: Connection of DOME parameters to software that runs in batch mode. This type of interaction can add hundreds of milliseconds to seconds per execution of the simulation in the external application, depending upon how long it takes to read/write the input/output files.

Model Objects

This section provides an overview of the main object types that may be used within models. Different types of models may not support all model objects.

Context **Attention:** Need to fix formatting errors.

Context objects can be used in models to organize entities into meaningful groupings or viewpoints. One might imagine context Door length parameter All context have support for builder-defined documentation.

Filters Filter objects are a special type of context. Filters automatically generate views containing entities with specific common characteristics. For example, a relations filter applied to the scope of a complete model will automatically generate a visualization of all relationship objects in the model. Similarly, an independent parameter filter will automatically generate a view showing all parameters that are not driven by relationships. Filters are used as tool to create visual structures that aid in the understanding of DOME models (standard views, for example).

Parameters Parameters are objects that contain data such as numbers (real, integer, complex), distributions, Booleans, vectors, lists, matrices, tables, or files. Parameters contain neutral data-types that DOME can perform computation upon, move over the Internet, and connect to data in other application types via plugin models. The form of a parameter's value is dependent upon its assigned data type. Similarly, valid operations for parameters in a relationship are dependent on the data type. Parameters support units when they are appropriate for the chosen data type. Constraints can be applied to parameters. Once again, the form of valid constraints will be dependent upon the parameter's chosen data type. For example, a real number might have a constraint such as $0 < \text{value} < 100$, while a file constraint might be $\text{size} < 1\text{MB}$. Constraints may be hard or soft. When hard constraints are violated the execution of a model is stopped. When soft constraints are violated warnings are generated. All parameters have support for builder-defined documentation.

Relations Relation objects are used to define mathematical relationships between parameters within a model. Then, when the value of parameter data changes, DOME determines which relations are affected and the correct order for executing the relations. The interface of a relation contains DOME parameters that are input variables or outputs, as determined by the body of the relation. Relation parameters can be mapped (connected) to other parameters within a model. The body of a relation defines a mathematical expression or procedure involving the relation parameters (e.g., $c = a + b$). The relation body is written in Python. Operations that can be performed on parameters are dependent upon the parameter's data type. When united parameters are used in a relation, expressions are checked for dimensional consistency and units are converted automatically if they are inconsistent. Thus, model builders are not burdened with ensuring that units are consistent. All relations have support for builder-defined documentation.

Visualizations Visualizations are objects that accept data from parameters and transform them into a various graphical visualizations. For example, using a visualization object a matrix might be viewed as a elevation map, or two vectors might be plotted as an XY graph. All visualizations have support for builder-defined documentation.

Application Modes

DOME3 is a set of specialized applications that together provide a complete WWSW environment. Each application is focused for a different type of user or use.

Build mode is an authoring application for model builders who define models and interfaces.

Deploy mode is used to place models and interfaces on DOME servers.

Run mode allows users to surf the WWSW and execute models. It is a browser application.

Server mode is used to manage the users, groups, passwords and server file space

Build Mode The build authoring application allows users to create models, projects, and playspaces. The application provides mechanisms to find and subscribe to interface parameters of other models already deployed in the World-Wide-Simulation-Web. When working in build mode the DOME models reside in one's local file space—not on a DOME server—and thus they are not available for use by others.

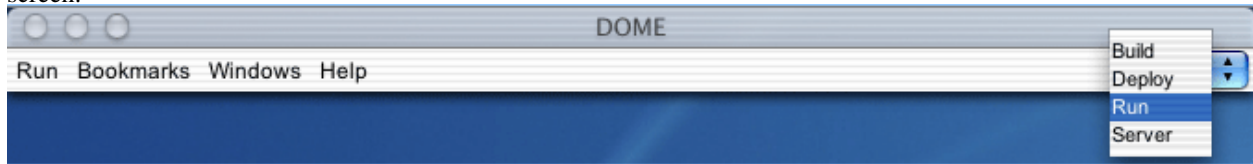
Deploy Mode The Deploy Mode application is used to move models and playspaces to servers so that they are accessible to others in the World-Wide-Simulation Web. During the deployment process use and editing access privileges are defined.

Run Mode The Run Mode application is the browser of the World-Wide-Simulation-Web. It is used to surf between DOME servers and log into standalone models or collaborative playspace work areas. Parametric what-if scenarios or various types of analyses can be executed through the run application.

Server Mode The Server Mode application is an administrative tool. It is used to determine how models and playspaces are organized, track use, perform software maintenance, and define users and groups.

Using DOME3

Using DOME3 is divided into four separate documents, one for each of the use-mode applications. A number of tutorial examples are also provided. To begin: Start the DOME3 application installed on your computer. This will execute an integrated environment that wraps the DOME suite. A DOME3 menu bar will appear at the top of your screen.



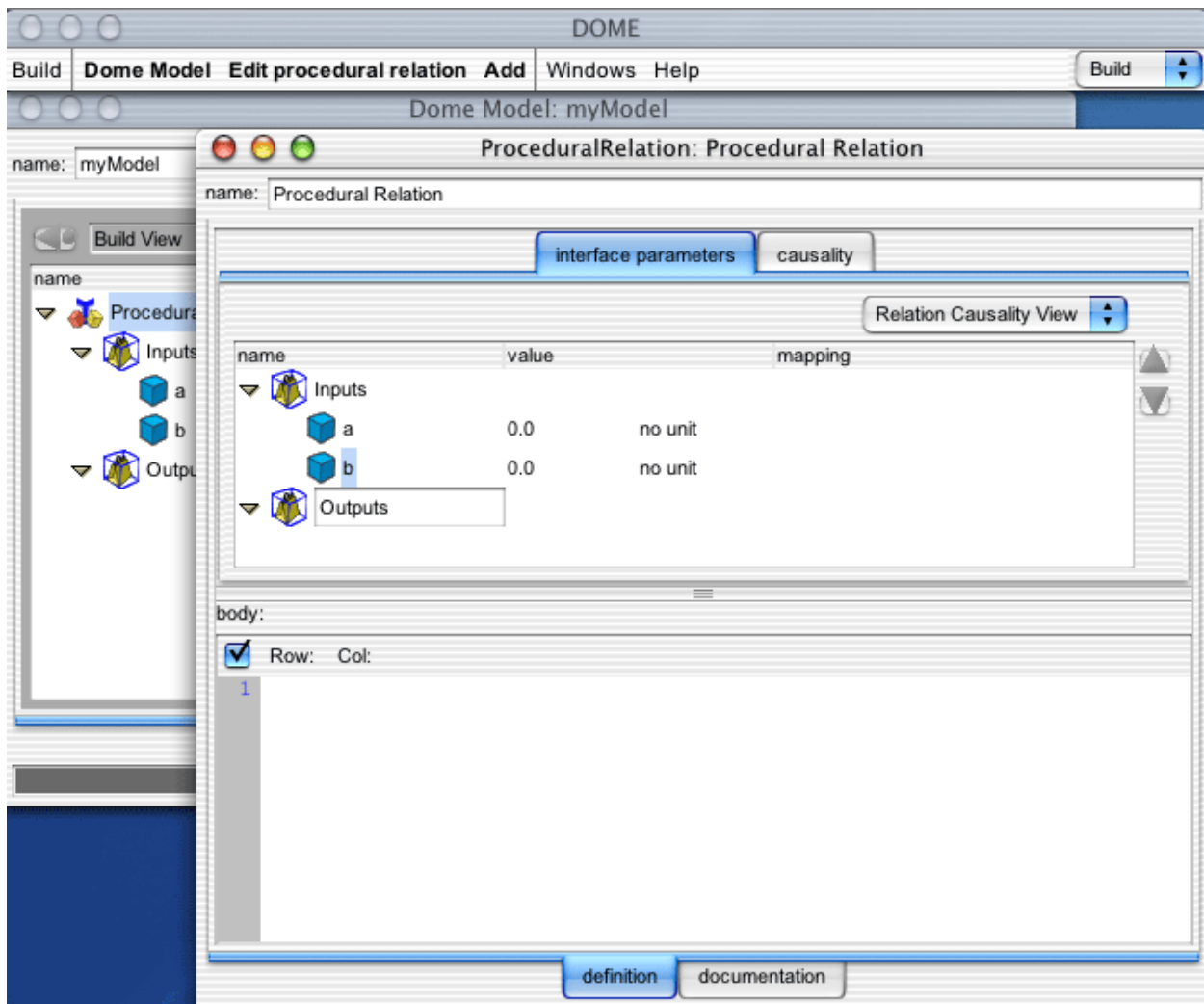
In order to avoid being confused when you start to build models or projects, it is important to understand the difference between an actual object and viewing a reference to an object. You may also want to review the general behavior of the build mode GUI environment.

Window Management

Keeping track of windows in multiple-window applications can be a challenge and DOME3 is no exception. Therefore, a number of mechanisms are provided to help avoid window bloat. * Window menu support for locating open windows (in the DOME3 menu bar). * Windows are color coded by model (colored square icons are in the left corner of the window title bar), making it easy to recognize all windows that belong to objects in the same model (not available on Mac OS X). when a model is minimized, all other object GUIs associated with the model are also minimized. * A minimized window can be opened by clicking on its icon. * when you move from one application mode to another, all open windows associated with the mode are hidden. They will reappear when you return back to the mode.

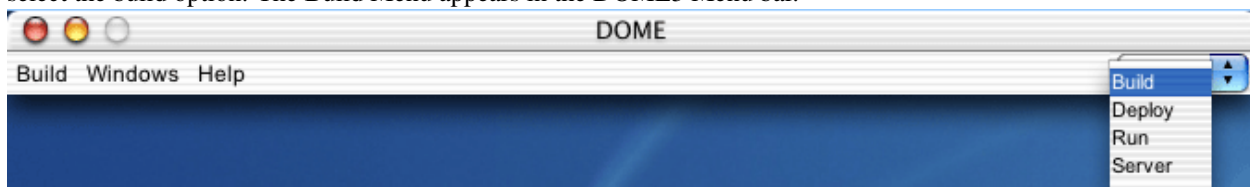
Context Dependent Menus

Context dependent menus are used throughout DOME3. Context dependent menus change according to the type of window that is selected (or in focus). Context dependent menus will always be in bold and placed between separators in the overall application menu bar. The figure below shows an example of the menu bar when a procedural relation GUI is in focus. Procedural relations may be added when creating DOME models.



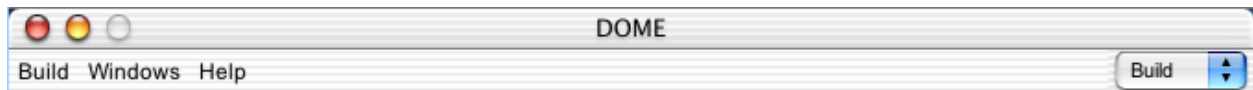
Build Mode

Build mode is used to create or edit models, projects, or playspaces and set up tools that can be applied to models. To start building: Make sure the DOME3 application is running. The DOME3 menu bar will be visible across the top of your screen (unless the application has been minimized). Using the combination box on the right of the menu bar, select the build option. The Build Menu appears in the DOME3 Menu bar.

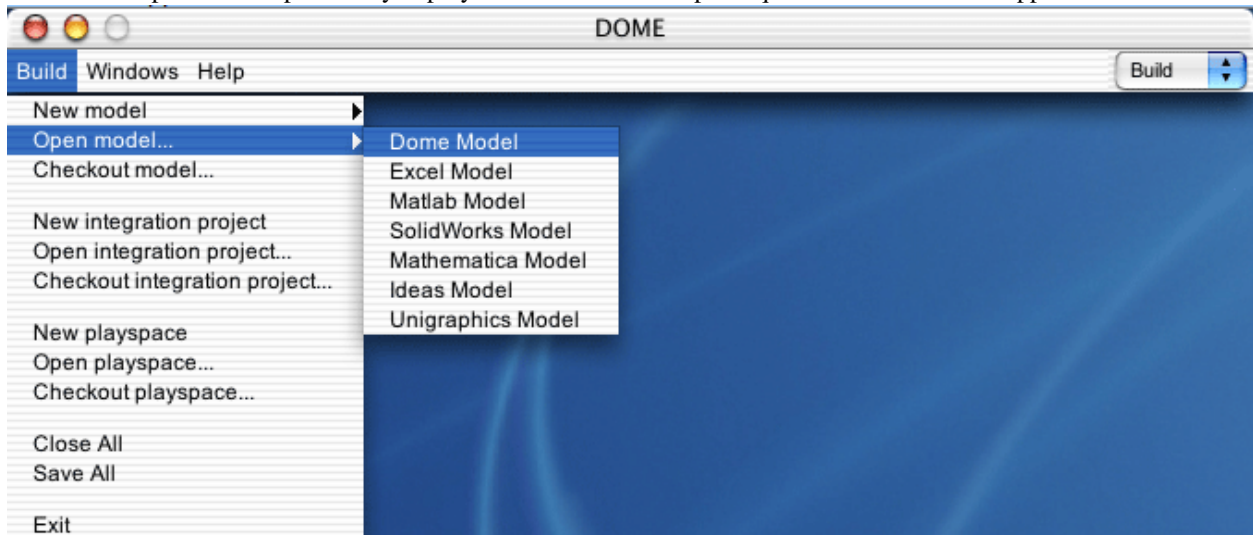


In order to avoid being confused when you start to build models or projects, it is important to understand the difference between an actual object and viewing a reference to an object. You may also want to review the general behavior of the build mode GUI environment.

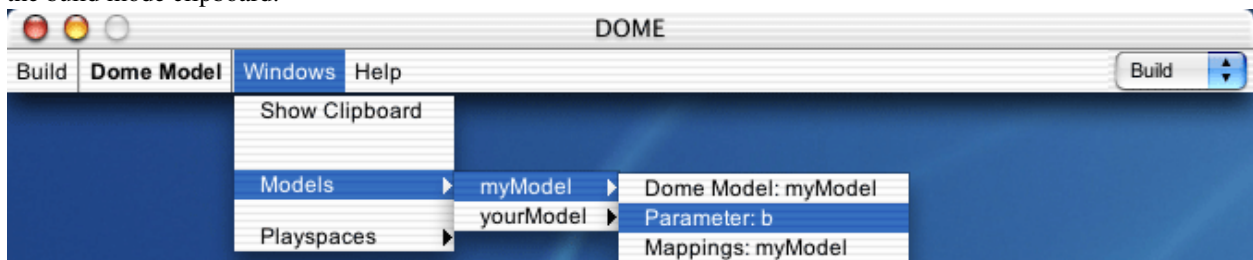
Build Menu Bar The Build Menu Bar is available at all times while in build mode. The build application-level menus (build, windows, help) are in plain (non-bold) typeface.



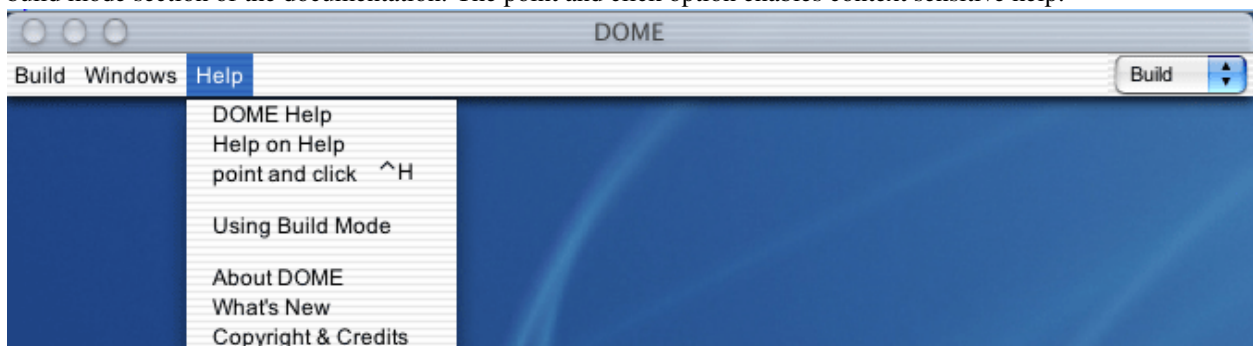
Build Menu The Build Menu is used to create or open models, projects and playspaces that are in your file space. Checkout is used to retrieve a deployed version from a DOME server so that it can be edited and, if desired, redeployed as a version update of the previously deployed model. The exit option quits the entire DOME application.



Window Menu The build windows menu tracks all build mode windows that are open. It is also used to hide/show the build mode clipboard.



Build Help Menu The build mode help menu contains standard help menu items, along with a direct link to the build mode section of the documentation. The point and click option enables context sensitive help.



Models All standalone models are created using the build mode application (starting from the build menu). This section provides instructions for building models in DOME:

- DOME Models (a DOME-native model)
- Abaqus Models (a batch plugin model)
- Adams Models (a batch plugin model)
- CATIA (a plugin model)
- Excel Models (a plugin model)
- I-deals Models (a plugin model)
- Matlab Models (a plugin model)
- Mathematica Models (a plugin model)
- Name-value Batch Model (a batch plugin model)
- Nastran (a batch plugin model)
- Solidworks Models (a plugin model)
- Unigraphics Models (a plugin model)

Before building DOME-native models you should understand the meaning of an object reference. All models have an associated message log and documentation editor. A clipboard is also available for copying and pasting model objects between different models.

References to Objects One of the most important concepts to understand when building and editing models is the difference between objects and references to objects. DOME3 models are made of different types of objects that capture data or describe interactions between data. The figure (at right) shows a model containing the parameters A, B and C (left of figure). The user's view of the model is seen through a DOME GUI. Views of models are created through context. In this case there are two context that contain references to objects in the DOME model. Context always contain references to objects, not the actual objects in the model. Thus, one can organize the model visualization using so that the same object C can be viewed through references in more than one context.

Now, if the reference B is selected and removed using the build GUI, the actual object B still remains in the DOME model.

Finally, if one of the references to C is selected in the GUI and the delete function is chosen, the object C in the model will be deleted along with all references to C in the GUI.

DOME Models DOME Models are fully implemented using only DOME parameters and relationships. They do not rely on any external programs to perform embedded computation. At times it may be convenient to implement simple models directly in a DOME model. DOME Models currently support sets of algebraic relationships without loops. DOME models support all types of DOME model objects. To work on a new DOME model: Put application in build mode. Select New->Dome Model the build menu, or if you already have a DOME model open in build mode you can select Dome model -> New in the DOME Model menu bar.

GUI

DOME Model Menu Bar The DOME model menu bar is available whenever a DOME model GUI is in focus and the definition tab is selected. This provides access to Dome model, edit, add, and tool functions. Depending on the standard view in use, all menu options may not be available.

When the documentation tab is selected, the documentation context menus are available.

DOME Model Menu	Menu item	Function
	New	Opens a new model of the same type (DOME model), avoiding the step of specifying the model as is required in the build menu.
	Open ...	Allows you to open other models of the same type (DOME model), whereas any type of model can be opened using the build menu.
	Save	Saves/overwrites the previously saved model file in your local file space. Build versioning occurs with model saves.
	Save as ...	Allows you to save the open model as a different model file.
	Close	Closes the current model, leaving the overall DOME application running.
	Test Model	Allows you to test execute a DOME model without going through the full deployment process.
	Test Relation	Allows you to test execute a relation. Available only when a relation is selected.

DOME Model Edit Menu The Edit definition menu appears when the definition tab of a DOME Model GUI is selected. Edit options are dependent upon the type of object that is selected. Thus all edit functions will not always be enabled in the menu.

The menu options apply only to DOME objects. Text field edit commands are available in a pop up menu.

TODO: Add table here

DOME Model Add Menu The Add menu appears when the definition tab of a DOME Model GUI is selected. When an object is added to a model, a reference is also added to the GUI visualization following tree insertion behavior. Objects available are listed in the menu. These include: context, parameters (real, integer, boolean, string, text, vector, matrix, enumeration, file and list), relations (procedural, equal), and visualizations. The table below describes add menu commands.

Edit Command	Result
Map->last selection	Available when a relation interface parameter or a model parameter is selected and a copied parameter is in the clipboard. Appends the last selection to the mapping list of the involved parameters.
Map->from clipboard...	Available only when a relation interface parameter or a model parameter is selected and a copied parameter is in the clipboard. Opens a dialog showing previous selections in the clipboard.
Add and Map->last selection	Available only when a relation object is selected. Adds a new parameter to the relation interface and maps it to the last copied parameter.
Add and Map->from clipboard...	Available only when a relation object is selected. Opens a dialog showing previous selections in the clipboard (see paste example above). After picking the desired selection, a new parameter is added to the relation interface and mapped to the selection.

DOME Model Tool Menu The DOME model Tool Menu is available whenever a DOME model is in focus and the definition tab is selected. The Tool Menu provides access to the Mapping Tool and the Interfaces Tool, which open in separate windows.

DOME Model Definition Tab When the DOME model definition tab is selected the DOME model menu (for adding objects, editing, mapping and defining interfaces) is available within the build menu bar.

The visualization combination box controls the overall type of model view. It is set to the standard list (tree-table) view in the figure below. The list view is the primary model editing view. There are also a number of alternative ways to visualize the model, including graph, matrix (DSM), and XML code.

Standard DOME Model List Views The DOME model list view is the main model building and editing view. There are three different types of list views that organize the contents differently: build view, object type view, and model causal view. The standard view combination box is used to switch between the the views.

All list views follow standard DOME tree table behavior. There are three columns in the DOME model list views. The left-most name column contains object icons and names. The middle value column displays a summary form of the object's value. Many object types support editing in the value column. Units may also be edited from the value column. The right-most mapping column lists other parameters that an object is mapped to, and is applicable to parameters only. Double clicking in a cell within the mapping column opens the mapping tool.

The standard view combination box and view controls are only function in the list view. They are disabled in when other model visualizations are in use. The list view controls are summarized below.

- The left arrow (beside the standard view combination box (reading build view) is used to change visualization scope. The current scope name is listed in the combination box.
- The up/down arrows on the right are used to re-order selected objects within a context.

DOME Model List: Build View Build View is the main view for defining and editing DOME models. All editing, adding, and tool functions are available. The different columns of the view are described in the standard list view page. Within this view you have complete flexibility to edit the model and structure it using context as desired. It is also possible to change viewing scope to different context within the view.

DOME Model List: Object Type View The Object Type View is a standard list view that includes filters that organize objects according to their type: parameter filter, relationship filter, and context filter, etc. This is a useful view to see only unique objects in a model without any multiple references (as are allowed in build view).

Since the contents of this view are generated automatically, one cannot add objects to the model in this view. There is no add menu associated with the view. However, copy and delete in the edit menu, along with mapping operations, are permitted.

DOME Model List: Causal View The Causal View is a standard list view that includes a number of filters to organize parameters according to the overall causal structure of the model: independents filter, indeterminates filter, intermediates filter, and results filter. This view, combined with the graph and DSM visualizations, are useful to develop and understanding of the model structure.

Since the contents of this view are generated automatically, one cannot add objects to the model in this view. The add menu is not available. However, copy and delete in the edit menu, along with mapping operations, are permitted.

Documentation Tab The DOME Model documentation tab contains the documentation editor and invokes the documentation editor context menus. The model can not be edited when the documentation tab is selected.

Adding Model Parameters Model parameters are used to create persistent data in a model. Parameter data types available in DOME models are listed in the DOME model add menu section. All parameter data types are detailed in the building parameters section. To add parameters to a DOME model...

- Make sure the model's add menu is available by selecting the list visualization build view in the definition tab.
- Select Add->desiredParameterType from the add menu.

References to new parameters are be inserted into the model list visualization. The location of the reference is determined by what is selected when the parameter is added. Once added, parameters can be defined as a constant and many data types also support the definition of constraints. References to parameters may be removed, or parameters may be deleted, from the model using the DOME model edit menu.

Adding Relations Relations are used to define interactions between parameters. Types of relations available in DOME models are listed in DOME model add menu section. All relation types are detailed in the building relations section. To add relations to a DOME model:

- Make sure the model's add menu is available by selecting the list visualization build view in the definition tab.
- Select Add->desiredRelationType from the add menu.

References to new relations will be inserted into the model list visualization. The location of the reference is determined by what is selected when the relation is added. References to relations may be removed, or relations may be deleted, from the model using the DOME model edit menu.

Mapping Parameters Mappings can be added and edited using the mapping tool, which can be opened from the tools menu. Using the mapping tool, relations may be mapped to model parameters in either a model centric or relation centric process. In most instances it will be most convenient to add new mappings using the mapping shortcuts available in the add menu. What are mappings for? If model parameters are to change as the result of a relationship executing, or if changes in model parameter data are to cause a relation to execute, model parameters must be mapped to a relation's parameters. When parameters are mapped they are, in effect, connected and will always reflect the same value. Mappings are commonly used to connect parameters used shared by different relations. Only parameters with compatible data types may be mapped to each other.

Relation Centered Mapping Process In a relation centered mapping process, the mapping tool is used to first pick the relation that you want to work with and then map its interface parameters to model parameters. The process involves the following steps.

- Selecting a relation
- Selecting a parameter in the relation
- Selecting a model parameter
- Adding the map

Selecting a Relation The model/relation/interface combination box in the mapping tool GUI includes a list of all relations in the model. Any relation can be selected from the list so that selected interface parameters may be mapped to selected model parameters. In this case the relation ProceduralRelation is being selected.

Selecting a Relation Parameter The mappings for combination box in the mapping tool GUI contains a list of all parameters within the selected relation. Any relation parameter can be selected from the list so that it may be mapped to selected model parameters. In this case the relation parameter x is being chosen.

Selecting Model Parameters In order to select another parameter for mapping to the relation parameter, use the copy command in the edit menu. In this case the model parameter Ef is being selected.

Adding a Map Once a relation, relation parameter, and model parameters are selected, you can add a map by using one of the map commands in the edit mapping menu. It is often convenient to keep the clipboard viewer open so the list of model parameters available for mapping is always visible.

When a mapping is made, the mapped model parameter will be appended to the mapping list and the mappings column of the model. In this case the parameter Ef has been mapped to x.

Model Centered Mapping Process In a model centered mapping process, the mapping tool is used to first pick model parameters that you want to work with and then map them to relation parameters. The process involves the following steps.

- Selecting the model
- Selecting a model parameter
- Selecting a relation parameter
- Adding the map

Selecting the Model The first item in the mapping tool GUI model/relation/interface combination box is the model associated with the mapping tool. Select the model from the list so that selected model parameters may be mapped to selected relation interface parameters.

Selecting Model Parameters The mappings for combination box in the mapping tool GUI contains a list of all of the model's parameters. Any parameter can be selected from the list so that it may be mapped to selected relation interface parameters. In this case the parameter Nuf is being selected.

Selecting a Relation Parameter In order to select a relation parameter for mapping use the copy command in the edit menu. This may also be done from either the relation edit menu or the model edit menu. In this case the relation parameter x is selected.

Adding the Map Once the model, model parameter, and relation interface parameter are selected, you can add a map by using one of the map commands in the edit mapping menu. It is often convenient to keep the clipboard viewer open so the list of model parameters available for mapping is always visible. When a mapping is made, the mapped model parameter will be appended to the mapping list and the mappings column of the model. In the example below the parameter Ef has been mapped to x.

Mapping Shortcuts There are convenient shortcuts for common mapping cases available in the add menu.

1. Map: mapping an existing model parameter(s) to an existing relation interface parameter(s) or existing model parameter(s).
2. Add and Map: mapping an existing model parameter to a relation and creating a corresponding relation interface parameter in the same step.

Simultaneously Add and Map Parameters You may map an existing parameter to a newly created parameter in the same step using the add and map option in the add menu.

- Use the edit menu to copy a parameter of interest into the clipboard. In the figure below the parameter Ef was copied.
- Apply the add and map->last selection command in the add menu and a new parameter will be created and mapped to the original selection. In the figure below, a procedural relation was selected so that a new relation parameter was created (Ef1) and mapped to the parameter Ef.

Mapping Existing Parameters You may map an existing parameter(s) to another existing parameter(s) using the map option in the add menu.

- Use the edit menu to copy a parameter of interest into the clipboard. In the figure below the parameter Ef was copied.

- Select the icon of parameter that you want to map to, and then apply the map->last selection command in the add menu. In the figure below the relation parameter RealParameter was highlighted and then Map->last selection was chosen. Note that Ef now appears in the mapping column for RealParameter.

Adding Context Context are used to organize models in build view. Details of how to work with context are in the building context section. To add context to a DOME model...

- Make sure the model's add menu is available by selecting the list visualization build view in the definition tab.
- Select Add->Context from the add menu.

A reference to the context will be inserted into the model list visualization. Its location is determined by what is selected when it is added. If you open the context GUI in a new window, the DOME model menu remains available for model building. References to context may be removed or they may be deleted from the model using the DOME model edit menu.

Adding Filters In the future, it will be possible to add filters to automatically organize model objects in a specific way. At present filters are only available in predefined views and interfaces. Details of different filter types filters are in the building filters section.

Adding Interfaces Interfaces are used to define what elements of the DOME model will be exposed for users when the model is deployed. Interfaces are adding to a model using the interfaces tool , which is accessible through the tools menu.

Testing DOME Models This feature is not yet available. In order to test execute a DOME model and bring all parameters into a consistent state select the test option in the DOME Model menu. Further details are provided in the interface testing section. It is also possible to test execute relations.

Examples: DOME Models This documentation will be available soon.

Plugin Models This documentation will be available soon.

Abaqus Models

Adams Models

CATIA Models

Excel Models

IDEAS Models

MATLAB Models

Mathematica Models

Maxwell Models

Name Value batch Models

NASTRAN Models

SolidWorks Models

Unigraphics Models

Message Log A single message log is associated with every model. It is usually opened by a message log button in the upper right of the model GUIs (see the GUI section of the specific model type you are working on). In build mode, the message log is used to provide information or notification of events that do not merit interrupting users with a pop-up dialog.

TODO: Add color reference table

Clipboard The clipboard viewer is opened from the build mode windows menu. The clipboard is a buffer that records all object copy selections (made using the model's edit menu). Each copy selection is placed in a numbered folder. You can choose to paste copies of clipboard selections into any model (using the model's edit menu), provided the model support all objects in the selection. The delete button removes selected elements from the clipboard (not the model), while the empty button clears the clipboard.

Mapping Tool Each model has a mapping tool. Mappings are used to connect parameters in a DOME model to relations or parameters in a model to interfaces. The mapping GUI opens in a standalone window (accessible through the model's tool menu or by double clicking on a mapping field in the model's list visualization).

There are also menu shortcuts for creating new mappings described in the DOME model mapping parameters and interface mapping sections, but the mapping tool must be used to remove existing mappings.

Mapping Tool GUI The mapping tool GUI contains three elements.

- the model/relation/interface combination box (to select a relation, model, or interface to be mapped)
- the mappings for combination box (to select a parameter within the chosen model, relation, or interface for which you want to see mappings)
- the mapping list area (which lists all parameters mapped to the selected parameter).

Mappings are committed as soon as they are made. If you want to delete a mapping use the delete command in the edit mapping menu.

Model, Relation, Interface Selector The model/relation/interface combination box allows you to select the view point from which you want to make mappings. For example, if the model is chosen as shown in below, all model parameters will be listed in the mappings for combination box. If a relation is chosen, the relation's parameters will be listed in the mappings for combination box. If an interface is chosen, the interface's parameters will be listed in the mappings for combination box.

Mappings for Selector The mappings for combination box provides a list of all parameters for the selected model/relation/interface. When a parameter is selected from this list, parameters mapped to it will appear in the mappings list. In this case the model parameter Ef is mapped to relation parameters IEf, tEf, and fEf.

Mapping List The mapping list shows all parameters mapped to the parameter selected in the mappings for combination box. Mapped parameters may be added or deleted from the list using the edit mapping menu.

Edit Mapping Menu The Edit Mapping Menu appears in the DOME menu bar when the mapping tool window is selected.

Menu item	Function
Map last selection	Maps the last copy-selected parameter to the parameter that is active in the mappings for combination box and adds an entry in the mapping list.
Map from clipboard	Opens a dialog showing previous copy selections in the clipboard. After picking the desired parameters from the clipboard, they will be mapped to the parameter that is active in the mappings for combination box and added to the mapping list.
Clear selection	.Unselects all selected parameters in the mapping list.
Select all	Selects all parameters in the mapping list.
Delete	Deletes mappings to the selected parameters in the mapping list.

Model Versioning Each time a model is saved, a new version number is included in the model file. This version number may be seen in the xml view but is not editable by users. The internal version number is used by version control mechanisms for checking deployed models out from servers for editing and redeploying updated versions of the edited models back onto a server.

Model Checkout Model checkout is used to retrieve any previously deployed model from a server, provided you have appropriate model editing privileges. In order to check a model out for editing...

- select the checkout model option using the build menu
- login to the server from which you want to check a model out
- select the desired model on the server
- specify location where the checked out files should be placed in your own file space
- open the model file for editing

Once you have edited the model, you can check the changes back into the server using the redeploy option in deploy mode.

Model Checkout: Select Model Option To checkout a model, select the Checkout model option in the build menu, as shown below. This will allow you to checkout any type of model.

Model Checkout: Login to Server You will be prompted to login onto the server where the model files that you want to checkout are deployed. The server may be located anywhere in the world, and you may checkout visible model files provided you can log into the server and model deployers have given you editing permissions on the model file.

Model Checkout: Select Model Once you have logged into a server, you may navigate the server file space using a browser. Locate and select the model that you wish to checkout. Only models for which you have editing permissions will be shown in the browser.

Model Checkout: Specify Location After selecting the model on the server, you will be prompted to pick a location on your local computer, to which the checked out model file will be downloaded. Any extra files associated with the model (such as custom GUIs, 3rd party model files) will also be saved in this location. On the Mac OSX, you will need to enter a name for the model file.

Model Checkout: Open File Once all of the necessary files have been downloaded, you will be asked if you want to open the file. You may open the file now, or open it later using the standard open option in the build menu.

Model Visualizations The visualization combination box in all model GUIs is used to display model contents using different types of visualizations.

- the list (or tree table) visualization has a slightly different structure depending on the model type, so they are described in the GUI/definition tab section of the documentation for each model type. This is the visualization used for editing models.
- the graph view can be used gain insight into the causal structure of models.
- the DSM (Design Structure Matrix) view can be used to easily view what depends on what within a model
- the XML view can be used to inspect the XML code that defines the model.

Graph Visualization All models support a graph visualization that can be selected using the visualization combination box. This view is useful for exploring the causal flow between parameters. For comparison, the graph below is from the same model shown in list view in the main model visualization page. All objects are shown with both their icon and their name. Instructions for manipulating the graph are available at <http://www.touchgraph.com/>. Click on the image for the dynamic graph layout. Editing is not supported in this view. Also, there is a problem with right mouse controls for Mac OSX that has not been addressed. Therefore, right mouse options are not available on Macs.

Matrix (DSM) Visualization All models support a Design Structure Matrix (DSM) visualization that can be selected using the visualization combination box. This view is useful for identifying what parameters depend on what, and in determining the order in which parameter data will be updated when a model is executed in run mode. For comparison, the DSM below is from the same model shown in list view in the main model visualization page.

The first figure below shows the DSM in an unsorted form. The first row may be read as areaR1 is calculated using widthR1 and height R2. The first column may be read as arealR1 is used in calculating areaR2. The model is shown at the bottom of the page after the sort button has been pressed. Sorting algorithms are used to create a lower triangular matrix. This indicates the order in which items will updated when the model is solved at run time. If the model cannot be lower triangularized there will be dependencies shown in red above the diagonal. If the sorted matrix has such entries it means that there is a causal loop in the model. Run-time solving currently does not support such structures but this will be addressed in the neat future. Editing is not supported in this view.

XML Visualization All models support an XML visualization that can be selected using the visualization combination box. This view displays the model's XML definition. For comparison, the XML below is from the same model shown in list view in the main model visualization page. This view currently does not support editing, but support for text editing is envisioned (but not a high priority). If you have made changes to the model since it was last saved, the refresh button should be used to get an updated version of the XML.

Examples: Creating Models This documentation will be available soon. It will be a master directory to the example section for each model type.

Model Objects This section describes model objects that can be added to models, what they are for and how to work with their GUIs.

- Parameters represent different kinds of data.
- Relations define how different parameters affect each other.
- Context are used to organize objects in a model.
- Filters automatically generate predefined object organizations.

- Visualizations graphically display data from parameters in forms such as charts or graphs.

Information about what objects are available within different types of models and instructions on how to add them into models is in the models section.

Building Parameters Parameters are generic shells for data that support a number of different data types. Parameters are added to models using add menu associated with model definition tabs. Parameters available for each type of model are listed in the menu bar documentation for the given model type.

All model add menus are structured so that parameters can be added with a specified data type. However, one can change a parameter's data type after it has been added to a model using its data type chooser.

A parameter may be defined as a constant, meaning that its data value may not be changed during run time. Additionally, constraints may be placed on data type values. Parameters also support documentation.

Parameter GUI A parameter GUI is shown below with its definition tab selected. Parameter GUIs open in a standalone window. When the definition tab is selected the parameter's data type editor is available.

- The name field is used to edit the parameter's name. Text field editing commands are available.
- The combination box to the right of the parameter name indicates its data type. You can use this combination box to change the data type.
- The main panel of the definition tab is that data-type editor, which varies for each data type. Within this area there will be a button that opens a constraint editor for the data type.
- Single clicking on the constant box (in the lower right of the GUI) toggles the parameter's data type between constant or variable.
- The documentation tab is used to access the parameter's documentation editor.

Parameter Data Types Data types supported within parameters include those listed in the table below. A parameter's data type is set using the data type chooser.

TODO: Add table

Boolean The Boolean data type is a logical type that can have the state of either true or false. Boolean data types support a number of methods and operators that can be used in relations.

The Boolean editor panel, which appears in the parameter GUI definition tab, is used to set the data type's value. The combination box sets the value to true or false.

It is also possible to set the value of the Boolean data type using the value column in a model's list/tree-table view. The example below is in a DOME model.

Boolean Methods and Operators Boolean methods and operators are used to write Python expressions that manipulate parameters in relations that support custom body definitions. A brief summary is provided on this page while more detailed explanations are available through the more details links. A Boolean parameter is considered true if its value is true, false if its value is false. There are two predefined constants of type Boolean: true and false. For other data type methods and operators see the method and operators page.

Methods:

TODO: Add tables

Logical Operators:

Type Conversions:

Comparitors:

Boolean Methods The Boolean data type supports a copyFrom method and a duplication method. The copyFrom method should be used (not the = operator) only if you require internal relation parameters to maintain types matching the relation interface parameters. A summary of other Boolean data type methods, operators and functions are on the methods and operators page.

TODO: Add Table

Boolean Logical Operators The Boolean data type supports some simple logical operators. A summary of other Boolean data type methods, operators and functions are on the methods and operators page.

TODO: Add Table

Boolean Conversions The Boolean data type supports a number of conversions to other parameter data types. A summary of other Boolean data type methods, operators and functions are on the methods and operators page.

TODO: Add table

Boolean Comparitors The Boolean data type supports a number of logical comparison operations. A summary of other Boolean data type methods, operators and functions are on the methods and operators page.

TODO: Add table

Enumerated The Enumerated data type is used for lists of name/value pairs. Enumerated data types support a number of methods and functions that can be used to in relations. Enumeration elements are indexed starting at zero. The Enumerated data panel, which appears in the parameter GUI definition tab, is used to select an element from a list of name/value pairs. The edit button opens the enumeration editor, which is used to add or remove elements.

It is also possible to set the selected element using the value column in a model's list/tree-table view. The example below is in a DOME model.

Enumeration Editor The enumeration editor is used to add or remove elements from the enumeration. It is opened using the edit button on the enumeration data type GUI. When an element is added to the enumeration, the value of the new element will have the type specified by the type combination box. Integer, real, Boolean, and string values are supported. These are Java primitive types, not DOME3 parameters. Each element in the list must be given a unique name. Any quantity can be assigned to the enumeration's value.

Enumerated Methods and Functions Enumerated methods and functions are used to write Python expressions that manipulate parameters in relation types supporting custom body definitions. A brief summary is provided on this page while more detailed explanations are available through the more details links. An Enumerated data type is considered true if it contains elements, false if it contains no elements. Elements are indexed starting from 0. For other data type methods and operators see the method and operators page.

Methods:

TODO: Add tables

Functions:

Comparitors:

Enumerated Methods The Enumerated data type support a number of methods as listed below. A summary of other Enumerated data type methods, operators and functions are on the methods and functions page.

TODO: Add table

Enumerated Functions The Enumerated data type supports a number of functions. A summary of other Enumerated data type methods and functions are on the methods and functions page. The first element in an enumeration is indexed with index 0.

TODO: Add Table

Enumerated Comparitors The Enumerated data type supports a number of logical comparison operations. A summary of other Enumerated data type methods and functions are on the methods and functions page.

TODO: Add Table

File The File data type contains a path to a file on the local machine. File data types support a number of methods and operators that can be used in relations. The file editor panel, which appears in the parameter GUI definition tab, is used to set the file path. The file type combination box sets a filter on the file chooser. Once you pick a file of a given type using the choose button, the combination box is forced to match the selected file's type and is disabled. If show file in browser is checked, the file will be opened in the browser application designated for the chosen file's type. File data types support constraints.

It is also possible to set the file using the value column in a model's list/tree-table view. The example below is in a DOME model. When you click on the value cell the editor, a file choose... button becomes available.

File Methods and Operators File methods and operators are used to write Python expressions that manipulate parameters in relation types supporting custom body definitions. A brief summary is provided on this page while more detailed explanations are available through the more details links. A File parameter is considered true if its size is greater than zero, false if its size is zero or the file is null. For other data type methods and operators see the method and operators page.

Methods: TODO: Add Tables

Functions:

Comparitors:

File Methods The File data type supports a copyFrom method and a duplication method. The copyFrom method should be used (not the = operator) only if you require internal relation parameters to maintain types matching the relation interface parameters. A summary of other File data type methods, operators and functions are on the methods and operators page.

TODO: Add table

File Comparitors The File data type supports a number of logical comparison operations. A summary of other File data type methods, operators and functions are on the methods and operators page.

TODO: Add Table

File Constraints File constraints are not yet supported.

Integer Number The integer number is a data type supporting a single positive or negative whole number of up to 9 significant figures. Integer numbers also support units. Integers support a number of methods and operators that may be used to manipulate their value in relations. The integer editor panel (shown below), appears in the parameter GUI definition tab and is used to set the value. When you make a value change the background will show stale color until the change is committed. If you try to commit an invalid value, the background of the value field will maintain the stale color, indicating that the change has not been accepted. The unit combination box on the right displays the unit assigned to integer. The change... option in the combination box allows you to change the value's units. The constraints button is used to define constraints on the integer value.

It is also possible to set the integer value and unit using the value column in a model's list/tree-table view. The example below is in a DOME model. When you click on the value cell the editor, a value editing field and a unit combination box becomes available.

Integer Methods and Operators Integer number methods and operators are used to write Python expressions that manipulate parameters in relations that support custom body definitions. A brief summary is provided on this page while more detailed explanations are available through the more details links. An integer parameter is considered true if its value is non-zero, false if its value is zero. For other data type methods and operators see the method and operators page.

Methods: [TODO: Add Tables](#)

Math Operators:

Math Functions:

Type Conversions:

Comparators:

Integer Methods The integer data type supports a copyFrom method and a duplication method. The copyFrom method should be used (not the = operator) only if you require internal relation parameters to maintain types and maintain units matching the relation interface parameters. A summary of other integer data type methods, operators and functions are on the methods and operators page.

[TODO: Add tables](#)

Integer Math Operators The integer data type supports a number of basic math operators for writing relations. These operators support automatic unit conversion functionality. A summary of other integer data type methods, operators and functions are on the methods and operators page.

[TODO: Tables](#)

Integer Math Functions The integer data type supports a number of basic math functions for use in relations. A summary of other integer data type methods, operators and functions are on the methods and operators page.

[TODO: Tables](#)

Integer Type Conversions The Integer data type supports a number of conversions to other parameter data types. A summary of other Integer data type methods, operators and functions are on the methods and operators page.

[TODO: Tables](#)

Integer Comparitors The integer data type supports a number of logical comparison operations. A summary of other integer data type methods, operators and functions are on the methods and operators page.

[TODO: Tables](#)

Integer Constraints Integer constraints are not yet available.

List The List data type is used to create lists of DOME parameters. The data type also supports a number of methods and functions that can be used to in relations. Lists are indexed starting at zero. The List data panel, which appears in the parameter GUI definition tab, is used to add or delete parameters in a list. The type selected in the combination box determines what type of parameter is added to the list. The constraint button opens the list constraint editor.

It is also possible to perform all list editing from a model's tree/table view (a DOME model is shown below). Parameters may be added, deleted or removed from this view using functionality available in the edit menu. Additionally, parameter references may be added to a lists using this view.

List Methods and Functions List methods and functions are used to write Python expressions that manipulate parameters in relation types supporting custom body definitions. A brief summary is provided on this page while more detailed explanations are available through the more details links. A List data type is considered true if it contains elements, false if it contains no elements. Elements are indexed starting from 0. For other data type methods and operators see the method and operators page.

Methods: TODO: Tables

Functions:

Comparitors:

List Methods The List data type support a number of methods as listed below. A summary of other list data type methods, operators and functions are on the methods and functions page.

TODO: Tables

List Functions The List data type supports a number of functions. A summary of other list data type methods and functions are on the methods and functions page. The first element in a list is indexed with index 0.

TODO: Tables

List Comparitors The List data type supports a number of logical comparison operations. A summary of other list data type methods and functions are on the methods and functions page.

TODO: Tables

List Constraints List constraints are not yet available.

Matrix The Matrix data type supports two-dimensional matrices of primitive integer or real numbers. Matrices support units and methods and operators that may be used in relations. Matrix elements are indexed from 0. The matrix editor panel (shown below), appears in the parameter GUI definition tab and is used to set matrix dimensions, units, and the value of elements. The number of rows and elements may be edited directly in the row/columns fields, or using the add or delete buttons. If the fix size checkbox is selected it is not possible to change the dimensions of the matrix. Element values may be typed directly into the matrix table or many elements may be selected and changed to a common value using the fill button. The type combination box in the upper right of the GUI controls whether matrix elements are real numbers or integers. The unit combination box is used to assign a unit to the matrix and the constraint button opens the matrix constraint editor.

It is also possible to change the size and unit of a matrix in a model's value column. The example below is in a DOME model. When you click on the value cell an editor with row/column editing fields and a unit combination box become available.

Matrix Add Dialog The Matrix editor's add button is used to open the add dialog. The dialog can be used to insert multiple rows or columns into the matrix in a specific location. Initial values for the row and column elements may also be specified. The default initial value is specified by the value set in the fill dialog.

Matrix Delete Dialog The Matrix editor's delete button is used to open the delete dialog. The dialog can be used to delete multiple rows or columns from a specific location in the matrix.

Matrix Fill Dialog The Matrix editor's fill button is used to open the fill dialog. The dialog is used to specify the initial value of new elements when rows or columns are added to the matrix. Additionally, if matrix elements are selected before opening the fill dialog, the fill value can be used to change the value of multiple existing elements in the matrix.

Matrix Methods and Operators Matrix methods and operators are used in Python expressions that manipulate parameters in relations. A brief summary is provided on this page and more detailed examples/explanations are available in the subsection links, or through the more details links. A Matrix parameter is considered true if it has a non-zero dimension, false if it has no dimension or is null. For other data type methods and operators see the method and operators page.

Methods: [TODO: Tables](#)

Operators:

Functions:

Type Conversions:

Comparators:

Matrix Methods The Matrix data type supports a copyFrom method and a duplication method. The copyFrom method should be used (not the = operator) only if you require internal relation parameters to maintain types and maintain units matching the relation interface parameters. A summary of other Matrix data type methods, operators and functions are on the methods and operators page.

[TODO: Tables](#)

Matrix Operators The Matrix data type supports a number of basic operators for writing equations. These operators support automatic unit conversion functionality. A summary of other Matrix data type methods, operators and functions are on the methods and operators page.

[TODO: Tables](#)

Matrix Functions The Matrix data type supports a number of functions for use in equations. A summary of other Matrix data type methods, operators and functions are on the methods and operators page.

[TODO: Tables](#)

Matrix Conversions The Matrix data type supports conversion to arrays. A summary of other Matrix data type methods, operators and functions are on the methods and operators page.

[TODO: Tables](#)

Matrix Comparitors The Matrix data type supports a small number of logical comparison operations. A summary of other Matrix data type methods, operators and functions are on the methods and operators page.

TODO: Tables

Matrix Constraints Matrix constraints are not yet available.

Real Number The real number is a data type supports a single, floating-point value in double precision with units and constraints. Scientific notation is permitted. The data type is supported with methods and operators that may be used to manipulate its value in relations. The real number editor panel (shown below) appears in the parameter GUI definition tab. When you make a value change the background will show stale color until the change is committed. If you try to commit an invalid value, the background of the value field will maintain the stale color, indicating that the change has not been accepted. The unit combination box on the right displays the unit assigned to the value. The change option in the combination box allows you to change the value's units. The constraint button is used to apply constraints to the data type.

It is also possible to set the real value and unit using the value column in a model's list/tree-table view. The example below is in a DOME model. When you click on the value cell the editor, a value editing field and a unit combination box becomes available.

Methods and Operators Real number methods and operators are used in Python expressions that manipulate parameters in relations. A brief summary is provided on this page and more detailed examples/explanations are available through the more details links. A Real number parameter is considered true if its value is non-zero, false if its value is zero. For other data type methods and operators see the method and operators page.

TODO: Tables

Real Methods The real data type supports a copyFrom method and a duplication method. The copyFrom method should be used (not the = operator) only if you require internal relation parameters to maintain types and maintain units matching the relation interface parameters. A summary of other real data type methods, operators and functions are on the methods and operators page

TODO: Tables

Real Math Operators The real data type supports a number of basic math operators for writing equations. These operators support automatic unit conversion functionality. A summary of other real data type methods, operators and functions are on the methods and operators page.

TODO: Tables

Real Math Functions The real data type supports a number of basic math functions for use in equations. A summary of other real data type methods, operators and functions are on the methods and operators page.

TODO: Tables

Real Type Conversions The real data type supports a number of conversions to other parameter data types. A summary of other real data type methods, operators and functions are on the methods and operators page.

TODO: Tables

Real Comparitors The real data type supports a number of logical comparison operations. A summary of other real data type methods, operators and functions are on the methods and operators page.

TODO: Tables

Real Constraints

Real parameter constraints are not yet available.

String The String data type is supports a single line of text characters of any length. Strings support a number of methods and operators that can be used in relations. The string editor panel, which appears in the parameter GUI definition tab, is used to set the content of the string and also to define constraints on the data type.

It is also possible to set a string's content using the value column of a model's list/tree-table view. The example below is in a DOME model.

String Methods and Operators String methods and operators are used in Python expressions that manipulate parameters in relations. A brief summary is provided on this page and more detailed examples/explanations are available through the more details links. A String parameter is considered true if it is not empty, false if it is empty or null. For other data type methods and operators see the method and operators page.

TODO: Tables

String Methods The String data type supports supports an equal method and a duplication method. The copyFrom method should be used (not the = operator) only if you require internal relation parameters to maintain types matching the relation interface parameters. A summary of other String data type methods, operators and functions are on the methods and operators page.

TODO: Tables

String Operators The String data type supports an addition operator. A summary of other String data type methods, operators and functions are on the methods and operators page.

TODO: Tables

String Functions The String data type supports supports a number of functions. A summary of other String data type methods, operators and functions are on the methods and operators page.

TODO: Tables

String Type Conversions The String data type supports a number of conversions to other parameter data types. A summary of other String data type methods, operators and functions are on the methods and operators page.

TODO: Tables

String Comparitors The String data type supports a number of logical comparison operations. A summary of other String data type methods, operators and functions are on the methods and operators page.

TODO: Tables

String Constraints

String constraints are not yet available.

Table

The Table parameter is not yet available.

Text This documentation is not yet available.

Vector The Vector data type supports primitive integer or real numbers with units. There are also methods and operators that may be used in relations. Vector elements are indexed from 0. The vector editor panel (shown below), appears in the parameter GUI definition tab and is used to set vector size, units, and the value of elements. The row and column radio buttons determine whether the vector is a column or a row. The number of elements may be edited directly in the elements field, or using the add dialog. The delete button removes selected elements. If the fix size checkbox is selected it is not possible to change the length of the vector. Element values may be typed directly into the vector table or many elements may be selected and changed to a common value using the fill button. The type combination box in the upper right of the GUI controls whether vector elements are real numbers or integers. The unit combination box is used to assign a unit to the vector and the constraint button opens the constraint editor.

It is also possible to change the size and unit of a vector in a model's value column. The example below is in a DOME model. When you click on the value cell an editor with an element number editing field and a unit combination box become available.

Vector Add Dialog The Vector editor's add button is used to open the add dialog. The dialog can be used to insert multiple elements into the vector in a specific location. Initial values for the elements may also be specified. The default initial value is specified by the the value set in the fill dialog.

Vector Fill Dialog The Vector editor's fill button is used to open the fill dialog. The dialog is used to specific the initial value of new elements that are added to the matrix. Additionally, if vector elements are selected before opening the fill dialog, the fill value can be used to change the value of multiple existing elements.

Vector Methods and Operators Vector methods and operators are used to write Python expressions that manipulate parameters in relations. A brief summary is provided on this page and more detailed examples/explanations are available through the more details links. A Vector parameter is considered true if it has a non-zero dimension, false if has no dimension or is null. For other data type methods and operators see the method and operators page.

TODO: Tables

Vector Methods The Vector data type supports a copyFrom method and a duplication method. The copyFrom method should be used (not the = operator) only if you require internal relation parameters to maintain types and maintain units matching the relation interface parameters. A summary of other Vector data type methods, operators and functions are on the methods and operators page.

TOOD: Tables

Vector Operators The Vector data type supports a number of basic operators for writing equations. These operators support automatic unit conversion functionality. A summary of other Vector data type methods, operators and functions are on the methods and operators page.

TODO: Tables

Vector Functions The Vector data type supports a number of functions for use in equations. A summary of other Vector data type methods, operators and functions are on the methods and operators page.

TODO: Tables

Vector Type Conversions The Vector data type supports conversion to arrays. A summary of other Vector data type methods, operators and functions are on the methods and operators page.

TODO: Tables

Vector Comparitors The Vector data type supports a number of logical comparison operations. A summary of other Vector data type methods, operators and functions are on the methods and operators page.

TODO: Tables

Vector Constrains Vector constraints are not yet available.

Data Type Chooser A parameter's data type may be changed using the change option in the parameter data type combination box, as shown below. This will open the data type chooser GUI. This functionality allows you to add parameters quickly and decide on their data type later, or you can change your mind about a parameter's data type without having to remove the parameter.

Data Type Chooser GUI The data type chooser GUI is used to pick a parameter's data type from a list of available options. To open the data type chooser use the data type combination box in the parameter GUI. When you change a parameter's data type, value information related to the old data type is lost unless there is a conversion between types. The data type chooser maintains program focus (it is a modal dialog), so you will not be able to select other DOME windows until it has been dismissed.

Units DOME will automatically convert units between related parameters to maintain unit consistency. All parameter data types that support units have a unit combination box in their editor panel. By default all data types supporting units initially have no units. Units can be chosen by selecting the change unit... option in the units combination box, which will open the unit chooser.

Additionally, there are a number of unitless constants defined under the constant category in the unit chooser. For example 1.0 with the unit pi is 3.14... It is also possible for to define custom units.

Units Chooser GUI The unit chooser is used to change the unit assigned to a data type. The unit chooser GUI is opened by selecting the change unit... option in a data type's unit combination box. The unit chooser, shown below, is operated working from left to right. The desired dimension for the data type is selected in the left-most column. Only one item can be chosen from this list. Then, unit options for the chosen dimension will appear in the units column on the right. One can filter the list of unit options according to measurement system using the combination box below the column.

User Defined Units The DOME units engine has been designed and implemented to allow model builders to define abstract or custom unit types, such as \$/kg, or kg/vehicle. This capability is not yet available.

Parameter Constraints Data types that support constraints will have a constraint button in their editor panel. Both hard and soft constraints may be defined in the constraint editor. In run mode, violation of a hard constraint will stop model execution, while violation of a soft constraint will generate a warning in the message log. Constraints are not yet available.

Conversion and Mapping Compatibility Currently only identical data types may be mapped to each other, with the exception of integer and real parameters. This, a Real parameter may only be mapped to another Real or Integer parameter with the same dimension (units will be converted). In all other cases on identical parameter types may be mapped to each other. It is intended to implement conversions between many of the different data types in the future.

Examples: Parameters This page contains links to examples showing how to use parameters and their methods.

TODO: Add table

Relations Relations are used to define interactions or dependencies between parameters. There are a number of relations specialized for different purposes. Relations are only supported in DOME Models and integration models (iModels). All relations have parameters (relation interface parameters) upon which the relation performs its computation. When model parameters (or parameters from other relations) are mapped to a relation's parameters, changes within a relation will propagate to the mapped parameters outside of the relation. Relations support documentation and constraints. Additionally, relations automatically perform dimensional analysis on computation involving data types that support units. The relation GUI section describes how to access relation definition functions in build mode.

Relation body Editor The relation body editor is the bottom panel in the relation GUI. The example below shows the body definition in a procedural relation. The body editor is present only in relations that support a custom body definition. Specifics are in the documentation for each relation type. The body editor is used to define mathematical or logical relationships between relation parameters using a Java implementation of the Python programming language. A custom Python editor provides a number of convenient features for defining relations. All relation types requiring a user-defined relation body also require their internal causality to be defined. Care must be taken to ensure that this causality definition is consistent with the code in the relation body. Additionally, it is important to have a basic understanding of how types and units are treated within a relation. ToDo: add screen shot

Data Types Within Relation Body Relation interface parameters will always be of the type (e.g., Real or Integer) specified by the person who defined the relation. However, during execution of the Python relation body, the type of internal parameters may actually change to other types. This is because Python is not a typed language and the internal Parameter types are determined based on the data in an assignment. Consider the following example. A procedural relation has the interface parameters...

TODO: Tables

and the Python body of the relation is:

```
c = a / b
d = c
```

When the relation body executes, new internal real parameter objects are created in Python. The objects are assigned types and values as they were specified in the relation interface. Then, the first line of the code is executed ($c = a / b$). The Integer “/” operator will divide their values and return a Real object with a value of 1.5. The = assigns the returned object to c. Therefore, the internal c is a Real Parameter (not an Integer) with a value of 1.5. Next, the second line will be executed ($d = c$). Thus, c is assigned to d. After the second line has executed, the internal d is a Real Parameter with a value of 1.5. Finally, the output interface parameters are updated.

- The interface parameter c is an Integer, so it is assigned the value 1 (from the Int() of the Real internal parameter c).
- The interface parameter d is a Real, so it is assigned the value 1.5 (from the Real internal parameter d).

The example shows that the derived parameter types may vary from the units of the external interface parameters.

TODO: Tables

and the Python body of the relation is: `c.copyFrom(a / b)` `d.copyFrom(c)` Once again, when the first line of the code is executed (`c.copyFrom(a / b)`), the Integer “/” operator will divide their values and return a Real object with a value of 1.5. However, unlike the = assignment (which assigned the returned object to c) the copyFrom method converts the result to an Integer parameter and copies the value 1 into c. Thus, the internal parameter c is an Integer, matching the type of its corresponding relation interface parameter. Thus, when the second line executes, c will be converted to a Real type and the resulting value will be copied into d, leading to the final output results being:

- The interface parameter c is an Integer with a value of 1.

- The interface parameter `d` is a Real with a value of 1.0.

TODO: Tables

Python Editor Features The relation body editor provides color highlighting according to Python syntax. Additionally, all DOME interface parameters names are highlighted. The relation editor does not do automatic indentation, so the user must insert tabs according to Python syntax. Other editing features are listed in the table below.

TODO: Tables

Python References The Open Book Project makes available The Python Bibliotheca. Included is the Python version of Allen Downey's book How to Think Like a Computer Scientist in an interactive form where you can read the book and type Python code to try it out at the same time. Standalone Python reference documentation is also online, but we find this style of documentation a bit difficult to use. A good book for learning the basics of Python quickly is The Quick Python Book by Daryl D. Harms and Kenneth McDonald. A good detailed reference book for Python is the Python Essential Reference (2nd Edition) by David M. Beazley and Guido Van Rossum.

Units Within Relation Body Relation interface parameters can have units. Thus you may specify the units of interface parameters. Unit conversion and dimensional analysis is used to guarantee that all inputs enter the relation with the units specified, and leave the relation with the desired units and dimensions. However, during execution of the Python relation body (in relation types that support custom body definitions) the units of internal parameters may be converted to other dimensionally consistent units. Consider the following example. A procedural relation has the interface parameters...

TODO: tables

and the Python body of the relation is:

```
c = a + b
d = c + 1.0
```

When the relation body executes, new internal real parameter objects are created in Python. The objects are assigned values and units as they were specified in the relation interface. Then, the first line of the code is executed (`c = a + b`). The Real "+" operator will convert `a` and `b` into the same units (let's say cm), add their values, and return a Real object with a value of 5.0 and units in cm. Last, the `=` assigns the returned object to `c`. Therefore, once the first line has executed, the internal `c` is 5.0 cm. Next, the second line will be executed (`d = c + 1.0`). The Real "+" operator assumes that the primitive double data type has the same units as `c`, so a Real object is returned with a value of 6.0 and units in cm. Then, the `=` assigns the returned object to `d`. Therefore, after the second line has executed, the internal `d` is 6.0 cm. Finally, the output interface parameters are updated.

- 5.0 cm is converted to inches, and the interface parameter `c` is assigned 2.0 inches.
- 6.0 cm is converted to inches, and the interface parameter `d` is assigned 2.4 inches.

The example shows that dimensional consistency of derived parameters is maintained within the relation, but the units used internally may vary from the units of the external interface parameters.

TODO: Tables

If, for some reason, you must know the units of derived internal relation parameters during relation execution, or you must combine primitive data types such as doubles or integers and DOME parameters with units, use the `.copyFrom` method described in the methods and operators section. Do not use the `=` assignment! The same example is now repeated below using the Real parameter `.copyFrom` method instead of the `=` assignment. A procedural relation has the interface parameters...

TODO: Tables

and the Python body of the relation is: `c.copyFrom(a + b)` `d.copyFrom(c + 1.0)` Once again, when the first line of the code is executed (`c.copyFrom(a + b)`), the Real "+" operator will convert `a` and `b` into the same units (cm), add their

values, and return a Real object with a value of 5.0 and units in cm. However, unlike the = assignment (which assigned the returned object to c) the copyFrom method converts the result to 2.0 inches (from 5.0 cm), and sets the value of c to 2.0. Thus, the internal parameter c retains the units of inches specified in the relation parameter interface. Thus, when the second line executes, c will be in inches, leading to the final output results being

- The interface parameter c is 2.0 inches.
- The interface parameter d is 3.0 inches.

TODO: Tables

Relation Causality Tab The Causality tab of a relation GUI is shown below. This panel shows how relation parameters depend upon each other. The edit button is used to define causality in relations that support custom body definitions. The edit button is disabled in relations that do not support custom definitions. A matrix visualization shows what-parameters-drive-what within the relationship. In the example below assume that the relation has interface parameters a, b, c, d, e. Assume the relation body definition equations are:

$$c = a + b \quad e = c + d$$

Only interface parameters that are relation output or indeterminate have a row in the visualization matrix. All interface parameters with outputs depending upon them are associated with a column. Thus, for example, one can read that c depends upon a and b by reading across the row labeled c, and e depends upon c and d by reading across the row labeled e.

Some relation types have a predefined causal structure, while other types require you to specify dependencies between interface parameters. Specific requirements are in the documentation of individual relation types. When working with relations that require you to provide a causality definition, the edit causality information button, as appears above, is available to provide access to the causality editor GUI.

Causality Editor GUI The internal causality editor displays a square matrix with a row and column for each relation parameter. Check boxes in the matrix are used to indicate dependency relationships. The causality definition does not automatically mirror the structure of the relation body. Care must be taken to ensure that the body of the relation and the relation causality definition are consistent. DOME needs this information to correctly solve sets of relations, so care must be taken. The matrix is read in rows. As one moves across a row, one should put checks in the columns that correspond to parameters that the row variable depends upon. The example below assumes that the relation has interface parameters a, b, c, d, e, and that the relation body definition equations are:

$$c = a + b \quad e = c + d$$

The causality editor matrix should be completed as shown below. One can read that c depends upon a and b by reading across the row labeled c, and e depends upon c and d by reading across the row labeled e.

The causality editor is modal, meaning that editor must be closed using either the Ok or cancel buttons before other model windows may be selected.

Relation Interface Parameter Tab The relation interface parameter tab is at the top of the relation GUI. An example for an equal relation is shown below. Relation interface parameter icons are blue. Relations perform their computation upon its interface parameters. The parameter tab is used to set names, default values, and documentation, etc. for interface parameters. The relation parameters may also be edited in a model's list view. The view combination box on the right of the interface parameter editor toggles between two views: input/output/local indeterminate; and model causal. Relation parameters are automatically placed in these views depending upon the relation's internal causality and how relation parameters are mapped to other parts of the model. The up/down arrows on the right of the editor may be used to reorder interface parameters within filters. Within a single relation, all interface parameters must have unique names. They have the same GUI as model parameters and can be configured in a similar manner. The values of these parameters will be used as a defaults in the relation if they are not mapped (or connected) to model parameters. Each relation type has an add and edit menu that is available when the relation GUI is in focus for adding and deleting interface parameters.

Model Causality View The model causal view of relation parameters contains four filters: independents; intermediates; results; and indeterminates. This view presents interface parameters from an overall or global model causality viewpoint, unlike the input/output/local indeterminate view, which limits its scope to the relation's internal causality. This view is useful for seeing what input interface parameters or local indeterminate interface parameters are actually dependent parameters from a global model viewpoint. The example relation below is also shown in the relation causal view example. In this case all of the relation input parameters are driven by external variables through mappings. Thus, even though they are inputs from the relation's viewpoint, they are intermediate variables from the overall model viewpoint.

Relation Causality View The input/output/local indeterminate view of relation parameters contains up to three filters. This view is useful for reviewing the internal causal structure of a relation independent of how the relation may be mapped in the model.

- The input filter shows interface parameters that are independent within the scope of the relation. Inputs may be changed by external parameters.
- The output filter shows interface parameters that are computed (intermediate variables or results) by the relation as a function of its inputs. Outputs may be used but not changed by external parameters.
- The local indeterminate filter shows interface parameters that do not have fixed causality based upon the internal structure of the relation. That is, interface parameters which have the potential to be both inputs and outputs are placed in this filter.

Since relations are specialized for different purposes, not all relations will have all three filters. Only relevant filters are included in the editor for each relation type. The example relation below is the same relation that is shown in the model causal view.

Relation Types There are different types of relations specialized for different internal computational solving or with predefined operations or behaviors. This section details the purpose of different relation types and how to use their editor GUIs. Types supporting user-defined relation bodies and custom behavior.

TODO: Tables

Procedural Relation Procedural relations are so-named because you must write procedural code in the body of the relation. Procedural code is a recipe that provides instructions in the exact order needed to correctly solve interactions between input and output interface parameters. There are no external solvers. The procedural relation gathers values from its input interface parameters, executes its internal program, and once completed, updates the value of its output interface parameters. Procedural relations are the most flexible type of relationship because complete software programs can be written within the relation body. The procedural relation editor GUI supports the three steps required to fully define a procedural relationship:

- Adding interface parameters
- Defining the relation's internal causality
- Defining the relation body

Procedural Relation Menus The Procedural Relation context Menus are available when its GUI is open and in focus. The Edit Procedural Relation Menu and Add Menu are used to create the relation's interface. TODO: add screen shot

Procedural Relation Add Menu The Add menu appears only when the definition tab of a procedural relation GUI is selected. Add menu options apply only to relation interface parameters. TODO: Screen shot of the model add menu

Edit Procedural Relation Menu The Edit Procedural Relation menu appears only when the definition tab of a procedural relation GUI is selected. The edit menu options apply only to the relation interface parameters. ToDo: add screen shot of the edit menu

TODO: Tables

Adding Interface Parameters Procedural relation interface parameters are added using the procedural relation add menu. When added, references to interface parameters will always initially appear in the input filter of the editor GUI. Once the causality of the relation has been defined, dependent interface parameters will move to the output filter. Interface parameters may also be added directly in the DOME Model list visualization.

Adding Mappings The process for mapping procedural relation interface parameters to model parameters is outlined in the DOME Model mapping parameters section. Additionally, mappings may be defined using the procedural relation edit menu. Procedural relation mapping restrictions include:

- Only compatible parameter data types may be mapped to each other.
- Only one model parameter may be mapped to an input interface parameter. However, a single output interface parameter may be mapped to multiple model parameters.

Procedural Relation Constraint Definitions This documentation will be available soon.

Defining Internal Relationm Causality Procedural relations require the users to define the causal relation between input and output interface parameters. Otherwise, all interface parameters will be seen as inputs. Care must be taken to ensure that the defined causality correctly matches the body definition. The internal causality tab of the procedural relation editor GUI provides access to the causality editor.

Defining the Relation Body The body definition panel of the procedural relation editor GUI provides access to the body editor. Procedural relations require users to write the body code that relates input parameters to output parameters. Methods, operators and functions of parameter data types are used in the Python code relation body. Before writing relations it is recommended that you review the sections discussing units and data types within the relation body. Relation body definition samples are provided in the examples section.

Procedural Relation Editor GUI The procedural relation editor is inserted into the main relation GUI, as shown below. Within the interface parameters tab there are two filters, one for inputs and one for outputs. The internal causality tab is used to access the relation causality editor. Procedural relations also require that you write the code it is execute in their body definition editor. ToDo: add screen shot

Examples: Procedural Relations This documentation will be available soon.

Equal Relations This documentation will be available soon. Must say which editors are available.

Iterator Relations The iterator relation is used to run loops in a DOME models. This documentation will be available soon. Must say which editors are available.

Twin Relations This documentation will be available soon. Must say which editors are available.

Relation Constraints Relations that support constraints will have a constraint button in body definition editor. Both hard and soft constraints may be defined in the constraint editor. In run mode, violation of a hard constraint will stop model execution, while violation of a soft constraint will generate a warning in the message log. Relations that support constraints include:

- Procedural Relation

Dimensional Analysis Dimensional analysis is performed on all relations involving parameter data types that support units. The remainder of this documentation will be available soon. Make a link to the section on units within a relation (in building relations/relation body editor)

Examples Examples of how to define the following relations are available.

- Procedural Relation

Mapping Relations Relation interface parameters must be mapped to model parameters if the relation's execution is to have external effects. The mapping process is described in the DOME Model mapping parameters section.

Relations GUI A procedural relation GUI is shown below with its definition tab selected. There is a generic relation GUI into which editors specific to different types of relations are inserted. When the definition tab or the relation GUI is selected (in build mode) the relation editor is available. Relation GUIs are opened in a standalone window from a model's list view.

- The name field is used to edit the relation's name. Text field editing commands are available.
- The main panel of the definition tab has three main components: the relation interface parameter editor; the relation causality editor; and the relation body editor. Which of these editors is available will vary with the relation type. Within this area there will be a button that opens a constraint editor for relations that support constraints.
- The documentation tab is used to access the relation's documentation editor.

Relation Menus

Each type of relation has its own context dependent add and edit menus for adding and editing relation interface parameters. Please see the relation-type specific documentation for details.

Testing Relations It is possible to test execute relations on their own without executing the entire models or going through the full deployment process. In order to test execute a relation and bring all parameters into a consistent state:

- Select the relation to be tested in the model
- Choose the Test Relation command in the model menu

The relation will then execute and make input/outputs consistent. Status information will be directed to the model's message log. TODO: add screen shot of the test option in the menu bar for A DOME model. Must mention that this is specifically for a DOME model as this is a more generic section.

Building Context Context objects are used to organize models and references to all model objects are allowed to be placed within it. Context support documentation. Typically all building activities that the given model supports (e.g., adding and deleting objects, mapping, etc.) are available within a context.

TODO: Tables

Context GUI The context GUI is shown below with its definition tab selected. The GUI is very similar to the DOME Model GUI.

- The name field is used to edit the context name.
- The left arrow (under the name label) is used to change scope in the visualization window.
- The text box (containedContext) shows the name of the scope at which the model is being viewed. This context is contained within the myContext.
- The visualization combination box is set to determine how the objects are visualized. The same visualizations as are available in the main model GUI will be available with the context visualization area. The DOME model list visualization is shown below.
- The up/down arrows on the right are used to re-order selected objects within the context.
- The documentation tab is used to access the context's documentation editor.

ToDo: update screen shot: context GUI.gif showing context named containedContext (have context within and be tunneled down).

Building Filters Filters are used to automatically organize model objects and are predefined. For example, the causal and object type views in DOME models are generated using filters. Since filters automatically determine their contents, they typically do not allow the direct addition of objects when building a model. However, they often support functions such as copy, delete, or mapping.

Filter Types A number of predefined filter types are currently available. In future releases it will also be possible for users to make custom filters.

- Parameters Filter
- Relations Filter
- Context Filter
- Visualization Filter
- Independents Filter
- Indeterminates Filter
- Intermediates Filter
- Results Filter
- Subscriptions Filter
- Input Filter
- Output Filter
- Local Interdeterminates Filter

Context Filter The context filter searches the model scope in which it is applied and creates references to all context objects that it finds. The filter GUI displays its contents using the list visualization view of the model GUI in which it resides. The filter is useful for determining what unique context are in a model.

Independents Filter The independents filter searches the model scope in which it is applied and creates references to all independent parameters that it finds. Independent parameters are those which are not mapped to the output of any relations. Thus, the value of their data may be changed directly by model users. The filter is useful for determining what independent variables are in a model. The filter GUI displays its contents using the list visualization view of the model GUI in which it resides.

Indeterminates Filter The indeterminates filter searches the model scope in which it is applied and creates references to all indeterminate parameters that it finds. Indeterminate parameters are parameters that cannot be deemed as either independent or dependent (driven by the output of relations). For example, a number of parameters in a twin relation will be indeterminate unless one of the twinned parameters is driven by another relation. The filter is useful for determining what indeterminate variables are in a model. The filter GUI displays its contents using the list visualization view of the model GUI in which it resides.

Intermediates Filter The intermediates filter searches the model scope in which it is applied and creates references to all intermediate parameters that it finds. Intermediate parameters are dependent parameters driven by the output of relations, but they are also inputs to other relationships in the model. Thus, the value of their data may not be changed directly by model users. The filter is useful for determining what intermediate variables are in a model. The filter GUI displays its contents using the list visualization view of the model GUI in which it resides.

Inputs filter The input filter searches the scope in which it is applied and creates references to all parameters that can be legally changed by actions outside of this scope. All inputs must be independent within the filter's scope, but may actually be dependent parameters from a global viewpoint if they are driven by a relation outside of its scope. The filter is useful for determining what variables may be legally changed by entities outside of the scope to which the filter is applied. The filter GUI displays its contents using the list visualization view of the model GUI in which it resides.

Local Indeterminates Filter The local indeterminates filter searches the scope in which it is applied and creates references to all parameters that are indeterminate from within its local scope. Thus, local indeterminate parameters cannot be deemed as either independent or dependent within the filter's scope, but they may actually be determinate from a global larger scope. The filter is useful for determining what parameters do not have a fixed causality within the local scope. The filter GUI displays its contents using the list visualization view of the model GUI in which it resides.

Outputs Filter The output filter searches the scope in which it is applied and creates references to all parameters that are results from actions within its scope. All outputs must be results or intermediate variables within the filter's scope, but that may actually be only intermediate parameters from a global viewpoint if they are used by a relation outside of its scope. The filter is useful for determining what variables will change as a result of changes in inputs. The filter GUI displays its contents using the list visualization view of the model GUI in which it resides.

Parameters Filter The parameters filter searches the model scope in which it is applied and creates references to all parameter objects that it finds. The filter GUI displays its contents using the list visualization view of the model GUI in which it resides. The filter is useful for determining what unique parameters are in a model.

Relations Filter The relations filter searches the model scope in which it is applied and creates references to all relations objects that it finds. The filter GUI displays its contents using the list visualization view of the model GUI in which it resides. The filter is useful for determining what unique relations are in a model.

Results Filter The relations filter searches the model scope in which it is applied and creates references to all relations objects that it finds. The filter GUI displays its contents using the list visualization view of the model GUI in which it resides. The filter is useful for determining what unique relations are in a model.

Subscriptions Filter The subscriptions filter searches the model scope in which it is applied and creates references to all interface objects that have been subscribed to from remote models. The filter GUI displays its contents using the list visualization view of the model GUI in which it resides. The filter is useful for determining what remote subscriptions are in a model.

Visualizations Filter The visualizations filter searches the model scope in which it is applied and creates references to all visualization objects that it finds. The filter GUI displays its contents using the list visualization view of the model GUI in which it resides. The filter is useful for determining what unique visualizations are in a model.

Filter GUI The filter GUI varies depending upon its type. The independents filter shown below uses the list visualization for the model in which it resides (in this case DOME model list visualization). Filters do not support user specified documentation. TODO: update screenshot of filter GUI.gif

Visualizations Visualization objects process data within parameters and present them in a unique manner, such as in plots or charts. The visualization object is not yet available.

Object Methods and Operators Object methods and mathematical operators are used to manipulate the values and other characteristics of objects in relations that allow users to write custom code in their body definition (such as the procedural relation, for example). Methods and operators are available for:

- Boolean
- Enumerated
- File
- Integer Number
- Matrix
- Real Number
- String
- Vector

Examples This page will be a directory pointing to the example directory for each of the object types. This documentation will be available soon.

Building Interfaces All models are executed through interfaces. Interfaces define what parameters and views of the model will be accessible by remote users. Models support multiple interfaces, so you can create different interfaces customized for different users of the same model. A default interface is automatically built for every model so that you can test or deploy a model without the step of explicitly defining an interface. All interfaces associated with a model are viewed, created and deleted using the interface manager, which is accessed through the model tools menu. The contents of interfaces are defined and mapped to model parameters using the interface editor.

Default Interface A default interface named “Default Interface” is automatically generated for every model. This allows you to test the model during the building process without having to take the time to manually define an interface before testing. It also allows you to deploy a model without defining an interface yourself, provided you are satisfied with the items that are automatically exposed in the default interface. The default interface is not user editable. However, in the interface manager, you can duplicate the default interface and use the duplicate as a starting point for defining other interfaces. The standard interface views in the default interface are populated as follows.

- The build view contains a flat list of all independent and result parameters in the model. The view only context exposes the entire model for browsing by model users.
- The interface causality view displays all independent model parameters in the input filter, and all model result parameters in the output filter.
- The system causality view will be the same as the interface causality view unless the interface has already been connected to other remote models.
- The model view reveals all objects in the underlying model. This will allow you to view the state of all model variables when the model is tested.

Interface Examples Documentation will be available soon.

Interface Versioning This documentation will be available soon.

Testing Interface This documentation will be available soon. Should be possible from both the Model Menu and the Interface Manager Interfaces Menu.

Interface Manager An interface manager is associated with each model. The interface manager lists all interfaces associated with a model and allows the addition, duplication, or deletion of interfaces using the interfaces menu bar. A non-user editable default interface will always be available in the list. The contents of individual interfaces are edited by double clicking on icons in the interface manager GUI. The interface manager is accessed through a model's tool menu. The figure below shows the tools menu for a Dome model. *ToDo: add screenshot (tools menu expanded)*

Interface Manager GUI The interface manager GUI is shown below. It is possible to expand interfaces to view their contents in the interface manager, but individual interfaces can only be edited by opening an interface's editor GUI. The list view follows general build mode GUI behavior. The default interface is automatically generated for all models. *ToDo: add screen shot of interface managers, with at least one interface expanded.*

Interface Manager Menu Bar The interfaces menu appears only when the interface manager is open and in focus. The interface menu options apply only to whole interfaces listed in the manager. The contents of individual interfaces are edited in the interface editor. *ToDo: add screen shot of the interfaces menu*

ToDo: Tables

Interface Editor The editor for a model interface is opened from the interface tool/manager. The interface editor allows you to add parameters to an interface and map them back to parameters in the model.

Adding Interface Parameters Interface parameters can be viewed as proxies linked to parameters in the model underlying the interface. When deployed, users are able to access, change, and write Dome Model relationships using interface parameters. The actual model is not exposed to users. All data types available within Dome Models are available in an interface (details of the different data types are in the building parameters section). There are two ways one may approach building an interface. You can build the interface first and then map it to the model, or you can start with the model and create mapped interface parameters in one step. Interface parameters may be deleted from the interface using the interface edit menu. *ToDo: Include green interface parameter icon.*

Building Interface After the Model If the model for which the interface is being created is already complete, interface parameters can be added and mapped to the model in one step. To add and map interface parameters in one step

- Copy the desired model parameter into the clipboard using the model's edit menu.
- Select the definition tab of the interface editor GUI.
- Make sure that either the build view or interface causality view is selected.
- Select Add and Map->last selection or Add and Map->from clipboard... from the edit interface menu.

ToDo: screen shot of interface with add and map menu expanded.

Building Interface Before the Model It is possible to add interface parameters and then map the interface parameters to the model at a later point in time. This can allow you to define and deploy a dummy interface for other to work with while the model underneath the interface is being completed. To add interface parameters

- Selecting the interface definition tab.
- Make sure that either the build view or interface causality view is selected.
- Select Add->desiredParameterType from the add menu.

When interface parameters are added in this way, they must be also be mapped to model parameters in a separate step if they are to affect parameters in the underlying model. If unmapped interface parameters are added in build view, they are assumed to be inputs and thus will appear in the input filter of the causality view. To add an unmapped output interface parameter, select the output filter in the interface causality view before using the add menu. Once interface parameters have been mapped to model parameters, the causality of the model will always determine whether interface parameters are inputs or outputs. ToDo: screen shot of interface with add menu expanded and a parameter selected.

Interface Editor GUI The interface GUI is shown below with its definition tab selected. When the definition tab is selected, the edit interface menu bar (for adding objects, editing, and mapping to model objects) is available.

- The name field is used to edit the name of the interface. The interface name can also be edited from the interface manager.
- The GUI combination box is used to assign an optional custom run-time GUI to the interface.
- The left arrow (beside build view) is used to change scope in the interface visualization panel.
- The text box (containing build view in the figure) shows the name of the scope at which the interface is being viewed. There are three standard interface views that organize the interface parameters in different ways. The interface is currently being shown at the scope of the entire build view. When in build view you can define the contents of the interface. There is also a fourth model view that can be used to directly expose parts of the model underlying the interface.
- The visualization combination box is set to list (tree-table).
- The up/down arrows on the right are used to re-order selected objects within a context.
- The values of interface parameters outside of the model view are user editable at run-time, provided that the changes do not violate the causality of the model. The interface causality view can be used to easily observe whether interface parameters are inputs or outputs.
- The documentation tab is used to access the interface's documentation editor. Parameters within an interface can import documentation from a mapped model parameter.

ToDo: add screen shot of interface editor GUI

Interface Visualizations The visualization combination box in the Interface GUI is used to display the interface contents using different visualizations. For example, the interface might be shown as a list or as text-based xml. The visualization combination box is not used to associate custom GUIs with an interface. Only list visualization is available at present.

Interface List Visualization The list visualization contains ... (this documentation will be available soon). The behavior of this view follows the standard DOME list/tree-table. Double clicking on a mapping field to open the relation mapper tool. Also need to document the editing of parameter data type values in this view and adding relation interface parameters through this GUI. Point out value and unit editor (when available) shown. For full data type editing and constraint definition, must work through the parameter GUI and the relation GUI.

Standard Interface Views The definition tab of the interface editor GUI provides three standard ways to organize the contents of an interface: build view, interface causality view, and system causality view. Additionally, there is a model view which can be used to directly expose selected objects from the model in a non-editable fashion. The button to the right of the text field displaying the current view is used to select between the different views. **ToDo:** add new screen shot showing the button to select view.

Interface Build View Build View is a standard interface view that may be used to add parameters to the interface using the edit interface menu bar (the interface causality view may also be used for this purpose). When the interface is deployed, parameters in the build view will be available to remote users for subscription. Within the build view, context may be added to organize interface parameters. At present it is not possible to add relations to an interface. **ToDo:** add screen shot

Interface Causality View The Interface Causality View is a standard interface view that includes a number of filters to organize parameters according to the overall causal structure of the interface: inputs filter, outputs filter, and indeterminate filter. The contents of this view are generated automatically according to the causality of the model underlying model. However, unlike the model causality view, one can add interface parameters to this view using the edit interface menu bar. Thus, one may directly declare an unmapped interface parameter as an output by placing it in the output filter. However, once the interface parameter is mapped to a parameter in the model the model's causality will define whether it is an input (independent model variable) or an output (model intermediate or result variable). **ToDo:** add Screen shot

Interface Model View The model view is a special view that is used to makes parts of the underlying model visible in the interface. For example, one might want to expose relationships in the model so users of the interface can better understand the algorithms used by the model. The model view can only reflect objects within the interface's model. Thus it is only possible to add and map model objects, with the exception of context. Context can be added directly to structure the view's contents. Objects in the model view are view-only, meaning that subscribers to the interface can see the objects at runtime but not change their values, and they cannot use the objects in build mode. **ToDo:** Add screen shot

Interface System Causality View The system causality view contains four filters: independents; intermediates; results; and indeterminates. The view presents interface parameters from an overall or global model causality viewpoint, unlike the interface causality view, which limits its scope to the internal causality of the model wrapped by the interface. This view is useful for seeing what input interface parameters or locally indeterminate interface parameters are actually dependent parameters from a global system model viewpoint, which accounts for how other remote models may have been connected to the model through the interface. **ToDo:** add screenshot

Interface Editor Menu Bar The Interface editor menu bar is available whenever an interface editor GUI is in focus and the interface definition tab is selected. This provides access to edit interface, add, and tool functions. In the figure below interfaces are being added to a Dome Model. The interface editor menus are appended beside the Dome model menu. **ToDo:** add screen shot of Edit interface

Interface Editor Add Menu The interface editor Add menu is available when the definition tab of an interface GUI is selected. When in build view, parameters may be added and context can be used to organize the view. In interface causality view only parameters may be added. The system causality view and model view do not support the addition of interface objects that are not mapped to model objects. **ToDo:** Screen shot of the interface Editor add menu

Edit Interface Menu The Edit Interface menu appears only when the definition tab of an interface is selected. Edit options are dependent upon the standard interface view selected. All edit functions will not always be enabled in the menu. The menu options apply only to DOME objects. Text field edit commands are available in a pop up menu.

TODO: Tables

Interface Editor Tools Menu The interface Tool Menu is available whenever an interface editor GUI is in focus. The Tool Menu provides access to the Mapping Tool and the Interfaces Tool, which will open in separate windows. **ToDo:** add screen shot of Tool model menu.

Adding Interface Context Context can be added to an interface in for the purpose of structuring the build view of interface parameters or the model view. Context are not mapped to objects in the model that underlies the interface. To add a context to the interface build view...

- Select the definition panel of the interface editor GUI.
- While in build view or model view, select add->context from the add menu.

ToDo: add screen shot

Adding Custom GUI Documentation will be available soon.

Adding Model View Elements The Interface model view is provided so that a ‘hole’ can be made through an interface, allowing users to observe selected objects in the model when it is deployed. To add model objects to the interface model view...

- Copy the desired model objects into the clipboard using the model’s edit menu.
- Select the definition tab of the interface editor GUI.
- Make sure that either the interface model view is selected.
- Select Add and Map->last selection or Add and Map->from clipboard... from the edit interface menu.

The model view may also be structured by adding context. If you want to place model view object in a context, the context must be added before the model objects are added and mapped. **ToDo:** add screenshot of the model view and add and map.

Importing Parameter Documentation Documentation will be available soon. Link back to the documentation editor as well.

Mapping to Model Parameters Interface parameters must be mapped to parameters in the underlying model if changes in interface parameters values are to affect the model. Parameters may be added to an interface and mapped to the model in a single operation. However, if interface parameters are added before corresponding parameters are defined in the model, mappings may be added as follows.

- Copy the desired model parameter into the clipboard using the model's edit menu.
- Select the definition tab of the interface editor GUI.
- Make sure that either the build view or interface causality view is selected.
- Select the interface parameter that is to be mapped to the copied model parameter.
- Select Map->last selection or Map->from clipboard... from the edit interface menu.

Additionally, one may map or edit interface parameter mappings using the mapping tool. TODO: add screen shot of the map option.

Integration Projects This documentation is not available.

Playspaces This documentation is not available.

Setting up Tools This documentation will be available soon

Documentation Editor All DOME objects that support documentation will have a documentation tab in their GUI. The documentation tab contains the documentation editor, as shown below (within a DOME model GUI). The URL field at the bottom of the editor may be used to point to an external documentation page on a web server. The open button can be used to test the URL. Documentation may also be entered directly into the comments field. The build documentation menu allows for simple formatting and editing of comments.

Documentation Menu The format documentation menu, as shown below, is available when an object's documentation editor is visible. Formatting capabilities provided under the Format documentation menu are:

- Standard java fonts.
- Font styles: plain, bold, italic, and underline.
- Custom font sizes.
- Custom font colors.
- Paragraph justification

Cut, copy and paste functions are available by first selecting text and then clicking the right mouse (<ctrl> click on a Mac) . TODO: add screen shot of documentation context menu

Checkout Checkout functionality is available in the build menu. Checkout allows you to retrieve models, playspaces, projects, and analysis tools from a server so that they may be edited and redeployed as an updated version of the original files. This allows upgrades to be made in a way that run mode users are seamlessly transferred to the most current versions. When you checkout any dome file, the file is downloaded to your computer for editing. All secondary files (such as custom interface GUIs or third party model files) associated with the dome file will also be downloaded. Additionally, a folder will be created in the dome file directory called domeVC. If you delete the domeVC folder you will not be able to redeploy the revised model as an update to previously deployed files. The domeVC folder contains version control information. This information is used to determine which files may be updated on redeploy, and is also used to check if the file has been modified by someone else while you had it checked out. At present, the version

control mechanism will only inform you of a potential conflict. Difference highlighting and merging mechanisms are planned for later generations of DOME3.

General Build GUI Behavior There are number of general behaviors associated with build mode GUIs. This section discusses list view or tree view editing behavior, the yellow background stale color, text field edit commands, and tree view insertion.

List Tree Views Many build mode GUIs use list views (list view includes tree or tree table views). The list view has been carefully designed for convenient editing. The table below summarizes list behaviors associated with different user inputs.

TODO: Tables

Stale Color Build mode text fields have a stale background color (yellow) to indicate when changes have not been committed. When a change is not committed, editing inputs only reside in the GUI window. However, the corresponding data in the object has not been altered. When changes are committed the information displayed in the GUI will be consistent with the object data and the background color of editing fields is white. Mechanisms to commit changes or cancel changes in editing fields are consistent with the behavior of list or tree views. An example of the stale background color (using the real number parameter GUI) is below.

Text Field Editing Edit commands are such as cut, copy and paste are available in all GUI text field elements. In order to access the pop up edit menu, place the cursor in the text field, select text as desired, and then click the right mouse (<ctrl> click on a Mac) . ToDo: Screen shot of edit pop up menu.

Tree View Insertion Tree and Tree-Table Views are hierachically organized views that adhere to the behaviors described in the list (tree) view section. When object references are inserted into these views (using model add menus), they will be located in the view according to the rules below.

TODO: Tables

Examples: Using Build Mode This documentation will be available soon. It will be a directory of the next level down examples within the build mode sections.

Using Deploy Mode

This documentation will be coming soon.

Using Run Mode

This documentation will be coming soon.

Using Server Mode

This documentation will be coming soon.

Tutorial Examples

Attention: Tutorials will be moved to the Project Wiki and can be accessed at (<https://digitalmfgcommons.atlassian.net/wiki/display/DMDIIDMC/Tutorials>)

The tutorial examples in this section are intended to step you through the process of using DOME as quickly as possible. Where appropriate, the tutorials also provide links to the detailed reference documentation. This page is the master directory for tutorial examples. The general (non plugin model specific) tutorials are structured so that a you can follow the entire tutorial in sequence, but the different tutorials can also be completed in any order following the links below or using the table of contents of the left.

Running Published Examples

This tutorial is intended to familiarize you with how to run models and projects. After completing the tutorial you should be able navigate between DOME servers and execute simulations.

Tutorials Available:

- Running models as a guest. This tutorial covers using the DOME client to log into a server as a guest and run models and projects as an individual user.
- Running models as a user. This tutorial covers using the DOME client to log into a server as a user and run models and projects, both individually and collaboratively. The tutorial is also used to illustrate different use privileges in comparison to the running models as guest tutorial.
- Bookmarks. This tutorial covers using and adding bookmarks to interfaces, projects, and playspaces.

Running Models as a Guest In this tutorial you will log in to a DOME server as a guest, navigate the server file space, and run models and projects. The tutorial is written assuming you have a DOME server running locally on your computer, and that you also have the DOME client application running. Step 1: Logging in

Figure 1: Putting DOME into run mode.

Make sure that your DOME client is in run mode using the mode combination box, as shown in figure 1.

Figure 2: Opening a new run browser window.

Use the Run->New Window menu open a new run browser window, as shown in figure 2.

Figure 3: A new run browser window before logging in to a server.

A run browser, as seen in figure 3, will open. Run browsers are similar in concept to a web browser. There is no content in the browser because you have not logged into a server yet.

Figure 4: Using the Login dialog to login as guest.

To login, press the login button to obtain the dialog as shown in figure 4. In this tutorial you will login as guest. Guest login does not require a password. The server field is used to specify which server you want to log into. This may be a server IP address or the server machine name. In this case you are logging into a server running on your computer, so you can use 'localhost' as the server address. You may type localhost or just leave the server field blank as shown in figure 4. By default this will log you in to the local server. When complete, hit the login button. Note: An alternative way to login as guest without using the login dialog is to type the server location in the server combination box in the top left of the run browser and then press the 'guest button'. Again, a blank server location defaults to the localhost server.

Figure 5: Run browser logged in to your local server.

When the login is successful you will see a public folder in the server's file space. When you log in to a server as a guest, you are automatically placed in the server file space dedicated to running models individually. The combination box reading 'server' indicates that you are in the server file space—a general space managed by the administrator for all users. There may also be file spaces for specific users or groups on the server. The view combination box indicates that you are in the part of the server file space that is dedicated to running models for individual use.

Step 2: Running models for individual use Running models for individual use means that a separate, dedicated instance of the model will be created for you when you start the model. This means that if other people are also working with the same model at the same time, changes they make will not affect your instance of the model and vice-versa.

Figure 6: Models and projects available to guests in the server file space for individual use.

Expand the Public and Tutorial Examples folders in the server file space (as shown in figure 6) by clicking on the triangles to the left of the folder icons. There are two models published in the Tutorial examples folder and one integration project that connects multiple models. 'Epoxy/Carbon lamina' and 'Polymer curing model' are models and 'As cured lamina properties' is a project, as indicated by their icons.

Figure 7: Interface to the Epoxy/Carbon lamina model

Next, expand the Epoxy/Carbon lamina model as shown in figure 7. A lamina is a sheet of carbon fibers woven into a fabric and then impregnated with an epoxy matrix. This material is used to make high-strength, light-weight components. When you expand the model, inside you will see a single interface called lamina properties interface. Interface icons look like a book with a key hole. All models are run through interfaces that provide controlled views of parameters within the model. DOME automatically organizes the parameters (or variables) in the interface into an input/output structure. This interface allows the matrix (epoxy) tensile modulus and Poisson ratio to be changed as inputs, along with the fiber tensile modulus, Poisson ratio, and fraction of the total lamina. Properties of the lamina—principle moduli E_1 , E_2 ; Poisson ratio ν_{12} ; shear modulus; and matrix volume fraction—are computed as outputs. This view of the model and interfaces (in figure 7) is static view provided for the convenient browsing of models on a server. The next step is to start the interface so that the model can be executed.

Figure 8: Opening the interface to start running the Epoxy/Carbon lamina model.

To open the interface and start running the model, first left click on the lamina properties interface icon. The name of the interface will become highlighted as shown in figure 8. Then go to the Browser menu and select open.

Figure 9: Interface opening and executing the model to ensure that the interface and model are consistent.

In a short period the interface will pop up, as shown in figure 9. During this time a running instance of the model is created for you on the server. The interface's input values are sent to the model and the model executes to make itself consistent with the interface. A red background color means that the parameter has yet to be recomputed for the new input values. Green means that the value has been updated but other downstream values related to the change have yet to be computed. A white background means the values are fully up-to-date. When all of the background is white the model and interface are fully consistent.

Figure 10: Changing input values.

To change the value of the matrix tensile modulus, click on its cell in the value column (the cell background will turn blue), enter 700000, and hit the enter key. The background color of the cell will change to yellow as in figure 10. This means that the value is entered in your client, but it has yet to be submitted to the model running on the server. Note that the submit button in the bottom right of the interface GUI is now enabled. Next, click on the value cell for the fiber tensile modulus and change it to $3.0E8$. Then, instead of hitting enter after making the change, click on the value cell for the fiber volume fraction. The fiber volume fraction becomes editable and the fiber tensile modulus is automatically entered in your client. Change the fiber volume fraction to 0.4 as shown in figure 10. Now, hit the submit button in the bottom right of the GUI. The values are sent to the model and it will execute based on the values that you have changed. You will see the background of outputs change from red to green to white. When all values are white the computation is finished. At this point you may want to make changes and submit several times. Note: If you start to change a value and would like to revert back to the previous value before entering it into the client, delete all characters in the value cell and hit return.

Figure 11: Changing from the Interface Causality View (input/output) to the Build View.

So far, you have been working with the Interface Causality View. This view is automatically generated by DOME. It structures the interface into inputs that can be changed by the user and outputs. It is also possible for the model builder to provide a different organization for the interface parameters. This is provided in the interface Build View (for builder's viewpoint). Figure 11 illustrates the process of using the view combination box to change the interface view. Left click on the combination box arrow on the right and select the 'Build View' option. In this case the builder organized the interface parameters along fiber, matrix, and lamina lines. You may also make and submit changes in build view. You will not be able to change the value of outputs. You may also switch back and forth between views at any time.

Figure 12: Closing the interface

Once you have finished running the model, you may close the interface by selecting the close option in the interface menu. This is shown in figure 12. This will close your GUI and also terminate your instance of the model that is executing on the server. Note: If you do not see the interface menu it is because your interface GUI is not in the foreground. Click anywhere on the interface GUI and then the interface menu will become available.

Step 3: Running an integration project for individual use Projects are a specific type of model that are used to connect any number of sub-models or projects together so that they operate as a single integrated simulation. Next, you will run a project that connects the lamina model with the polymer curing model. The polymer curing model computes the modulus for the epoxy matrix that accounts for the temperature at which it is cured. Before continuing, you may want to expand this model in your run browser to look at its interface.

Figure 13: Interface causality view of As cured lamina properties integration project.

When you expand the As cured lamina properties project, you will see the integrated model interface shown in figure 13. The integrated system allows you to input the fiber tensile modulus, the lamina curing temperature, and the nominal matrix modulus.

Figure 14: Selecting the build view of the project interface.

You can also see the build view of the project interface in the run browser before you open the project interface. First, select the project interface (integrated model interface) by clicking on the interface icon and then choose the Build option in the view menu, as shown in figure 14. The project builder has organized the parameters according to which model they are from. You may start running the project by double clicking on the project's interface, or you may single click on the project interface icon and select open from the browser menu.

Figure 15: Running interface for the integrated project.

In a short period a running version of the project will appear as shown in figure 15. On the server, dedicated instances of the project and the lamina and processing model that it uses have been started. While starting, the project will solve to ensure that all of the models and the project interface are consistent.

Figure 16: Entering changes and switching from list visualization to graph visualization of the interface.

Change the fiber tensile modulus and lamina curing temperature as shown in figure 16. These changes can now be submitted as you did when running the lamina model. However, in this case, before submitting the changes we will change from the spreadsheet-like list visualization to a graph visualization. This is done by clicking on the visualization combination box and choosing graph.

Figure 17: Project executing while in the graph visualization.

A graph visualization of the interface parameters is now shown. This view allows you to see how the different interface parameters are related to each other. You may submit your changes and watch the change propagate through the graph as the integration project executes. The image in figure 17 was captured during the execution process. You may use the visualization combination box to switch back to the list view to see parameter values at any time. When you are finished making changes and submitting values, you may close the project by clicking on the X in the interface window title bar. The location of the X may be on the left or right depending on the operating system your computer uses (It is on the right for windows users). Alternatively, you may select the close option in the interface menu. When you close the project, your running project and any other models that it uses are terminated on the server or servers.

Running Models as a User In this tutorial you will log in to a DOME server as a user, navigate the server file space, and run models and projects both individually and collaboratively. Comparisons will be made with the running models as a guest tutorial to illustrate different use permissions. The tutorial is written assuming you have already completed the running models as a guest tutorial, and have both a DOME server and client running locally on your computer.

Step 1: Logging in as a user

Figure 1: View of run browser logged into your local server as guest. If you just completed the running models as a guest tutorial, you will see the run browser as shown in figure 1. If not, log in as guest and expand items in the browser as shown in the figure.

Figure 2: Logging in as a tutorial user.

Next, press the run browser login button and the login window will appear. The top half of the login window has a combination box of your previous logins (provided the remember password box was checked when you logged in). This is convenient way to relog into a server if you are closing and opening new run browsers, or are switching between program modes. This information is destroyed when the DOME client application is quit. You will now login as a user. The user tutorialUser has login permissions on your server, so enter this in the user name field. The password is 123. Since you are again logging into the local server, you can leave the server location field blank. Hit the lower login button (highlighted with the red box) to complete the login.

Figure 3: Logged in to empty user file space.

You can tell that you are logged in to the server as tutorialUser by looking at the text right below the server location combination box in the upper left of the window of figure 3. When logged in as a user, the run browser is automatically set to show the user file space. Unless your server has been pre configured, there will be no users (including tutorialUser) with permission to have their own space on the server. In Step 2 you will navigate to the general server file space where the tutorial models are located.

Step 2: Running models for individual use In this step you will run several instances of the Epoxy/Carbon lamina model. Logged in as tutorialUser, you will have different access privileges from when you used the model logged in as guest.

Figure 4: Changing the run browser to the server file space.

Switch the browser to the server file space as shown in figure 4. Note that the view combination box indicates that you are viewing models available for individual use that is separate, dedicated instances models will be created for you each time you open a model interface.

Figure 5: Epoxy/Carbon lamina model expanded to show two interfaces.

Once you have expanded the public folder, the tutorial folder, and the Epoxy/Carbon lamina model you will see two interfaces (figure 5). Any model or project can have many interfaces that provide different views for different users. In this case there are two interfaces that tutorialUser is allowed to see. If you press the back arrow (highlighted with the red circle in figure 5) the run browser will switch back to the guest login from which you started this tutorial. If you then expand the Epoxy/Carbon lamina model you will see that only one interface (lamina properties interface) is made available for guests when the model was published using the deploy mode application. If you are now looking at the window for the guest login, return to the tutorialUser login (as pictured in figure 5) by pressing the forward arrow to the right of the highlighted back arrow. In general, you may log into many different servers, or as different users on the same server, within a single run browser. The forward/back arrows can then be used to navigate between servers. You may also use the server location combination box at the top of the window for the same purpose. Now, double click on the fiberInterface icon to open the interface. Alternatively, you may single click on the icon and then select open from the browser menu.

Figure 6: Running the fiberInterface to the Epoxy/Carbon lamina model.

The fiberInterface will open shortly and you can make and submit changes to run the model as shown in figure 6. The fiberInterface only exposes fiber related inputs and provides the properties of the lamina as outputs.

Figure 7: Documentation for the fiber volume fraction vf.

Double click on the icon for the vf parameter. The GUI for the parameter (in this case a real number) will pop up in a separate window. Other data types, such as matrices or vectors have different GUIs. If you click on the documentation tab (figure 7) you will see that simple documentation has been provided for the parameter. All model interfaces and parameters support documentation. Close the vf parameter by clicking on the X in GUI window title bar.

Figure 8: Opening the lamina properties interface. The fiberInterface is positioned underneath the run browser window.

The next step is to open a second interface to the Epoxy/Carbon lamina model. First move the fiberInterface window underneath the run browser window, as shown in figure 8. Click on the lamina properties interface and select open from the browser menu, as shown in the top of figure 8.

Figure 9: The lamina properties interface and the fiberInterface, each connected to a separate instance of the model.

After a short period the lamina properties interface will open as shown in figure 9. First, note that the fiberInterface only exposes the subset of input parameters related to the fiber properties, whereas the lamina properties interface provides both fiber and matrix input parameters. Make and submit changes in the lamina properties interface and note that this does not affect the fiberInterface. If you make change in the fiberInterface this does not affect the lamina properties interface. This is because you are running interfaces to models available for individual use (indicated by the view combination box in the run browser). Each time a model interface is opened, a separate, dedicated instance of the model is created and connected to the newly opened interface. If you like, you can open a second independent version of the lamina properties interface, or you could use the back button on the run browser to open a third instance of the lamina properties interface logged in as guest.

Figure 10: Bringing the fiberInterface into the foreground using the Windows menu.

There are now several windows open. Use the Windows menu to bring the fiberInterface into the foreground.

Figure 11: Closing the fiberInterface.

With the fiberInterface in the foreground, select close from the Interface menu to close the interface and terminate the model for this interface on the server (see figure 11).

Figure 12: Running the lamina properties interface.

You will still have a running version of the lamina properties interface. You can make and submit changes, as in figure 12, since this instance of the model was not affected by closing the fiberInterface. Close the lamina properties interface and any other interfaces to the Epoxy/Carbon lamina model that you may have opened.

Step 3: Running projects for individual use In this step you will run integration projects, again for individual use. Your use permissions as tutorialUser will be different from when you ran the As cured lamina project while logged in as guest.

Figure 13: Expand the As cured lamina properties project in your run browser.

Make sure that you are in the run browser window for tutorialUser. If you are looking at the guest window, use the forward button to navigate to the tutorialUser window. Expand the As cured lamina properties project in your run browser as shown in figure 13. In this case, the project was published using deploy mode with permissions that allows tutorialUser to see the interfaces of the contents of the project. You can see the overall project interface that you used in the running models logged in as guest tutorial in the interfaces folder. In the resources folder you can see the sub-models that are being integrated by the project. In the iModels folder you can see interfaces to the model inside of the project that joins the resource models. If you wish, use the back button to switch to the guest browser window. Expand the project and verify that only the project interface is visible. This is because guests were not given privileges to view the contents of the project when it was deployed. Be sure to switch back to the tutorialUser window before continuing.

Figure 14: Opening the As cured lamina properties project.

You could select the project interface integrated model interface and open it to run the project with the same view as in the running models as guest tutorial. As an alternative, this time you will open the project GUI so that you will be able to make changes to the project interface and also see the changes propagate through the resource model interfaces.

Click on the As cured lamina properties project icon and then select open from the browser menu as shown in figure 14.

Figure 15: Making and submitting changes through the project GUI.

When the project GUI opens, expand the folders and models/interfaces until your view is the same as in figure 15. Change the lamina curing temperature and the nominal matrix modulus as shown in the figure and use the submit button to execute the models in the project. Note that the changes propagate through the interface of the Polymer curing model. If you wish, you can also double click on the integrated model interface icon to open it in a separate window. Because the interface is being run from within the open project GUI, this interface window will be connected to the same running instance of the project. When you are finished running the project, close the project window.

Step 4: Running models or projects in a collaboration playspace Until now, all of the models and projects have been run for individual use, meaning that each open interface is connected to a different running instance of the model. In this step you will see how different clients can work together using a shared running instance of a model.

Figure 17: Switching to view collaborative use workspaces available in the server file space.

To begin, switch from the view from individual use model to collaborative use workspaces. Use the view combination box as shown in figure 17.

Figure 18: Opening the Polymer curing team playspace.

Expand the folders until your view is the same as in figure 18. In the server file space there is a collaboration playspace called Polymer curing team. Playspaces can be identified by the playspace icon to the left of the name. Click on the playspace icon to select the playspace and then choose open from the browser menu as illustrated in figure 18.

Figure 19: The open Polymer curing team playspace window.

After a short period a window will open for the polymer curing team playspace. A playspace may provide access to many models and projects. However, this playspace provides collaborative access to only one model, the Polymer curing model (see figure 19). Expand the Polymer curing model until your window matches the view in figure 19. You can now make and submit changes directly in the playspace window. Make a change to the curing temperature and press the submit button. Then, change the value back to match figure 19. Note: You can also double click on the polymer curing interface icon to open the interface in a separate window. If you choose to try this now, close the interface window before proceeding to the next step.

Figure 20: Navigating to the guest browser window and changing to the collaborative use view.

Normally different users collaborating through a playspace would be working on different computers. However, in this tutorial you will use the same computer to also enter the playspace as a guest user. Begin by using the Windows menu to bring the run browser into the foreground. Figure 20 shows the run browser in the foreground with the open playspace (which you entered as tutorialUser) in the background. Use the back arrow (highlighted with a red circle) to switch to the guest browser window as shown in figure 20. Then use the view combination box to switch to collaborative use.

Figure 21: Opening the playspace logged in as the user guest.

Expand the folders to match the view in figure 21, select the playspace, and then open it.

Figure 22: Two playspace windows. The top playspace is for tutorialUser while the bottom user is for guest.

In a short moment a second playspace window will open. This playspace is for the user guest. Expand the Polymer curing model and its interface. Move the new playspace window to the bottom of the screen and then click on the first playspace window (underneath the browser window) to bring it in to the foreground. Your screen should look like figure 22. You now have two playspace windows. The top playspace window is for tutorialUser. The bottom playspace window is for guest. This may be a bit confusing, but normally each user would be on a different computer. You may now make and submit changes in either playspace window. The two playspace windows reflect the same values since guest and tutorialUser are sharing the same running instance of the model through the Polymer curing team playspace. Note: Playspaces also have a conference feature that allows users in the playspace to chat with each

other while running models. If you wish, press the conference button in each playspace window and pass messages back and forth between guest and tutorialUser. Close the conference windows before continuing to the next step.

Figure 23: Closing the playspace window.

When you are done, close one of the playspace windows by first bringing the window into the foreground and then choosing the close option from the Playspace menu. Now, close the remaining playspace window.

Figure 24: Closing your run browser.

Finally, bring the run browser to the foreground and close it using the browser menu as shown in figure 24. Alternatively you can click on the X in the browser window title bar.

DOMe Models

This section of the tutorial is intended to familiarize you with how to build and deploy DOME-native models. These tutorials assume that you are familiar with materials covered in the running models tutorials, and that you have access to a DOME server on which you have been provided with a user account with file space (so you can upload models to the server). If you do not have such an account, you should create an account for yourself on a local server that you can run on your computer. You can do this by completing the first step of the server management as an administrator tutorial now. Tutorials Available:

- Model of rectangular solid with hole. This tutorial covers building a simple DOME model, creating interfaces for the model, deploying the model and setting use permissions, running the model, and editing updating the deployed model.
- Model with iterative loop. This tutorial covers building a DOME model with a loop, making interfaces, deploying the models and setting use permissions, and running the model.
- DOME models containing file objects: This tutorial covers building a DOME model that uses parameters containing data in external files.

Model of a Rectangular Solid with a Hole This tutorial covers several aspects of building DOME models and the iModels that are used in projects to connect other models. Please check that you have covered the prerequisites for the tutorial as outlined on the main DOME models tutorial page. You will model the volume of a rectangle with a hole through its center. Later, the model will be revised to include a cost figure.

The tutorial is divided into the following sequential steps.

Step 1. Starting a new model and building mathematical relations: You will use the build application to define relations for the rectangle geometry and volume.

Step 2. Connecting data in different relations: You will map parameters in different relations so that changes will propagate from one relation to another.

Step 3. Building interfaces for the model: You will create an interface for the model that defines a view that will be used to execute the model in run mode.

Step 4. Deploying the model: You will deploy the model into your user account and set access permissions on the deployed model.

Step 5. Running the model: You will execute the newly deployed model.

Step 6. Revising the model: You will be introduced to DOME versions control and will add a relation to compute cost to the model

Step 7. Redeploying an updated version: You will redeploy the updated model as an updated version of the previously deployed model.

Step 1: Starting Model and Building Relations In this step you will use build mode to start a new DOME model and implement two relations, one to calculate the block width and length as a function of the hole diameter, and another to calculate the volume of the solid. You will learn how to: add parameters (real numbers); set units; and both add and test relations.

Figure 1: Changing to run mode.

Start the DOME client application. You do not need a server running until later steps in the tutorial when your model is deployed. Make sure that the client is in build mode, as shown in figure 1.

Figure 2: Starting a new model.

Use the Build→New model→Dome Model menu item to open a new model (figure 2).

Figure 3: Changing the model name.

Click in the name field and change the name of the model from the default Dome Model to Rectangular solid with hole, as shown in figure 3. This is the model name that you will see in the run browser (in run mode).

Figure 4: Adding a procedural relation.

Now, add a procedural relation to the model. Use the Add→Procedural Relation menu item as shown in figure 4.

Figure 5: Adding real parameters to the relation.

Next, you will add real parameters to the relation. Select the relation by clicking on its icon, and then add a parameter using the Add-Real menu item as shown in figure 5. A relation parameter should then appear in the input filter. Note: If the parameter appears outside of the relation (not in the input filter), this means that you forgot to select the relation before adding the parameter. You can delete this parameter by first selecting it (click on its icon) and then choosing Edit→Delete.

Figure 6: Relation containing three parameters.

Add two more real parameters to the relation so that you now have three real parameters in the input filter, as shown in figure 6.

Figure 7: Changing the parameter names.

Rename each of the parameters as shown in figure 7. When you click on the parameter's name it will become editable. When you leave the cell or hit return the change will be entered. Note: Any object in a DOME model, except parameters that are inside of a relation, can have spaces in their name.

Figure 8: Adding units to the width parameter.

Next, add units to the width parameter. When you click on the value column cell for the width, a unit combination box will appear. Use the combination box to select change unit, as in figure 8. Note: When working with physical quantities is a good idea to assign units so that you can take advantage of DOME's dimensional analysis checks in relationships and automatic unit conversion. These features help prevent modeling errors.

Figure 9: Selecting length units.

The unit chooser will appear as shown in figure 9. Select the length category from the list of dimensions on the left of the chooser. Several length units will then appear in the units list on the right. Use the combination box below the units list to filter only for metric units.

Figure 10: Selecting a millimeter unit.

Select millimeter, as shown in figure 10, and then click ok to dismiss the unit chooser.

Figure 11: Units added to all parameters in the relation.

Add length units to the other two parameters in the relation, as shown in figure 11. Note that holeDiameter is in centimeters.

Figure 12: The relation editor GUI.

Open the relation's editor GUI by double clicking on the relation icon, as shown in figure 12.

Figure 13: Using the relation GUI to define the the width and length equations.

Enter the equations for the width and length in the relation body area as shown in figure 13. The body editor has a number of convenient features. In particular, you can type the first character of a parameter name and the use <ctrl><shift> to auto complete the rest of the name. Note: The body of the relation is written in Python. A complete listing of operations available for each type of parameter is in the methods and operators section of the reference documentation.

Figure 14: Defining relation causality

You also need to define the relation's causality so that DOME can correctly solve sets of many relations. Click on the relation causality tab, as highlighted in figure 14 (1). Then, click on the edit causality button (2). The edit causality window will appear. Use the check boxes to indicate that the width depends on the holeDiameter and that the length also depends on holeDiameter, as has been done in figure 14. When you are done click the ok button. Note: A more detailed explanation of how to read the causality editor is in the in the causality editor reference documentation.

Figure 15: Correctly completed causality information.

After closing the edit causality window the causality tab will appear as in figure 15.

Figure 16: Setting an initial value for the hole diameter.

Return to the view of relation parameters by clicking on the interface parameters tab highlighted in figure 16. Note that the relation parameters are now sorted into both the input and output filters. Click on the value column cell for the hole diameter and enter a value of 1.0. The value is entered when the cell background changes back to white.

Figure 17: Testing the relation.

It is always good practice to test relations as you go. Use the Edit procedural relation->Test menu item as shown in figure 17. Note: If this menu is not available, the relation GUI is not in focus. Click on the relation GUI or use the Windows menu to bring it into the foreground.

Figure 18: Successfully tested relation.

After a short moment, the output values of the relation should update as shown in figure 18. Note that DOME has handled the centimeter to millimeter conversions. If there are unit related problems with your relation, a dialog will appear and you should edit the units accordingly. If nothing seems to happen, there is probably a syntax error in the relation. Syntax errors are directed to the message log. The message log button is in the main model window. Carefully check you relation body and causality information in comparison to the figures in the tutorial.

Figure 19: Changing the relation name.

Finally, change the name of the relation to something more meaningful, as shown in figure 19. Use the X in the relation window title bar to close the window.

Figure 20: Adding a second relation.

Use the Add->Procedural Relation menu item to add a second relation to the model. Change the name of the relation to volume relation, as has been done in figure 20.

Figure 21: Adding parameters to the relation and setting their units.

With the volume relation selected as shown in figure 21, use the Add->Real menu item to add 5 real parameters to the relation. Change the names of the parameters to match the figure, and assign units to the parameters. Note that the vol parameters has been assigned the volume unit cubic millimeter.

Figure 22: Adding a real parameter using the relation context menu.

Double click on the volume relation icon to open its editor GUI, as shown in figure 22. Use the Add menu as shown in the figure to insert a sixth real parameter into the GUI.

Figure 23: Defining the constant PI.

Change the name of the new parameter to PI and set its value to 1.0, as shown in figure 23. Then use the constant category of the unit chooser to assign the parameter the unit of PI. In an equation, the value of 1.0 PI will be the correct 3.14... Note: If you cannot find PI in the constant list, this is probably because the unit filter is set to metric. Change the combination box to show all units, as is the case in figure 23.

Figure 24: Define the body of the relation and the relation causality.

Now, define the volume equation in the relation body and the causality of the relation as shown in figure 24, following the same process you used to define the width/length relation.

Figure 25: Testing the procedural relation.

Change input parameters to non-zero values. Remember that the PI parameter must remain 1.0 for the equation to be correct. Test the relation as shown in figure 25.

Figure 26: The tested relation.

If there were no errors, the value of the volume is now calculated. If there are errors, correct units or equation syntax accordingly. Check the message log using the message log button in the main model GUI if necessary. Close the volume relation GUI.

Figure 27: The two completed relations shown in the mode GUI.

The two relations should now appear in the model GUI as shown in figure 27. You have now completed step 1 of the tutorial.

Step 2: Connecting Data In this step you will connect data between the width/length relation and the volume relation created in step 1. You will learn how to: add mappings that connect the data in different parameters so that changes in the data of one object will propagate to the data of another object; interact with different model visualizations.

Figure 1: Adding a real parameter to a model.

To begin, the model should be open as it appears in figure 1. The first step will be to connect the holeDiameter in the width/length relation to the dia in the volume relation. With the current solving implementation, it is not possible to connect the input of one relation to the input of another relation directly. Instead, we will add a third parameter to the model (outside of the relations) and use this to synchronize the both variables holeDiameter and dia. As a short cut, copy the relation parameter holeDiameter. Select the holeDiameter icon by clicking on it so it is highlighted as in figure 1. Then choose the Edit->copy menu item. This action enters the selection into the clipboard.

Figure 2: Pasting a copy of the parameter into the model.

Now, make sure that neither of the two relations, or parameters within the relations (e.g. holeDiameter) are selected. If they are, you can deselect them by clicking on the white space at the bottom of the window. Nothing with the model will be highlighted, as in figure 2. Use the Edit->Paste copy->Last selection menu item (also shown in figure 2) to paste a copy parameter of the parameter into the model. Note: If you are unfamiliar with the difference between pasting a copy of a parameter vs pasting a reference to a parameter, you may want to brush up on objects vs. references to objects and the edit menu options.

Figure 3: The copied parameter added to the model.

If you have been successful you will see the new model parameter called holeDiameterCopy at the bottom of the window, as shown in figure 3. Model parameters have red icons. Note: If the parameter copy appeared inside one of the two relations, it means that the relation was selected when you pasted the copy. Use the Edit->Delete menu item to delete the this parameter, and then click on white space at the bottom of the window to make sure that neither of the relations are selected. Then repeat the Paste->Copy->Last selection again.

Figure 4: Moving and renaming the pasted model parameter.

Move the new parameter to the top of the model (figure 4). Drag and drop is not supported yet, so select the new parameter by clicking on it, and then use the up arrow on the right of the window to move the parameter. The relations

will collapse when you move the parameter, so you will need to re-expand them. Rename the model parameter as shown in figure 4.

Figure 5: Starting the connection (or mapping) process.

You will now start the process of connecting the diameter parameter to the holeDiameter and dia relation parameters using a mapping. After doing so, all three parameters (diameter, holeDiameter, and dia) will reflect the same value at run-time. Select diameter, and then copy it into the clipboard using the edit menu, as shown in figure 5.

Figure 6: Mapping to the hole Diameter.

Next, select the holeDiameter parameter and then use the Add→Map→Last selection to create a mapping between the copied parameter (diameter) and the holeDiameter).

Figure 7: Mapping between holeDiameter and diameter.

If you have been successful, the model will appear as shown in figure 7. If you accidentally make a mapping to the wrong object, you can delete the mapping using the mapping tool, which can be opened by double clicking on the parameters mapping column cell.

Figure 8: Mapping the dia parameter.

Repeat the mapping process for the dia parameter. Since diameter is still the current selection in the clipboard, you do not need to recopy it. Select the dia parameter as shown in figure 7 and then add a mapping to the last selection.

Figure 9: Successfully mapped parameters representing the hole diameter.

You will now see the mapping as shown in figure 9. diameter is mapped to both holeDiameter and dia. Note: Both holeDiameter and dia could have been mapped to diameter at the same time by selecting both holeDiameter and dia before performing the add→map action. <ctrl><shift> is used for multiple selection.

Figure 10: Repeating the mapping process for the block width.

Now repeat a similar mapping process for the block width. Since width is an output of the width/length relation and w is an input to the volume relation they can be mapping directly. Input–output mappings can be made directly, unlike input–input mappings. Output–output mappings are not allowed as this would over constrain the model. Select the width parameter and copy it into the clipboard as shown in figure 10.

Figure 11: Mapping width to the w parameter.

Select the w parameter and then use the add menu, as shown in figure 11, to make the mapping between width and w.

Figure 12: Mapped width and w parameters.

The mapping will now appear as shown in figure 12. Note: The values of the two widths may still be different in the build view, but they will be synchronized during model execution. If they are different, you can re-run the width/length relation and w will be synchronized with width. (select the width/length relation and then choose DOME Model→Test Relation).

Figure 13: The length and l parameter are now also mapped.

Finally, repeat the mapping process to connect the output length to the input l. The mappings will appear as shown in figure 13. All mappings needed for this model are now complete.

Figure 14: Accessing the mapping tool.

The mapping process that you followed in this step uses menu shortcuts for mapping. You can also perform all mappings using the mapping tool. Figure 14 shows the location of the mapping tool under the tool menu. The mapping tool can also be opening by clicking on a cell in the mapping column.

Figure 15: Model causality view

So far you have been working in the model build view. The is the view in which you can add objects to the model. However, it is not a good view to understand the causal structure of a model once you start to link together relations using mappings. In figure 15, the model has been switched to a Model Causality View. This view uses filters to

sort the objects by causality. It is useful to determine what variables are free, or independent, and what variables are intermediate or result parameters.

Figure 16: Graph visualization.

In addition to the spreadsheet-like list visualization of the model, there are other visualizations that help to elucidate the structure of a the model. In figure 16, the visualization combination box has been used to switch to a graph visualization. The right mouse can be used to collapse and expand regions of the graph, but in general this visualization is effective for fairly small models.

Figure 17: DSM visualization

Finally, in figure 17, the visualization combination box has been used to show a matrix, or design structure matrix (dsm), style presentation of dependency information. This view is helpful in larger models.

Figure 18: Return to the List visualization and Model Causality View.

As a final step, put your model visualization back to list and the Model Causality View. Step 2 is now complete.

Step 3: Building Interfaces In this step you will create an interface for the model that you completed in step 2. Interfaces are the exposed views of a model made available to run-mode users. You will learn how to: use the interface tool; create a new interface; use context to organize the interface; add parameters to the interface; and save models.

Figure 1: Opening the Interfaces tool.

If you just completed step 2, you will have the model open and showing the Model Causality View for the list visualization, as in figure 1. Now, open the Interfaces tool using the Tools->Interfaces menu item.

Figure 2: Creating a new interface.

When the Interface tool window opens, as in figure 2, you will see an interface called Default interface. This is an auto-generated interface that contains model parameters that are in the Independent or Results filter of the Model Causality View. In this tutorial we will build our own interface from scratch rather than using the default interface. All models can have multiple interfaces. With the interface tool in the foreground, use the Interfaces->New menu item to create a new interface.

Figure 3: The newly created interface.

The newly created interface will appear in the interface tool window, as shown in figure 3. Double click on the interface Icon to open an editing window for the new interface.

Figure 4: The interface editing window.

The interface editor window will appear in its build view, as shown in figure 4. Start by giving the interface a new name, such as volume interface.

Figure 5: Adding a context to the interface.

With the interface window in the foreground, select the Add->Context menu item. You can think of context as file folders—we will use them to organize the parameters that we put into the interface.

Figure 6: Two context have been added to the interface.

Add a second context to the interface. Name them as shown in figure 6. When you are done, click on the free dimensions context to select it.

Figure 7: Selecting independent model parameters that can be user inputs.

Next, bring the model window into the foreground by either clicking on it or using the Windows menu. Use <ctrl><click> to multiple select both the diameter and the block height h. Do not select PI, as this is something that you do not want users to be able to change. Then, as shown in figure 7, select the Edit->copy menu item to place the two parameters into the clipboard.

Figure 8: Adding the paramameters into the interface.

Now, you will add copies of the chosen model parameters to the interface, and then map them to the corresponding data in the model. Bring the interface window back into the foreground, as shown in figure 8. Select the free dimensions content, and then choose Add→Add and Map→Last selection from the menu.

Figure 9: The new interface parameters mapped to the corresponding parameters in the model.

Figure 9 shows the new interface parameters that were added. Interface parameters are green. The interface parameters are separate copies of the parameters in the model. However, they have mappings to the original parameters in the model (note the mapping column in figure 9). Thus, the interface and model parameter will mirror the same values in when the model is running. If desired, you can change the names of the interface parameters, or even their units, so long as the new units are dimensionally consistent with the model parameter to which they are mapped (e.g., changing centimeters to inches). Note: If the parameters did not appear inside the context, it means that the context was not selected. You can delete the interface parameters and repeat the Add and Map with the context selected.

Figure 10: Selecting intermediate variables for the interface.

Bring the model window into the foreground once again. Select the intermediate variables length and width as shown in figure 10. Use the Edit→copy menu item to place this selection into the clipboard.

Figure 11: Adding the intermediate parameters to the interface.

Now, bring the interface window into the foreground once again. Select the driven dimensions context, and then choose the Add→Add and Map→Last selection menu item. The new length and width interface parameters will now appear in the interface.

Figure 12: Adding vol to the interface.

Finally, return to the model and copy the vol parameter into the clipboard. Return to the interface window and add and map the interface parameter. The result is shown in figure 12. Optional step: Interfaces are designed to hide the model implementation from users at run-time. However, there are times that you might want to see the insides of an executing model, especially while debugging. There is a read-only model view intended for this purpose. If you would like to expose the relations used in this model, first return to the model window and switch to the Object Type view. Select the two relations and then copy them into the clipboard. Bring the interface window back into the foreground. Click on the model view button (the button with 3 cubes to the right of the interface name) and the model view window will open. With the model view in the foreground, select the Add→Add and Map→Last selection menu item. Mapped copies of the relation will appear in the model view.

Figure 13: Saving the model and interfaces.

The final step is to save your model on your local file space. The writes the model and interfaces out into an XML file. Use the DOME Model→Save menu item shown in figure 13.

Figure 14: Giving the model file a name.

You may want to make a directory for your model since a number of files will be saved to disk. You will be asked to give a name for the model file. The file name is just used on your local file system. The model name defined in the model window (Rectangular solid with hole) is the name that will be seen when the model is available in run mode.

Step 4: Deploying and Setting Permissions In this step you will move the model and interface files that you saved in step 3 to a server where they can be executed by remote users. You will learn how to: check model files into a server using a deploy mode wizard; select which interfaces will be available to users; and assign use permissions.

For this step you will need to have a server running on which you have a user account providing you with file space. This was addressed in the prerequisites for DOME modeling tutorials. The tutorial is written assuming that you will be logging into a server running locally on the same machine as your client application. Recall that the account for joe smith is used as a tutorial placeholder for your account.

Figure 1: Switching the client application to deploy mode.

To begin, switch the client application to Deploy, as shown in figure 1.

Figure 2: Starting the Model deploy wizard.

Select the Deploy->Model... menu item to start the model deployment wizard, as shown in figure 2.

Figure 3: Logging in to a server.

The deploy wizard will step you through a series of choices that need to be completed during the deployment process. The first step, shown in figure 1, is to log into a server on which you have an account with file space for saving models. Click on the highlighted login button in the main panel of the wizard to open the login window as shown in figure 3. Log in to your account on the local machine (remember, joe smith is a placeholder for your user name, as was defined in the server management as an administrator tutorial. The tutorial user joe smith has a password 123). Unlike the login window in the figure 3, you may also have cached logins available in the login window.

Figure 4: Successfully completed login.

After you have logged into the server, the wizard will display a summary of your login information as shown in figure 4. With the login complete, the next button is enabled so that you can proceed with the deployment process.

Figure 5: Choose the model file saved on your computer.

Use the browse button in the select model panel to open a file chooser. Navigate to the model file you saved in step 3 and select it, as shown in figure 5.

Figure 6: Adding a model description.

You can also add a model description as shown in figure 6. This description will appear in the run browser beside the model name. Press the next button to continue.

Figure 7: Selecting the location in your file space on the server.

Select the folder in which you want the model to be located on the server. Select the folder tutorial models as shown in figure 7. If you did not complete the file space management portion of the server management as an administrator tutorial, you will probably not have a folder called tutorial models. If this is the case, select your Public folder and then click the add folder button (highlighted in figure 7). Enter the folder name tutorial models. With the destination folder selected, you may now proceed to the next step.

Figure 8: Setting permissions for who can edit the model.

Now, you will be asked to define editing permissions for the model, as shown in figure 8. As owner of the model, you are automatically given all editing permissions. Editing permissions pertain to being able to check the model out from the server, edit the model's content and then redeploy it back to the server or other servers. Set model or iProject edit permissions: provides permission to decide who can set edit permissions. If you do not have this permission you will not be able to set any of the permissions in this list. Delete model or iProject: provides permission to delete the deployed model from the server during server file space management. Modify model or iProject: provides permission to check out the model from the server for editing, and to redeploy the edited model onto the server as a version update of the original model. Copy model or iProject: allows the user to check out the model, but they cannot redeploy it as a version update to the original model. They can only deploy it as a different model. Since copy is a subset of modify, this is always available when modify permissions are available. Set interface use privileges: This is available only if you have modify or copy permissions. This gives you permission to decide which users can use what model interfaces in run mode. Leave yourself as the only user with full editing permissions and proceed to the next step.

Figure 9: Choosing which interfaces will be available.

The next step, shown in figure 9, is to determine which interfaces will be made available to users. Select the volume interface for your model and proceed to the next step.

Figure 10: Assigning which users may use the interface.

Now, as shown in figure 10, you can define which users will be allowed to see and use available model interfaces. The combination box allows you to select which of the available interfaces you are setting use permissions for. In this case there is only one interface available (volume interface). As model owner you are automatically given full use permissions. Subscribe to interface: allows the interface to be used as a sub-model when you are building an

integrated project. View and run in a new playspace: allows use of the interface for individual and collaborative use, as illustrated in the running models tutorials. Save in a new playspace: the current implementation does not yet have the functionality related to this permission.

Figure 11: Provide guest with use permissions.

Leave yourself with full use permissions, and add guest as a user for the interface as shown in figure 11. Leave guest with full use permissions and proceed to the next step.

Figure 12: Summary of deployment choices.

A summary of your deployment choices will be displayed. If you change your mind, you can use the back button to go back through the wizard and make modifications. Press the deploy button to initiate the transfer of the model and interfaces to the server.

Figure 13: Completion of the deployment process.

When the file has been transferred to the server you a status dialog will appear as shown in figure 13. Dismiss the dialog. The model is now available for use in run mode.

Step 5: Running the Model In this step you will run the new model that you deployed in step 4. This material has been covered in the running published models tutorials. If you chose to add the optional model view to the interface (in step 3) you will be able to try this view for the first time.

Figure 1: Log in to the server.

Begin by switching DOME to run mode and log into the server on which you have an account, as shown in figure 1. You will have your login cached from deploying the model in step 4.

Figure 2: Open the volume interface.

Navigate to your model as shown in figure 2. Select the volume interface and open it.

Figure 3: The running interface in Interface Causality View.

The model will open in Interface Causality View (figure 3). This view organizes the parameters into inputs that the user can change and outputs. You may make changes and submit them to the model.

Figure 4: The interface build view

Switch the interface to Build View, as shown in figure 4. This is the view that you made when you created the interface (step 3). Run the model from this view. If you wish, you may also explore the different visualization options using the visualization combination box. Option: If you added the model view in step 3, you can open the model view (using the model view button to the left of the message log). This view is read-only, but you can make changes in the main interface window and see data in the relations change as the model executes.

Step 6: Revising the Model In this step you will edit the model that you deployed in step 4 by adding a cost calculation. This will be done in a way that lets you redeploy the changes as a version update in step 7. You will learn how to: understand the DOME version control system; open and edit an existing model; and save the model so that it can be redeployed. The step assumes that you are comfortable will defining relations in DOME models (step 1), making mappings (step 2), and adding parameters to interfaces (step 3).

Figure 1: The folder containing the model files on your computer file space.

We will begin with an explanation of DOME version control. Navigate to the folder where you saved your DOME model in step 3. You will see something similar to figure 1 (In our case the model was saved in a folder called DOME models). When you saved the model in step 3, the model file rectangularSolid-DOME.dml was created. The directory containing the model interface files was also created. You will also note that there is a folder called domeVC (VC for version control). This folder was created when you deployed the model to the server in step 5 . In this folder there is information that will allow DOME to determine when you can redeploy/update models already published on a server,

and whether the version on the server has changed since you last deployed or checked out the model. Without the domeVC information it is not possible to redeploy version updates.

Figure 2: Checkout model option (not required in this tutorial)

Since you built and deployed the model on the same computer that you are using now, you are ready to begin editing the model. However, if you were now on a different computer, you would need to checkout the model from the server. Checkout is available in build mode, as highlighted in figure 2. The checkout process will download copies of the model files from the server to your computer and also generate the necessary DOME version control information.

Figure 3: Opening the model.

Open your model by choosing the Build→Open model...→Dome Model menu item.

Figure 4: Select the model file.

Navigate to the folder containing your model and select it. Note that the file chooser filters out the interface folders and the domeVC folder so that it is easy to locate the model file.

Figure 5: Adding a new relation to the open model.

The model will open as shown in figure 5. You will be adding a cost calculation to the model. Add a procedural relation to the model using the Add menu.

Figure 6: Define the cost relation.

Rename the relation and add three real parameters to the cost relation. Name the parameters as shown in figure 6. Don't forget to assign units to the parameters and a value to the costPerVolume. Complete the equation shown in the relation body.

Figure 7: Define the causality for the relation.

Define the relation causality by selecting the causality tab and opening the edit causality information dialog as shown in figure 7. Test the relation (in the Procedural relation edit menu) and then close the relation GUI.

Figure 8: Copying the vol parameter into the clipboard.

Expand the volume relation in the model window as shown in figure 8. Select the vol parameter and copy it into the clipboard as shown.

Figure 9: Mapping vol to volume.

Select the volume parameter, and then Add→Add and Map→Last selection to make the mapping (figure 9).

Figure 10: Completed mapping between the volume and vol parameters.

Your model should now appear as shown in figure 10. You can double click on a cell in the mapping column if you need to use the mapping tool to edit the mapping. Changes to the model are now complete.

Figure 11: Copying the cost parameter into the clipboard.

The final step will be to add the cost variable to an interface. First, select the cost parameter and copy it into the clipboard as shown in figure 11.

Figure 12: Adding cost to the interface.

Use the Tool menu to open the interface tool. In this case you will add the cost parameter to the existing volume interface. One could also create a duplicate interface and leave the original volume interface unchanged. Double click on the volume interface to open it in a separate window. Then choose Add→Add and Map→Last selection to add a copy of the cost parameter to the interface and map it to the cost parameter in the model. This is shown in figure 12.

Figure 13: Rename the interface.

The interface will now appear as shown in figure 13. Rename the interface to volume and cost interface.

Figure 14: Save your changes.

Use the Save menu option to save your changes. Note: If you chose Save as..., you will not be able to redeploy the modified model as a version update. This is because all DOME model objects must have unique identities. Thus, during Save as..., the model, interface, and interface objects are all given new identities. If you want to keep a backup of the original model that can be redeployed, you should make a copy of the folder containing the model file, interfaces, and domeVC folder.

Step 7: Redeploying Updated Version In this step you will redeploy the model edited in step 6. This will replace the model available on the server with the newer version. This step is very similar to step 4 (deploying the model).

Figure 1: Opening the deploy model wizard.

Switch to build mode and open the deploy model wizard as shown in figure 1.

Figure 2: Log in to the server where the model was originally deployed.

Log into the model where the model was originally deployed. You will need to log in as the same user as when you deployed the model (figure 2). Press the next button when you are logged in.

Figure 3: Selecting the model file.

Use the file browser to select the model that you saved on your computer, as shown in figure 3.

Figure 4: Selecting the redeploy option.

Select the redeploy option in the Select model panel (figure 4). The deploy wizard will then contact the server and use your domeVC information to see where the model can be redeployed. If there is more than one possible location you will be asked to pick from the list of options. (A model can be deployed in several different locations if desired) User and permission information on the server will be downloaded into the wizard so that you do not need to redefine all of this information, although you may edit the information. In figure 4, you can see that the model description has been retrieved from the server. Proceed with the next button. Note: If you select redeploy and DOME determines that you cannot redeploy the model onto the chosen server, the redeploy model radio button will automatically switch to deploy as new model. You will also receive a warning if there is a newer version deployed on the server.

Figure 5: Model location.

When the Select location panel appears, the model location will already be selected to where the model was previously located. Since you are redeploying, you cannot change this location.

Figure 6: Summary of deployment options.

Press next through the permissions steps until you reach the final summary panel. The permissions will be retained as defined in the original deployment. Finally, use the redeploy button to update the model on the server.

Figure 7: The updated model in run mode.

Switch to run mode, login if needed, and navigate to the model as shown in figure 7. Note that, since both the model and the interface were changed, their version numbers have been increased. You may run the model as desired. This completes the final step of the Rectangular Solid with Hole tutorial.

Model with Iterative Loop This tutorial will be available soon.

Integration Projects

This section of the tutorial is intended to familiarize you with how to build and deploy integration projects. These tutorials assume that you have completed the running models tutorials and, at least, the rectangular solid with hole DOME model tutorial. You will need access to a DOME server on which you have been provided with a user account with file space (so you can upload models to the server).

Tutorials Available:

1: Block and pin assembly: This tutorial covers the basics of building and deploying an integration project—creating a new project, subscribing to resource models, creating an integration model, defining project interfaces, deploying the project, and running the project.

2: Integrating two models to form an iterative loop: This tutorial covers building a DOME model with a loop, making interfaces, deploying the models and setting use permissions, and running the model.

Block and Pin Assembly This tutorial covers several aspects of building integration projects. You will make a project that keeps the block and pin consistent as if they were in an assembly, and you will also compute the assembly's cost. You will subscribe to the pin and block models as shown on the right. The block is the same as was modeled in the block with hole tutorial. Please check that you have covered the prerequisites for the tutorial as outlined on the main projects tutorial page.

The tutorial is divided into the following sequential steps.

Step 1. Subscribing to models. You will use the build application to subscribe to the block and pin models.

Step 2. Integration: You will complete an iModel that contains relationships needed to join the two resource models.

Step 3. Adding project interfaces: You will create an interface for the model that defines a view that users will use to execute the model in run mode.

Step 4. Deploying the project: You will deploy the project into your user account and set access permissions.

Step 5. Running the project: You will execute the newly deployed model.

Step 1: Making Subscriptions In this step you will use build mode to start a new integration project. You will learn how to: navigate to locate published resource models (rectangular solid with hole and pin models) and add them to your project; add an integration model to your project (iModels are similar to DOME-models); and finally, how to add subscriptions to resource models to the iModel.

Figure 1: Opening a new project.

Working in build mode, select the Build→New integration project menu item to open a new project, as shown in figure 1.

Figure 2: Changing the project name.

Change the project name to 'Block and pin assembly'. The name field is in the process of being edited in figure 2. The next step will be to add resource models to the integration project. Resource models are published (deployed) sub-models that you are interested in working with—in this case the block (rectangular solid with hole) and pin models. Adding resource models to a project simply makes the models accessible for subscription/use within the your project. If you are familiar with IDEs (integrated development environments), adding resources to a project is similar to making files known to a software project. The files can be part of your software project but you may not be using them in your code actively .

Figure 3: Initiating the process of adding resources to the project.

Start the process of adding the resource models by selecting the Add→Resource menu item, as shown in figure 3.

Figure 4: Logging into the server where the resource models are located.

A resource browser will open, as shown in figure 4. You will use the resource browser like a run browser to locate the models or projects that you want to work with within your new project. Log into your local server using your username. Remember that joe smith is the tutorial placeholder for your own user name. If you have just completed the DOME model tutorials, you will probably have a cached login as shown in the figure.

Figure 5: Navigating to the Pin model and adding it to the project.

Once you have logged in, you will need to change from your use file space to the server file space, as highlighted in figure 5. Provided that your server installation was installed with the tutorial models, you will be able to navigate to

the Tutorial resources folder shown in the figure. Click on the model icon (not the interface) to select the model and then press the add resource to project button.

Figure 6: Adding the resources model.

Move the browser window down slightly, as in figure 6, so that you can see the resource list in the project GUI. You will see the Pin model is located in the list. Select the Rectangular solid with hole model and once again press the add resource to project button. You should see the two resources in the project GUI, as highlighted in figure 6. Press the done button to close the resource browser.

Figure 7: Renaming the resources.

Rename the Pin model with a new name more suited to this integration application, as is done in figure 7. This name is just a local name for convenience within the project. Note that the resource name is not lost and, of course, this has no effect on the actual deployed model. The green icon is the resource icon, and its name field is not editable—it always provides the actual name of the deployed resource model. Rename the ‘rectangular solid with hole’ model to ‘Block for my assembly’. Renaming allows you to provide meaningful model names for the context of your integration. For example, if you were modeling a car suspension, you might add a ‘wheel’ model as a resource to your project 4 times, and then give the 4 wheel models names left-front, left-rear, right-front, right-rear.

Figure 8: Adding an integration model to the project.

The renamed models can be seen in figure 8. The next step is to add an integration model to the project, in which you will define relationships to connect the block and pin models. Select the Add→Integration model menu item as shown in figure 8.

Figure 9: Copying desired resource model interfaces into the clipboard.

The iModel is visible in figure 9. You are now ready to select the interfaces to which you wish to subscribe so they may be used in the iModel. In this case each resource model has only one interface, so select both models interfaces holding <ctrl><shift> for multiple selection. Copy the selected interfaces into the clipboard using the Edit→copy menu item as shown in figure 9.

Figure 10: Adding subscriptions to the iModel.

Next double click on the iModel icon to open the iModel GUI. The iModel GUI is in the foreground in figure 10. Use the Add→Add and subscribe menu item to add interface subscriptions to the iModel.

Figure 11: Subscribed interfaces in the iModel.

In figure 11 you can see the subscribed interfaces in the iModel. The model name associated with each subscribed interface is in the value column. If you expand the pin interface, you can see its input/output parameters. Subscription parameters have yellow icons. You can also view the interface in its build view by selecting the interface and then choosing build from the View menu. However, for the next steps the interface causality view shown in the figure is convenient to work with. You might note that the subscription interface looks very much like a procedural relation in a DOME model. In fact, you can think of the the subscribed interface simply as a special type of relation, and iModels are just a special type of DOME model that support the addition of subscribed interfaces. Therefore, the remainder of the modeling to integrate the pin and block models will be just like building a DOME model, as you did in the Rectangular solid with hole tutorial.

Figure 12: Changing the iModel name.

Change the iModel name to assembly integration model, as in figure 12. Also, expand both the pin interface and the volume and cost interface and take a moment to familiarize yourself with their contents.

Figure 13: Saving the project.

Save your work by choosing the Integration Project→Save menu item. Give the project files the name blockAnd-PinAssembly, as in figure 13. A number of XML files will be saved to disk, so you may want to create a new folder for the project before saving. You have now completed the process of adding resource models to the project and then adding subscriptions to a project iModel.

Step 2: Defining Integration Relationships In this step you will complete the project iModel started in step 1. You will make the mappings needed to keep the block and the pin in parametrically consistent states—as if they are part of an assembly. Additionally, you will define a relationship to calculate the cost of the assembly based upon the cost of the sub-components. The process will be the same as for building DOME models.

Figure 1: Adding a real parameter to the iModel.

Your view of the project's integration model should be the same as in figure 1. The first thing you will do is connect the diameter used by the pin and the hole diameter used by the block so that they will be the same. Since no intermediate calculations are needed between the two parameters, you will be able to use mappings. However, since input-input mappings are not supported by the DOME solver currently, you will need to create a model parameter to drive them, just as you did in the Rectangular model with hole tutorial (figures 1-9). Choose Add->Real to put a new parameter into the model as shown in figure 1. Note: When you add the parameter, elements of neither interface should be selected or you will get a 'cannot paste' dialog. Clicking on the white space at the bottom of the window will deselect the interfaces.

Figure 2: Name the new parameter.

Rename the model parameter (at the bottom of figure 2) to 'hole diameter'. Give the parameter the unit centimeter and a default value of 4.

Figure 3: Coping the hole diameter into the clipboard.

With the hole diameter parameter selected, choose the Edit->copy menu item to add it to the clipboard, as shown in figure 3.

Figure 4: Hole diameter mapped to the pin interface diameter.

Select the diameter in the pin interface and then choose the Add->Map menu item. The hole diameter and pin diameter will now be mapped as in figure 4.

Figure 5: Mapping the block's hole diameter.

Now, select the diameter in the volume and cost interface of the block. Choose the Add->Map menu item and the block diameter will also be mapped to the hole diameter parameter, as highlighted in figure 5.

Figure 6: Graph visualization of the model.

The graph visualization is useful to check the structure of the model. Use the visualization combination box to switch to the graph as shown in figure 6. You can see how the hole diameter drives the diameter in both of the subscribed interfaces.

Figure 7: Copying the block height.

Switch back to the model list visualization. Now, you will work on making the length of the pin match the h of the block. Add another real parameter to the model (using the Add menu) and name it block thickness, as shown in figure 7. Set its unit and default value to match the figure.

Figure 8: Coping the block thickness into the clipboard.

Select the block thickness and copy it into the clipboard as shown in figure 8.

Figure 9: Fully mapped block thickness.

Select the length parameter in the pin interface and choose Add->Map. Select the h parameter in the block volume and cost interface and choose Add->Map. You should now see mappings as highlighted in figure 9.

Figure 10: The new model graph.

Switch to the graph visualization to verify the model structure, as shown in figure 10.

Figure 11: Adding an assembly cost relation to the model.

Switch back to the model list visualization. Choose Add->Procedural relation to add a relation to the model. Rename it to match figure 11.

Figure 12: Copying the pin cost into the relation.

Select the pin cost and copy it into the clipboard as shown in figure 12.

Figure 13: Adding and mapping pin cost into the relation.

Now, select the assembly cost relation and choose Add->Add and Map to add a relation parameter and map it to the pin cost in one step. You will see the mapping information appear in the mapping column.

Figure 14: Rename the relation parameter.

Rename the relation parameter to pin cost as shown in figure 14. Note that after entering the change, the mapping information for the parameter cost in the pin interface will update to the new relation parameter name.

Figure 15: Adding the block cost to the relation.

Now, select the cost parameter in the block volume and cost interface. Use the Edit->copy menu to put the selected parameter into the clipboard. Select the assembly cost relation, and then use the Add->Add and Map menu item to add a new parameter to the relation and map it to the cost in the block's interface in a single step. Finally, as shown in figure 15, rename the new relation parameter from cost1 to block cost.

Figure 16: Graph view of the model structure.

Switch back to the graph visualization to verify the model structure. Your graph should look like figure 16.

Figure 17: Adding additional parameters to the relation.

Switch back to the list visualization, select the assembly cost relation, and then add a new real parameter to it using the Add menu, as shown in figure 17.

Figure 18: Complete set of relation parameters.

Repeat the process to add a second new parameter to the relation. Name the two new parameters as shown in figure 18. Set the default value for the assembly labour and assign dollar units to both parameters.

Figure 19: Defining the body of the relation and its causality.

Double click on the assembly relation icon to open its editor window. Complete the equation as shown in figure 19. Also, define the causality for the relation using the causality editor. Test the relation, and then close the relation window.

Figure 20: Graph of the model.

Once again, switch to the model graph visualization. The structure of your graph should match figure 20.

Figure 21: The completed integration model.

Switch the model back to the list visualization and collapse the subscribed interfaces to match figure 21. You are now ready to add project interfaces. You may want to save the project at this point.

Step 3: Adding Interfaces In this step you will create an interface for the project that you completed in step 2. The interface will expose parameters in the project iModel for use in run mode. The actions in this step are very similar to adding interfaces to a DOME model, as in step 3 of the rectangular solid with a hole tutorial.

Figure 1: Opening the project interface tool.

To begin, use the Tool->Project Interfaces menu item to open the project interface tool, as shown in figure 1. Note: Integration models can also be provided with interfaces, but you will not be working with these in this tutorial.

Figure 2: Renaming the new project interface.

Use the Interfaces->New menu item to add a new project interface. Rename the interface to match figure 2-'assembly costing interface'. Double click on the interface icon open the interface's window.

Figure 3: Renaming a context in the interface.

With the interface window in the foreground, use the Add->context menu to add a context to the interface. Rename the context as shown in figure 3.

Figure 4: Selecting user changeable design variables.

Bring the integration model window into the foreground, and switch the view to Model Causality View as highlighted in figure 4. Select the independent parameters hole diameter and block thickness and copy them into the clipboard. This is shown in figure 4. Use <ctrl><shift> for multiple selection.

Figure 5: Adding the the independent parameters to the interface.

Bring the interface window back to the foreground. Select the design variables context, and then choose Add->Add and Map->Last selection as shown in figure 5. This will add parameters to the interface and map them back to the parameters in the model.

Figure 6: Newly added interface parameters.

Your interface should now look like figure 6.

Figure 7: Adding a context to the interface.

Use the Add menu to add another context to the interface. Name the context 'block configuration' as has been done in figure 7.

Figure 8: Copying the block parameters into the clipboard.

Bring the integration model window back into the foreground and switch back to Build View, as highlighted in figure 8. Expand the block volume and cost interface and select the parameters diameter, h, width, and length. Use the Edit menu to copy them into the clipboard.

Figure 9: Adding the block parameters to the interface.

Bring the interface window back into the foreground, select the block configuration context, and then Add and Map the block parameters into the interface, as shown in figure 9.

Figure 10: Rename the block interface parameters.

For clarity, rename the block interface parameters as shown in figure 10.

Figure 11: Adding another context to the interface.

Now, add another context to the interface. Name it 'pin configuration', as in figure 11.

Figure 12: Selecting the pin parameters.

Return to the model window and expand the pin interface subscription, as shown in figure 12. Select the length, diameter, head diameter, and head height parameters and copy them into the clipboard.

Figure 13: Add pin parameters to the interface and rename them.

Return back to the interface window and, with the pin configuration context selected, use the Add and Map command to add the selected pin parameters to the interface. Rename the parameters as shown in figure 13 for clarity.

Figure 14: Add a cost parameters context to the interface.

Add a final context to the interface and name it 'cost parameters', as shown in figure 14.

Figure 15: Selecting the cost parameters from the model.

Return back to the model window, expand the assembly cost relation and select the pin cost, block cost, and assembly cost. Then, use the Edit menu to copy the selection into the clipboard. This is shown in figure 15.

Figure 16: Cost parameters added to the interface.

Return back to the project interface window and select the cost parameters context. Use the Add and Map command to add the selected cost parameters to the interface, as has been done in figure 16. Save the project. The project is now complete and ready for deployment.

Step 4: Deploying the Project In this step you will move the project and interface files that you saved in step 3 to a server where they can be executed by remote users. You will learn how to: check project files into a server using a deploy mode wizard; and assign a variety of project-related use permissions.

Figure 1: Opening the project deploy wizard.

Switch DOME to deploy mode and select Deploy->Project, as shown in figure 1.

Figure 2: Login in to your server.

Log into to your local server using your username (joe smith is a placeholder username). This step is completed in figure 2. Press the next button to continue with the deployment process.

Figure 3: Locate the project file in your file space.

Press the browse button and locate the blockAndPinAssembly in your file space, as shown in figure 3.

Figure 4: Add description.

Add a comment to appear with the project name in run mode. 'My first project' was added in figure 4. Proceed to the next step.

Figure 5: Select deployed location.

Select the tutorial models folder in your user account. This will be the location of the deployed project on the server. Proceed to the next step.

Figure 6: Setting project editing permissions.

Set the project editing permissions as shown in figure 6. These permissions are identical to those described in the deploy step of the rectangular model with hole tutorial. Also, concepts related to checking out models/projects/playspaces for editing and redeployment were covered in step 6 of the rectangular solid with hole tutorial. Leave yourself with editing permissions and proceed to the next step.

Figure 7: Select project interfaces to be made available.

Select the assembly costing interface so that it will be available on the server (figure 7). In this case the project has only one interface, but in other cases you may have defined several different interfaces to your project. Proceed to the next step.

Figure 8: Setting use permissions for the project interface.

Add guest to the list of users who will be able to see the project interface and use it in run mode. The interface use permissions are identical to those described in the deploy step of the rectangular solid with hole tutorial. Proceed to the next step.

Figure 9: Setting permissions on who can see inside of the project.

The running models as a user tutorial illustrated what seeing inside of a project means—that the user can see what resources have been subscribed to, and observe the resource models executing from within the project run window. Leave yourself with full permissions to see inside of the project as in figure 9. The meaning of the different permissions on this set are as follows: May set iProject content visibility permissions: Gives permission to edit this permission set. May set interface use permissions for iModels: Gives permission to edit use permissions on the interfaces of iModels within the project. May see contents while subscribing: Gives the user permission to see inside the project when it is a resource in another project. This would allow the user to directly subscribe to resources models within the subscribed project (avoiding hierarchical encapsulation). May see contents and run iProject in new playspace: Allows the user to see inside of the project in run mode, and to execute the project for both individual use and collaborative use. May see contents and save iProject in new playspace: The functionality to support this permission has not been implemented yet.

Figure 10: Selecting which integration model interfaces will be available on the server.

The next step is to select which iModel interfaces to make available to project users. In this tutorial, leave the Default interface unchecked since we will not address iModel interfaces while building the project. Proceed to the next step.

Figure 11: Summary of your deployment options.

The final step summarizes the deployment options that you selected, as shown in figure 11. Press the deploy button and in a short period you will receive a confirmation dialog that the deployment was successful. Your new project is now ready to run.

Step 5: Running the Project In this step you will run the new project that you deployed in step 4. This has been covered in the running published models tutorials so it is not duplicated here.

Figure 1: The project viewed in the run browser.

Switch to the run mode application and log into your user account. When you navigate to your tutorial models folder, you should see the new project as shown in figure 1. When you run the project you may want to try the project interface in the Build View that you created when making the interface. If you start the project by opening the project window (double click on the project icon), you will be able to see changes propagating through the resource models while the project is executing. If you log in as guest and navigate to your user space, you will only be able to see the project interface. You have now completed the block and pin assembly tutorial.

Collaboration Playspaces

This section of the tutorial is intended to familiarize you with how to build and deploy playspaces. These tutorials assume that you are familiar with materials covered in the running published models tutorials, and that you have access to a DOME server on which you have been provided with a user account with file space (so you can upload playspaces to the server).

If you do not have such an account, you should create an account for yourself on a local server that you can run on your computer. You can do this by completing the first step of the server management as an administrator tutorial now.

Tutorials Available:

1: Playspace for lamina design team. This tutorial covers building and deploying a playspace that contains models and projects described in the running published models tutorial.

Playspace for Lamina Design Team This tutorial covers introductory aspects of building and deploying a playspace. Please check that you have covered the prerequisites for the tutorial as outlined on the main playspace tutorials page. You will create and run a playspace containing models and projects related to carbon/fiber composite lamina. You worked with these models in the running models as a guest and user tutorials.

There are three steps in the tutorial.

Step 1: You will open a new playspace using build mode and then select models to be accessed through it.

Step 2: You will use a deploy wizard to move the playspace into your user account and to set access permissions.

Step 3: You will run models through the newly created playspace and use a playspace chat tool to communicate with other users of the playspace.

Step 1: Build Lamina Team Playspace In this step you will use build mode to start a new playspace. You will add models and projects to the playspace. You will learn how to: open a new playspace; locate models for the playspace; and add selected content to the playspace.

Figure 1: Opening a new playspace.

Begin by using the Build->New playspace menu item to open a new playspace, as shown in figure 1.

Figure 2: The playspace build window.

The playspace window that will appear is shown in figure 2. Its functionality is very simple—it allows you to give the playspace a name and add/remove content.

Figure 3: Changing the playspace name.

Change the playspace name to ‘Lamina design team’, as shown in figure 3. This is the name that users will see while using the playspace in run mode.

Figure 4: Opening the add models window.

Now, you will add content to the playspace. Press the add button highlighted in figure 4 and the add models window will open. This window contains the familiar run browser.

Figure 5: Logging in to your account.

Press the login button in the browser and then log into the server that you have been using for the tutorials. In figure 5 there is a cached login for the tutorial ‘placeholder user’ joe smith.

Figure 6: Switch to the server file space.

You will need to change from you user account file space to the server (adminstrator group) file space, as shown in figure 6. This is where the tutorial models that will be used in the playspace are located.

Figure 7: Making models available within the playspace.

Expand the folders in the server file space so that your view matches figure 7. Select the Epoxy/Carbon lamina model and then press the put selection in playspace button. The model will appear in the playspace window as highlighted in the figure. Repeat this process to add the Polymer curing model and the As cured lamina properties project to the playspace. Note: The content of the model is not actually moved to the playspace. The playspace only contains a URL. Thus, when models are run through a playspace they are still executed on the server where they were originally deployed, and user access to interfaces is still determined by those defined by the original publishers of the model.

Figure 8: The playspace containing three models and a project.

Press the done button on the add models window. Your playspace will now look like figure 8.

Figure 9: Saving the playspace.

The final step is to save the playspace. Use the DOME Playspace→Save menu item as shown in figure 9.

Figure 10: Naming the playspace file.

Navigate to where you want to save the playspace in your computer file system. Give the playspace file the name Lamina team, as shown in figure 10.

Figure 11: The completed and saved playspace.

Figure 11 shows the completed playspace. The path and name of the playspace file is in the dark text field at the bottom of the playspace window. The playspace is ready to be deployed. Use the Dome Playspace→Close menu item to close the playspace build window bore moving on to the next steps.

Step 2: Deploy Lamina Playspace In this step you will move the playspace file that you saved in step 1 to a server where it can be accessed by remote users. You will learn how to: check playspace files into a server using a deploy mode wizard; and assign use permissions that will determine who can enter the collaborative playspace.

Figure 1: Opening the deploy playspace wizard.

Put the DOME client application in deploy mode and select the Deploy→Playspace... menu item to start the deploy wizard, as in figure 1.

Figure 2: Completed login phase.

Log in to your user account on the server that you have been using for the tutorials. This step is completed in figure 2 for the placeholder user joe smith. Press the next button to proceed with the deployment process.

Figure 3: Select the Lamina team playspace file.

Use the browse button to open the file chooser as shown in figure 3. Select the Lamina team file that you saved in step 1.

Figure 4: Adding a description.

If desired, add a description to appear with the playspace name in run mode. In figure 4 the description ‘My first collaboration work area’ has been entered. Press the next button.

Figure 5: Locating the playspace on the server.

The next step is to define where the playspace is to be located on the server. Note that, as highlighted in figure 5, playspaces are located in the collaborative use portion of your file space. Select the Public folder. Use the add folder button (also highlighted in the figure) to create a new folder. Give the folder the name ‘tutorial playspaces’.

Figure 6: Selecting the playspace destination folder.

Select the tutorial playspaces folder as the destination for your playspace as in figure 6, and then press the next button.

Figure 7: Setting Edit permissions for a playspace.

Now, you will determine who can edit the contents of playspace. Give yourself editing permissions. The playspace edit permission set shown in figure 7 is very similar to edit permissions for models and projects. The meaning of the different edit permissions are summarized below. Set playspace editing permissions: provides permission to decide who can set the edit permissions. If you do not have this permission you will not be able to set any of the permissions in this list. Delete playspace: provides permission to delete the deployed playspace from the server during server file space management. Modify playspace: provides permission to check out the playspace from the server for editing, and to redeploy the edited playspace onto the server as a version update of the original. Copy playspace: allows a user to check out the playspace, but not redeploy it as a version update to the original. The user can only deploy it as a different playspace. Since copy is a subset of modify, this is always available when modify permissions are available. Set playspace use privileges: This is available only if you have modify or copy permissions. This gives you permission to decide which users can join the playspace in run mode. Concepts related to checking out models/projects/playspaces for editing and redeployment were covered in step 6 of the rectangular solid with hole tutorial. Proceed to the next step.

Figure 8: Adding users to the playspace.

Add guest as a user to the playspace as in figure 8. This means that guest logged into the server will be able to see and join the playspace. However, what they can see of the models inside the playspace is determined by the individual model use permissions. A summary of the use permission list is below. Change values: allows the user to run models in the playspace, provided that they have use permission for the specific model. Save state versions: the current implementation does not yet have the functionality related to this permission. Open as a separate playspace: the current implementation does not yet have the functionality related to this permission.

Figure 9: Users added to the playspace.

Figure 9 shows guest and yourself added as users/members of the collaborative playspace. Proceed to the next step.

Figure 10: Summary of deployment options.

The final step provides you with a summary of your deployment options. Press the deploy button to move the playspace file to the server.

Figure 11: Confirmation of successful deployment.

When deployment is complete, a confirmation dialog will appear as shown in figure 11. Close the playspace wizard.

Step 3: Running Lamina Playspace In this step you will use the playspace deployed in step 2. This material has been covered partially in the running published models as a user tutorial. However, this step also illustrates using an interface GUI from within a playspace and the playspace chat tool (conference).

Figure 1: Logging into the server.

In run mode use a run browser window, as shown in figure 1, to log into the server with your account that you used in step 2 when deploying the playspace.

Figure 2: Switch to your collaborative use space.

When you log in, you will be placed in your individual use file space, as shown in figure 2. Use the view combination box to change to your collaborative use file space.

Figure 3: The lamina design team playspace.

When you expand your tutorial playspaces folder you will see the Lamina design team playspace, as shown in figure 3. To enter the playspace, either double click on the playspace icon or select the playspace and choose the Browser->Open menu item.

Figure 4: Running the Epoxy/Carbon lamina in the playspace GUI.

Within the open playspace window you can run models directly by expanding interfaces in tree view (figure 4).

Figure 5: Using an interface window within the playspace.

You may also open model or project interface windows from within the playspace. The lamina properties interface is shown in figure 5. Note that the value in the interface is the same as the value in the playspace window view. Both windows reflect the same running instance of the model. Further, if a model or project has multiple interfaces, you can open all interfaces and execute them on the same instance of the model. (In the individual use file space each interface would open with separate running instances of the model).

Figure 6: Conferencing with a collaborative user.

Of course, the main point of the collaborative playspace is to allow users in different places to work together on the same running instance of the models and projects within. The conference button (figure 6) has been used to open the playspace's chat tool. The tool lists all users that are members of the playspace, as well as those currently online and working in the playspace. On a remote machine the user tutorialUser has joined the playspace. Now, both you and tutorial user will be able to make changes and share the effect of each other's changes. If you have a colleague who can log into your server and navigate to your user file space, you may want to try collaborating with them now. They are able to join because you gave guest permission to use the playspace (during step 2). The chat tool can be used to communicate with your collaborators while you are working with the models. Close the playspace window. You have now completed the Lamina design team tutorial.

Interfaces

This set of tutorials focuses on different aspects of model and project interfaces. All Interface tutorials assume that you have covered the running published models tutorials. Additional prerequisites may be required in the individual tutorials.

The interface tutorials available are:

1. Using a Custom GUI for the rectangular solid model: You use a custom interface GUI in run mode.
2. Adding a Custom GUI to the rectangular solid model: You will add an existing Custom GUI to a model interface.
3. Writing a custom GUI for the rectangular solid model: You will write a custom JAVA swing GUI for a model interface.

Using Custom GUI for Rectangular Solid Model All interfaces may have multiple custom GUIs in addition to the standard DOME GUIs seen thus far in the tutorial. In this step you will use a custom interface GUI for the rectangular solid with hole model. This model is similar to the one you built in the DOME model tutorial but we will use a pre-deployed version of the model. We do assume you have completed the prerequisites for the DOME model tutorials.

Figure 1: Log into your account.

Log into your user account on the server that you have been using for the tutorials. You will be placed in the individual use portion of your file space on the server, as in figure 1. joe smith is the tutorial placeholder for your user name.

Figure 2: Opening the rectangular solid with hole model.

Switch to the server file space as shown in figure 2. Expand the folders until you can see inside of the Tutorial resources folder. Open the volume and cost interface of the Rectangular solid with hole model.

Figure 3: The Standard DOME interface GUI.

As in figure 3, the interface will open and display the standard, Dome Default GUI. Use the GUI combination box (in the upper right of the window) to select the custom interface GUI. In general, a given interface may have none-to-many custom GUIs associated with it.

Figure 4: The custom interface GUI.

The custom interface GUI is shown in figure 4. You may make changes and submit them as if working in the Dome Default GUI. In this case, the custom GUI has been written to provide a picture of the block. You may switch back and forth between default and custom GUIs at any time. This completes the using a custom GUI for the rectangular solid model tutorial.

Adding Custom GUI to Rectangular Solid Model In this tutorial you will learn how to associate a custom interface GUI with a model interface. This tutorial is only needed if you anticipate writing custom user interfaces for the models that you build. You will open an existing model, associate a pre-defined custom GUI with the interface, deploy the model, and then run the model.

The tutorial is divided into the following steps.

Step 1. Open model and associate a custom GUI: You will use the build application top open a model and then associate a custom GUI with one of its interfaces.

Step 2. Deploy the model and custom GUI: You will deploy the modified model.

Step 3. Using the custom GUI: You will use the custom GUI to execute the model in run mode.

Step 1: Associate the GUI with an Interface In this step you will use build mode to associate a custom GUI with a model interface. You will open an existing tutorial model, make a copy of it, associate a pre-defined custom GUI with one of its interfaces, and save your changes.

Figure 1: Opening an existing model. Begin by using the Build->Open model->Dome Model menu as shown in figure 1.

Figure 2: Select the rectangularSolid model.

Navigate to the dome3/tutorialModels/interface/rectSolidCustomGUI folder. You will see the rectangularSolid model as shown in figure 2.

Figure 3: The opened model.

The model should open as in figure 3. The model's name is Rectangular solid with hole and custom GUI.

Figure 4: Save a copy of the model.

You should work with a different copy of the model, elsewhere on your file space, so that the models setup for the tutorial remain in tact. Use the Dome Model→Save as... menu as in figure 4 for this purpose.

Figure 5: Naming your own copy of the model.

Once you have decided where to save the model copy, give it the name shown in figure 5.

Figure 6: Open the interface tool.

Now that you are working on a new copy of the model, open the interface tool using the Tools→Interfaces menu as in figure 6.

Figure 7: Opening the add custom GUI window.

Open the volume and cost interface. Then, use the graphical interface combination box to select the add... option as shown in figure 7.

Figure 8: Giving the custom GUI a name.

Once the Custom GUI Chooser window has opened, complete the name to appear in interface combobox field as shown in figure 8. Then press the select button to locate the pre-defined custom GUI that was made for the interface.

Figure 9: Selecting the jar file contain the custom GUI.

Navigate to the dome3/tutorialModels/interface/rectSolidCustomGUI folder and choose the file rectVolumeAndCost-GUI.jar.

Figure 10: Set the main class in the custom GUI class.

The contents of the jar file will then be listed in the select main GUI class portion of the chooser, as shown in figure 10. This is the file that the interface will use to call the constructor for the custom GUI. Select the class file as in the figure, and then press the set button.

Figure 11: Test the validity of the GUI class selection.

After you have selected the main class, press the test button to verify that the selected class is of the correct type, as shown in figure 11. Press OK to dismiss the dialog and the custom GUI chooser.

Figure 12: Switching to the custom interface.

Now that you are back to the interface GUI, switch to the custom GUI using the graphical interface combination box, as in figure 12.

Figure 13: The custom interface GUI.

The custom interface GUI should appear as in figure 13. If the custom interface GUI is blank, check the message log for errors. Use the edit... option in the graphical interface combination box to check that you did not make any mistakes.

Figure 14: Saving your changes.

Save your changes using the Dome Model→Save menu item. The custom GUI is now associated with the volume and cost interface.

Step 2: Deploy the Model In this step you will deploy the updated Rectangular solid with hole and custom GUI model to a server where it can be executed. During the deployment process the files needed for the custom interface GUI are transferred to the server, allowing remote users to use the custom interface GUI. Deploying models is covered in many different tutorials, so only key steps are covered.

Figure 1: Using the deploy wizard to log into your server.

In deploy mode use the Deploy→Model menu to open the deploy wizard. Log into your account on the server that you have been using for the tutorials, as in figure 1. joe smith is the tutorial placeholder for your user name.

Figure 2: Select the your model.

Proceed to the Select model page and then browse to select the model that you modified and saved in step 1. This is shown in figure 2.

Figure 3: Adding a description for the model.

Add the description as shown in figure 3. This will appear beside the model name in the run browser.

Figure 4: Selecting the location of the model on the server.

Proceed to the Select location on server panel. Select the tutorial models folder inside your Public directory, as has been done in figure 4.

Figure 5: Selecting interfaces to deploy.

Proceed to the Select interfaces to deploy panel. Select the volume and cost interface for deployment. If you wish, you may also make the default interface available. Complete the remaining deployment steps and deploy the model to the server.

Step 3: Running the Model In this step you will use run the model and custom interface you deployed in step 2. The process is the same as covered in the Using custom GUI for rectangular solid tutorial, with the exception that your model will be located in your own file space.

Figure 1: Opening the interface with a custom GUI.

Use a run browser to log into your account on the server. Expand your tutorial models folder, as shown in figure 1. Select the volume and cost interface and then open it.

Figure 2: Selecting the custom interface GUI.

The interface will open showing the default DOME interface GUI. Use the combination box, as in figure 2, to switch to the custom GUI. You may make and submit changes using the custom GUI or the Dome default GUI.

Writing Custom GUI for Rectangular Solid This tutorial provides a very brief overview of how to write a custom interface GUI. It is useful only if you plan on making your own custom GUIs. It is assumed that you know how to work with Swing and are comfortable with Java development. Source code is for the custom GUI used in the Using custom GUI for the rectangular solid model and the Adding a custom GUI to the rectangular solid model tutorials. Some key points and code excerpts are discussed below.

The complete source code for the custom GUI shown above is in `dome3/tutorialModels/interface/rectSolidCustomGUI/VolumeAndCostC`

Figure 1: DOME classes that need to be imported.

Figure 1 shows the DOME classes that are needed for the custom GUI. You will need to include `dome.jar` as a library. The `CustomGui` and `ModelInterfaceBase` classes are essential. The `Templates` class is not required—it is a convenience class that is used throughout DOME for creating Swing components. The `Templates` class was used in this custom GUI. JavaDoc is available for the `CustomGui` and `Templates` classes.

Figure 2: Constructors for the custom GUI.

You will need to write two constructors for the class. The top constructor in figure 2 is the one that will be used by the DOME interface. The second constructor is a standard constructor that can be used to run the GUI for standalone testing. The constructor used by the DOME interface calls the standard constructor and then the additional `setInterface` method.

Figure 3: Set interface method for the custom GUI.

The `set interface` method connects the value of the DOME interface parameters to Swing components in your custom GUI. For example, ‘diameter’ is the name a parameter in the DOME interface and `holeDiameter` is a `JTextField` in the custom GUI. There are a number of static methods in the `CustomGUI` class for connecting parameters containing various DOME data types to different Swing components.

Figure 4: Using the `Templates` class.

Figure 4 shows an example of using the DOME Templates class to create a DOME formatted label. A large number of Swing co
You are not required to use this convenience class when making your own custom GUIs.

Figure 5: main method needed to test your GUI independent of DOME.

If you want to test your custom GUI as a standalone component, you will need to define a main method similar to the one in figure 5.

Server Management

This tutorial is intended to familiarize you with how to manage your accounts, server file space, and passwords. After completing the tutorial you should be able to use the DOME server mode application as an administrator or as a user.

Tutorials Available:

- 1: Server management as an administrator. In this tutorial you will create users, passwords and add users to group. You will create groups and add users groups. Additionally, you will navigate the server file space learning how to add and delete content.
- 2: Server management as a user. In this tutorial you will change your user password. If you have been provided with file space on the server you will also learn how to add and delete content in your file space.

Server Management as an Administrator This tutorial is intended to provide an overview of server administrative capabilities available to members of the administrators group. This includes: managing users and their passwords; groups; your own password; and the server file space. If you are completing this tutorial to proceed with the DOME models tutorial, you only need to do step 1 at this time.

Step 1: Creating users, passwords and adding users to groups.

It is assumed that you have a local server running on your computer and that you are running the DOME client application.

Figure 1: Switching the DOME client to server mode.

Make sure that your DOME client is in server mode using the mode combination box as shown in figure 1.

Figure 2: Opening the login window.

Use the login option under the Server menu to bring up a login window (figure 2).

Figure 3: Entering login information.

In a default server installation, the user root is a member of the Administrators group. Enter the user name root in the user name field as shown in figure 3. The default password for the user root is cadlab. For security reasons you should change the root password (this is covered in step 3 of this tutorial). Any server can be administered from a computer running the DOME client application in server mode. In this tutorial, it is assumed you will be logging into a server running on the same computer as your client. Therefore you do not need to enter a location or machine name in the server field. If your server is on another computer you will either need to provide the machine name or IP address. Select the administrator radio button to login as an administrator. By default, 'login as' is always set to user. Once your login window is completed as shown in figure 3, press the login button.

Figure 4: Listing the users on the server.

Under the Users menu select the List Users option as shown in figure 4. This will open the user list window.

Figure 5: The user list window.

The user window list will look like figure 5 if your server has the default installation configuration. This is a list of all users known to the server. The green circle to the left of each user name indicates that the user is active and they will be able to log into the server. If the circle is red, the user has been made inactive and will not be allowed to log into the server.

Figure 6: Opening a new user dialog.

The next step will be to make a user account for yourself. From the Users menu, select the New option as shown in figure 6.

Figure 7: Completing the new user information.

Enter the user name you would like to have in the name field. In tutorials the user joe smith will be a placeholder for your chosen user name. You can enter a description for yourself if desired. The description is optional. A password is required and must be entered twice in the two lines adjacent to the password label. You will be prompted if the two entries do not match. Choose a password that you can remember. The password for joe smith is 123. Check can save models so that, in addition to just logging in to the server, file space on the server will be allocated so that you can deploy models that you create onto the server. Similarly, check can save playspaces so that you will be able to deploy playspaces onto the server. Finally, press the add button and the new user will be added to the server database.

Figure 8: Opening the edit user window.

Next, you will edit the the newly defined user. Click on joe smith in the Users List window and then select edit from the Users menu as show in figure 8.

Figure 9: The edit user window and the view/edit groups window.

The edit user window in figure 9 allows you to change: the user's description; password; group membership; whether they can save models or playspaces on the server, and whether they are active or inactive. User names are not editable. You can either delete an old user or make them username inactive so they cannot log into the server. Click on the view/edit groups button and the view/edit groups window will open as shown on the right of figure 9.

Figure 10: Adding the user to the tutorialGroup.

You will use the view/edit groups window to add joe smith to the tutorialGroup. This will allow joe smith to access models that have been provided on the server for several tutorials. Members of the tutorialGroup were given permission to use the tutorial models. Click on the tutorialGroup so that it is selected as shown in figure 10. Then click on the left arrow to make joe smith a member of the group.

Figure 11: joe smith is a member of the tutorialGroup.

As shown in figure 11, joe smith will be a member of the tutorial group once you press the commit button. Commit the changes now. To complete step 1 of this tutorial, click OK in the edit user window and the close button on the user list window. If you have been completing this tutorial to provide yourself with a user account for the DOME models tutorial, you can return to the DOME models tutorial now.

Step 2: Creating groups and adding users to groups. You are currently logged in as a member of the administrators group, which gives you access to define users and groups. You will now go through a similar process to create a new group on the server and add users to the groups.

Figure 12: Opening the List Groups window.

Use the List Groups item in the groups menu to open the group listing window as shown in figure 12.

Figure 13: Opening the new group window.

If your server is in the default installation configuration, there will be only two groups. The other group available is the tutorialGroup of which joe smith is now a member. To add a new group, select new in the Groups menu as shown in figure 13.

Figure 14: Competing information needed to make a new group window.

Complete the information as shown in figure 14 to define the new group. Remember that joe smith is a placeholder for the user name that you created in step 1 of this tutorial. Permit space for deployment into the joe smith friends group by checking can save models and can save playspaces. When complete, press the add button.

Figure 15: Opening the edit group window.

Now, select the joe smith friends group and choose Edit from the Groups menu so that you can add users to the group.

Figure 16: Adding joe smith and tutorialUser to the group.

Press the view/edit member buttons, and then select joe smith from the list of users and groups on the right of the view/edit members window. Press the left arrow button circled in figure 16 to move joe smith into the members column. joe smith is now a member of the group, as shown in the figure. You may also add tutorialUser to the group by repeating the process.

Step 3: Changing your administrator password. If the server you are running for this tutorial will remain available to other users who should not have administration permissions, you should change the password for the user root that you are currently logged in under.

Figure 17: Changing the password for the user you are currently logged in under.

To do this, first select change password from the Options menu (you could also choose to edit the user root from the users list). This is shown in figure 17.

Figure 18: Entering the new password.

Figure 18 shows the the change password window for the user root. Type in the existing password (cadlab) in the old password field and then type in the new password twice, once in each of the two new password fields. Be sure to use a password that you can remember! When you are done press the OK button.

Step 4: Managing server file space.

As a member of the administrators group, you have access to all of the different users and groups file space on the server. At present the server file management tools are very primitive.

Figure 19: Opening the manage file space window.

Select manage file space from the Options menu to open a file space window.

Figure 20: View of the server file space.

The window will open displaying the server file space for individual use. If you expand the folders as shown in figure 20, you will recognize this as the file space you navigated in the running models tutorials. The server space is synonymous with the file space provided for the administrators group. Only members of the administrators group can deploy to, and edit within, this file space. The editing controls are in the bottom right of the window. Models, projects and folders can be deleted by first selecting and then pressing the delete button. Folders can be deleted only if they are empty. The public and private folders cannot be deleted. Public means that any user with permission will be able to browse the folder in run mode to view content within (provided that permissions for the models/projects within allow the given user to see them). Private means that only members can browse items within the folder. Only folders can be renamed. This is done by selecting the folder and pressing the rename button. If you want to rename a model, project, or playspace you will need to change the name in build mode and then redeploy. The folder button in the bottom right is used to add new folders. New folders can be added to folders only. Select the folder you would like to put a new folder into and then press the folder button.

Figure 21: Switching to the user file space.

As administrator, you can manage the file space of all users on the server. Figure 21 shows what the user file space will look like for a default installation where the tutorials have been followed. You can switch to the user file space using the combination box as shown in the figure. You may recall there are other users with permission to log into the server (e.g., guest, tutorialUser, root) but these users do not appear in the window. This is because these users did not have can save models or can save playspace permissions. Therefore they do not appear in the file space listing. You can see joe smith's private directory only because you are logged in as the administrator root.

Figure 22: Adding a folder to joe smith's public folder.

If the Public folder for joe smith is empty (meaning that you have not completed the DOME models tutorials yet), select the folder and click on the folder button. You will be prompted to name the new folder. Give the new folder the name shown in the figure and press the add button. You will be able to use this folder when you complete the DOME modeling tutorials.

Figure 23: The new folder in joe smith's user space.

There will now be a tutorial models folder in joe smith's public directory as seen in figure 23.

Figure 24: Switching to the groups file space.

As administrator you can also access the groups file space. If you are following the tutorials on a default installation of the server you will see that there are two groups with file space on the server—administrators and joe smith friends. Expand the administrators group as shown in the figure. Note that the administrators group file space looks just like the server file space. This is because the server file space is mapped to the space of the administrators group. This is the space that guest users are first directed to when they log into a server using the run mode application.

Figure 25: Switching to the collaborative use space.

Figure 22 shows groups with a collaborative use space. Use the view combination box to change to the collaborative use file space. The administrators group is expanded to show the tutorial example playspace that is used in the running models as a user tutorial. The collaborative use space can be managed in a manner similar to the individual use file space.

Figure 26: Completing the tutorial by logging out.

Close the file management window using the X in the window title bar. Logout as shown in figure 26 to complete this tutorial.

Server Management As a User This tutorial is intended to provide an overview of server administrative capabilities available to you as a user on a dome server. This includes managing your password and your server file space (if you have been provided with one).

Step 1: Changing your password. It is assumed that you have a local server running on your computer and that you are running the DOME client application. It also assumes that you have a user account on the server. If you do not have a user account, see step 1 of the server management as an administrator tutorial.

Figure 1: Switching the DOME client to server mode.

Make sure that your DOME client is in server mode using the mode combination box as shown in figure 1.

Figure 2: Opening the login window.

Use the login option under the Server menu to bring up a login window (figure 2).

Figure 3: Logging in as joe smith.

If you just completed the server management as administrator tutorial or the DOME model, project, or playspaces tutorials you will probably have your user login cached, as illustrated in figure 3 (remember that joe smith is the tutorial placeholder for your user name). If your login is cached, select it in the recent logins combination box and then press the login button highlighted in the top of the window. If your login is not cached, complete the login information in the bottom part of the window and press the lower login button. You do not need to specify the server location to log into a local server running on the same computer as your client application.

Figure 4: Opening the change password window.

Select change password from the Options menu as in figure 4 to open the password change window.

Figure 5: Entering the new password.

Figure 5 shows the change password window for the user joe smith. Type in your existing password (123 for joe smith) in the old password field and then type in your new password twice, once in each of the two new password fields. Be sure to use a password that you can remember! When you are done press the OK button.

Step 2: Managing server file space. If you have permission to deploy models and playspaces onto a server, you will want to manage your file space. At present the file management tools are very primitive.

Figure 6: Opening the manage file space window.

Select manage file space from the Options menu to open a file space window.

Figure 7: View of the server file space for joe smith.

The window will open displaying your file space for individual use. If you expand the folders as shown in figure 7, you should be able to recognize it as your space on the server. The editing controls are in the bottom right of the window. Models, projects and folders can be deleted by first selecting and then pressing the delete button. Folders can be deleted only if they are empty. The public and private folders cannot be deleted. Public means that any user with permission will be able to browse the folder in run mode to view content within (provided that permissions for the models/projects within allow the given user to see them). Private means that only you can browse items within the folder. Only folders can be renamed. This is done by selecting the folder and pressing the rename button. If you want to rename a model, project, or playspace you will need to change the name in build mode and then redeploy. The folder button in the bottom right is used to add new folders. New folders can be added to folders only. Select the folder you would like to put a new folder into and then press the folder button.

Figure 8: Switching to the collaborative use space.

If you have a file space for collaboration playspaces on the server, you will be able to use the view combination box to switch to this file space, as shown in figure 8. This file space can be managed in the same manner as the individual use file space.

Figure 9: Switching to the groups file space.

If you are a member of any groups with file space on the server, you will be able to switch to the groups file space as shown in figure 9. Only groups that you are a member of will be shown.

Figure 10: Completing the tutorial by logging out.

Close the file management window using the X in the window title bar. Logout as shown in figure 10 to complete this tutorial.

Plugin Models

This section of the tutorial is intended to familiarize you with how to build and deploy plugin models. All of the plugin tutorials assume that you are familiar with materials covered in the running models tutorials and that you have completed the first DOME model tutorial to the extent that you are comfortable with adding parameters to models, defining causality in relations, creating model interfaces, and deploying models. Building and deploying a plugin model is quite similar to building a DOME-native model, with the exception that relations cannot be added to a plugin model. You can think of a plugin model as being similar to a single DOME relation, with the relation's computation being performed by a 3rd party software application.

Tutorials are available for:

- Excel

Excel Models This section of the tutorial is intended to familiarize you with how to build and deploy Excel-based plugin models. Please check that you have covered the prerequisites as outlined on the main plugin tutorial page. The tutorials also assume that you are familiar with Excel.

Tutorials Available:

1: Excel model of rectangular solid with hole. This tutorial covers building a simple spreadsheet in Excel, wrapping it in a plugin model, creating interfaces for the model, deploying the model and setting use permissions, and running the model.

Excel Model of a Rectangular Solid with a Hole This tutorial covers several aspects of building Excel models. You will model the volume and cost of a rectangle with a hole through its center. The model will be functionally equivalent to the model you made in the first DOME model tutorial. This will allow you to compare the two implementations.

The tutorial is divided into the following sequential steps.

Step 1. Making the spreadsheet: You will make a spreadsheet to represent the volume and cost of the rectangle with a hole.

Step 2. Wrapping the Excel model: You will open a new Excel plugin model, add parameters to the model and connect them to cells in the Excel spreadsheet, define the model's causality, and configure the model so that it can locate your spreadsheet.

Step 3. Building interfaces for the model: You will create an interface for the model that defines a view that will be used to execute the model in run mode.

Step 4. Deploying the model: You will deploy the model into your user account and set access permissions on the deployed model.

Step 5. Running the model: You will execute the newly deployed Excel model and compare it with the functionally equivalent DOME model created in an earlier tutorial.

Step 1: Building the spreadsheet In this step you will make the spreadsheet needed to compute the volume and cost of the rectangular solid with a hole. The goal of this step is to prepare a spreadsheet so that it can be wrapped and deployed for use in DOME in the subsequent steps of the tutorial. If you do not want to build your own spreadsheet, you can find a prepared file in the DOME/tutorialModels/plugins/excel/rectSolid folder.

Figure 1: Excel spreadsheet for the rectangular solid with a hole.

The spreadsheet needed for the tutorial is shown in figure 1. If you build the spreadsheet yourself, it is recommended that you copy the spreadsheet exactly (including cell locations) so that your steps in later portions of the exercise will match the tutorial information. The cell equations are as follows.

width: $D2 = 2 * B2$

length: $D3 = 3 * B2$

volume: $D4 = D3 * D2 * B4 - (B4 * B2^2 * PI()) / 4$

cost: $D5 = D4 * B5$

Step 2: Wrapping the Excel Model In this step you will use build mode to start a new Excel model and create a DOME wrapper model for the block with hole spreadsheet. In this step you will: add parameters and map them to cells in the spreadsheet; define causality information for the model; and provide setup information so that DOME can find the excel spreadsheet.

Figure 1: Opening a new Excel model.

Begin by switching to the build mode application and use the Build→New model→Excel Model menu item, as in figure 1, to start a new Excel wrapper model.

Figure 2: Adding a context to the model.

Instead of adding parameters directly to the model, you will start by adding a context to organize parameters in the build view. Use the Add→Context menu item as shown in figure 2.

Figure 3: Renaming the context.

As shown in figure 3, give the context the name 'Independent parameters'. You will put the model inputs in this context.

Figure 4: Two real parameters added to the context.

With the Independent parameters context expanded and selected, Use the Add→Real menu item and add two parameters to the context. Rename the parameters as shown in figure 4. Set the values and units to match those expected in the Excel spreadsheet.

Figure 5: Adding a second context to the model.

Now, add a second context to the model as shown in figure 5. Name the context ‘Derived parameters’.

Figure 6: Additional real parameters added to the model.

Using the Add→Real menu item to add four additional parameters to the Derived parameters context. Name them and assign units as shown in figure 6.

Figure 7: Adding cell references to map the parameters.

The next step is to provide information so that DOME can connect the model parameter values with the spreadsheet data. This mapping information has been added to the model in figure 7. With the spreadsheet open as a guide, enter the sheet and cell reference into the right column for all parameters in the wrapper model. For example, the hole diameter value is in Sheet1, cell B2—or using correct syntax Sheet1!B2.

Figure 8: Defining the model causality.

Next, switch to the model causality tab and define the causality as shown in figure 8. Pressing the Edit causality information button opens the editor window. This information is needed so that DOME can correctly solve systems of models.

Figure 9: Graph visualization of the model

Once defined, you might want to switch back to the definition tab and switch to the Model Causality View or the graph visualization to verify the model causality, as shown in figure 9.

Figure 10: Opening the model file list.

The next step is to associate the spreadsheet file with the wrapper model. Switch to the setup tab as shown in figure 10. Expand the files context and double click on the icon for the model files parameter.

Figure 11: Adding files to the model file list.

The model file parameter is a list. This list parameter will be used to define which Excel files are to be associated with the wrapper model. First, make sure the parameter type combination box in the upper right of the model file window is set to file. This combination box is selected in figure 11. It should be set to file by default. Then press the add button to add a file parameter to the list.

Figure 12: Editing the path of the file parameter.

Rename the new file parameter to match figure 12. Then, click in the value field and press the choose button to locate the Excel spreadsheet.

Figure 13: Selecting the spreadsheet file.

Use the file chooser to navigate to and select the blockWithHole.xls spreadsheet file that is on your computer.

Figure 14: File parameter with path to the excel spreadsheet.

The value of Block and hole spreadsheet parameter is now the path to the spreadsheet file. Since there are no other files associated with the Excel model, close the model files window.

Figure 15: Completing the remainder of the setup information.

The main model file is a parameter selected from the model file list. Since there is only one file parameter in this case, it is already set to the Block and hole spreadsheet parameter. Moved files can be executed being true means that a copy of your excel spreadsheet file will be deployed to the server and executed within the server file space. Your local copy of the spreadsheet will not be used during model execution. Make consistent when loaded is false. This means that the model will not automatically run when you open its interface. It will update correctly when there is a manual submission of values to the model. The software version needs to be set to the version of Excel that is being used on the server where the model will be deployed. Change this to match your version. Run in foreground is false. This means that when users execute the wrapped Excel model on the server, Excel will run in the background.

Figure 16: Renaming the model.

Switch back to the definition tab as shown in figure 16. Rename the model to ‘Excel rectangular solid with hole’.

Figure 17: Saving the model.

Use the Excel Model→Save menu to open the file save dialog.

Figure 18: Naming the model file

Give the model file the name shown in figure 18, and then save the model. The wrapper model is now defined. The next step will be to create an interface for the wrapper model.

Step 3: Building an Interface In this step you will create an interface for the model that you completed in step 2. Interfaces are the exposed views of a model made available to run-mode users. The process of building interfaces is the same for DOME models, plugin models, and integration projects. Therefore, this step of the tutorial is a quick summary of the process.

Figure 1: Opening the interfaces tool.

Begin by opening the interfaces tool using the Tools→Interfaces menu shown in figure 1.

Figure 2: Add a new interface.

Rather than using the default interface, we will create a new interface from scratch. Use the Interfaces→New menu item as shown in figure 2.

Figure 3: Renaming the new interface.

Give the new interface the name ‘volume and cost interface’ as in figure 3. Double click on the interface icon to open it in a new window.

Figure 4: Add context to the interface.

Use the Add→Context menu item to add two context to the interface. Name them as shown in figure 4.

Figure 5: Adding model parameters to the clipboard.

Switch back to the model window and select the two independent parameters hole diameter and block height. Copy them into the clipboard using the Edit→copy menu item as shown in figure 5.

Figure 6: Adding the copied parameters to the interface.

Return to the open interface window, expand and select the free dimensions context, and then use the Add→Add and Map→Last selection menu item to add the parameters to the interface.

Figure 7: The completed interface.

Repeat the process of copying parameters in the model into the clipboard and then Add and Mapping them into the interface until the interface is complete, as shown in figure 7.

Figure 8: Save the model.

Finally, use the Excel Model→Save menu item to save your interface changes. The wrapper model is now ready for deployment.

Step 4: Deploying the Excel Model In this step you will move the model and interface files that you saved in step 3 to a server where they can be executed by remote users. The deployment process is the same for any type of model, so it is covered briefly in this tutorial. Remember, you need to deploy the Excel model to a server running the version of Excel that you chose when you completed figure 15 in step 2.

Figure 1: Opening the deploy model wizard.

Switch to the Deploy mode application and use the Deploy→Model menu item to open the model deployment wizard (figure 1).

Figure 2: Log into the server with your user account.

Log into the server that you have been using for the tutorials, as in figure 2. joe smith is the tutorial placeholder for your username.

Figure 3: Select the Excel Wrapper model.

Proceed to the Select model stage of the wizard. Browse to select the model definition file as is shown in figure 3.

Figure 4: Add a description for the model.

In figure 4 the description 'My first excel model' has been added as a description to appear with the model's name on the server. Proceed to the next step.

Figure 5: Selecting the deployment location on the server.

Select your tutorial models folder as the model's destination on the server, as has been done in figure 5. Proceed to the next step.

Figure 6: Editing permissions.

Leave yourself with editing permissions as in figure 6. Editing permissions were outlined in figure 8 of step 4 in the DOME model version of the rectangular solid with hole tutorial. Proceed to the next step.

Figure 7: Making the volume and cost interface available.

Check the volume and cost interface so that it will be available for use on the server as shown in figure 7. Proceed to the next step.

Figure 8: Setting user permissions.

Leave yourself with use permissions for the interface, as is the case in figure 8. Use permissions were discussed in figure 10 of step 4 in the DOME model version of the rectangular solid with hole tutorial. Proceed to the next step.

Figure 9: Summary of deployment choices.

After reviewing the summary of your deployment options, as shown in figure 9, deploy the model to the server. The wrapper model definition file and a copy of the excel spreadsheet will be uploaded to the server. The assigned use permissions will be committed to the server database. You are now ready to run the wrapped Excel model.

Step 5: Running the Excel Model In this step you will run the wrapped excel model that you deployed in step 4. You will be able to compare this model with the DOME-native version of the model. In general, this material has been covered in the running published models tutorials so explanations in this tutorial are brief.

Figure 1: Run browser logged into your user account.

In run mode, log into your user account and navigate to your tutorial models, as shown in figure 1. joe smith is the tutorial placeholder for your user name. If you completed the DOME model tutorial number 1, you should see that both the Excel version of the rectangular solid with hole model and the DOME implemented version of the model are available. Select the interface of the wrapped Excel model and open it by either double clicking on the icon, or by using the Browser->open menu.

Figure 2: Interface to the Excel model open in build view.

You may now make changes and submit them to execute the model as in figure 2. Note the Excel version of the model is, from a user perspective, indistinguishable from the DOME implemented version (you may want to run the DOME version now for comparison). Thus, the Excel-based models can be used or subscribed to without any special distinction.

This completes the tutorial.

Developer Guide

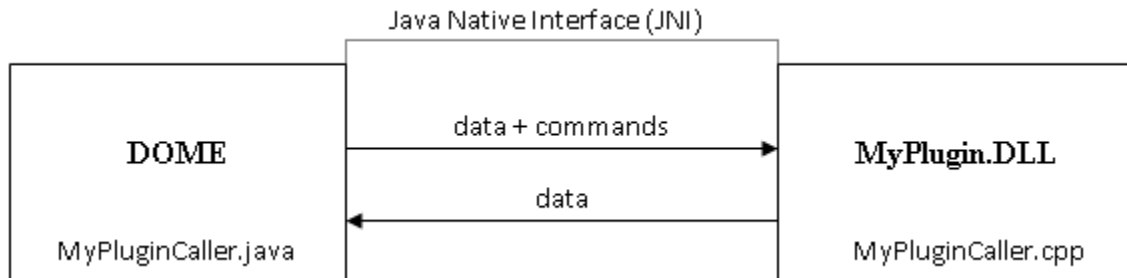
The developer guide provides API documentation to be used by software programmers developing new applications intended to interact with DOME.

Plugin Developer Guide

The developer guide provides documentation to be used by software programmers developing new plugins for DOME.

Overview of the DOME Plug-in API DOME employs a plug-in API to communicate with third-party applications such as Excel, Matlab, various CAD tools, and any other application that exposes a programming interface. On one side of the API is DOME's Java code. On the other side is a dynamically linked library (DLL) or shared library (for Linux/Unix) written in C++.

The plug-in API uses the Java Native Interface (JNI) to transport data and commands from the DOME environment to the plug-in libraries. The libraries in turn talk directly to the third-party applications using vendor-supplied API's. Finally, the JNI transport is used to deliver any results back to DOME. A simple depiction of this interaction is shown in the following figure.



Plug-in Structure The structure of a DOME plug-in was designed with the intention of being as uncomplicated as possible while shielding the author from having to understand JNI. On the Java side, all that is needed is a class (e.g., `MyPluginCaller.java`) containing one or more native methods. (A method is made native using the native keyword, much like static or public.) On the C/C++ side, a corresponding set of C functions (`MyPluginCaller.cpp`) is required.

The functions are generic enough to accommodate any plug-in. For this reason, the same functions are used for all plug-ins, and each function takes the same list of arguments. Only the names and the return values of the functions are different. The list is as follows:

```
void callVoidFunc (long ptr, int func, Object[] args)
boolean callBoolFunc (long ptr, int func, Object[] args)
int callIntFunc (...)
double callDoubleFunc (...)
String callStringFunc (...)
long callObjectFunc (...)
double[] callDoubleArrayFunc (...)
double[][] call2DimDoubleArrayFunc (...)
```

Each of these generic functions is used to access more specific functions located in the plug-in model and data classes. For example, all functions that return a void result are accessed through `callVoidFunc`, whereas all functions that return a boolean result are accessed through `callBoolFunc`, and so on.

The rest of the plug-in structure is illustrated in the figure below. In addition to the plug-in caller, there are three primary classes: the delegate (`MyPluginDelegate`), the model (`MyPluginModel`) and the data (`MyPluginData`). The delegate translates the input parameters from the generic functions into commands, model and data objects, and data

values. The delegate also instantiates the model and data objects, and passes the commands and data values to those objects where the useful work is performed.

	<p style="text-align: center;">MyPluginCaller.java / MyPluginCaller.cpp</p> <p>This pair of classes represents the interface between the Java and C++ sides of the plug-in. Both classes necessarily contain the same set of functions. Their purpose is to pass data from the Java environment to the native (C++) environment and back again. The functions are generic: they all take the same set of input arguments and are distinguished only by their different return types. Each function does nothing more than supply its input arguments to the plug-in delegate (below) and return arguments back to the Java environment.</p>	
<p style="text-align: center;">JNIHelper</p> <p style="text-align: center;">Utility class</p>	<p style="text-align: center;">MyPluginDelegate.cpp</p> <p>The delegate receives calls from the MyPluginCaller and delegates the useful work to the model and data classes (which themselves are instantiated inside the delegate.) Input parameters supplied by the MyPluginCaller are unwrapped (using JNIHelper) and used to call functions inside the model and data classes to perform a particular task. A large amount of error checking takes place in the delegate, whereas almost no error checking occurs in the MyPluginCaller.</p>	
	<p style="text-align: center;">MyPluginModel.cpp</p> <p>The C++ representation of the plug-in model. This class makes direct calls to the third-party API to process or act on the model parameters.</p>	<p style="text-align: center;">MyPluginData.cpp</p> <p>The C++ representation of the model parameters. This class is responsible for passing data to and from the third-party API.</p>

Putting it together: an example using the CATIA plug-in This section uses the completed CATIA plug-in to illustrate a working example of the plug-in structure. It starts with an example of how to prepare the Java and C++ classes involved in the JNI transport. This is followed by an execution sequence walkthrough, which demonstrates the plug-in programming template. Areas where the plug-in author is required to modify code will be pointed out here as well as in clearly marked comments in the source files.

Preparing the Java side The CATIAPluginCaller class (in package mit.cadlab.dome3.plugin.catia) contains, in addition to the generic native functions described above, a set of user-defined constants, shown below. Each constant refers to a specific function in either the model class (CATIAModel.cpp) or data class (CATIAData.cpp) of the plug-in library. It is up to the plug-in author to decide which functions he or she needs and to create a constant for each. An abridged set of constants is shown below. These particular constants, which are mapped to functions that create, load, execute and destroy the plug-in model, are common to all the plug-ins developed thus far:

```
public static final int MODEL_INIT = 0;
public static final int MODEL_DESTROY = 1;
public static final int MODEL_LOAD = 2;
public static final int MODEL_UNLOAD = 3;
public static final int MODEL_IS_LOADED = 4;
public static final int MODEL_EXECUTE = 5;
```

Preparing the C++ side Every plug-in requires a header file corresponding to the PluginCaller class. The header file is generated automatically using the javah utility. From the root directory where the class files are located (e.g., dome3/development/out), run the javah utility with the following usage:

```
javah mit.cadlab.dome3.plugin.catia.CATIAPuginCaller
```

A file named CATIAPuginCaller.h is produced and should be copied to the C++ project workspace. Looking inside this file, we see that the Java methods have been converted to C-style function prototypes. For example:

```
void callVoidFunc (long ptr, int func, Object[] args)
```

becomes:

```
JNIEXPORT void JNICALL Java_mit_cadlab_dome3_plugin_catia_CATIAPuginCaller_callVoidFunc (JNIEnv *,
```

Further, the constants are converted to preprocessor definitions. For example:

```
public static final int MODEL_INIT = 0;
```

becomes:

```
#define
mit_cadlab_dome3_plugin_catia_CATIAPuginCaller_MODEL_INIT 0L
```

To improve readability, the mit_cadlab_dome3_plugin_catia prefix was eliminated (using a text search and replace) from the constants. As a result, the line above gets reduced to:

```
#define CATIAPuginCaller_MODEL_INIT 0L
```

This naming simplification is recommended but entirely optional.

Note that the header file must be generated whenever the constants are modified, added or removed.

The C source file, CATIAPuginCaller.cpp, contains the bodies of the functions whose prototypes are CATIAPuginCaller.h. The source file may be reused for all plug-ins, with only minor modifications that are highlighted in the file.

Stepping through an execution sequence: Instantiating the model The following walkthrough of an execution sequence demonstrates the programming pattern that is used in all the plug-in calls. This same pattern is replicated many times over, once for each different call (the CATIA plug-in uses more than forty.) Mastery of just a single sequence prepares new plug-in authors to develop their own set of calls.

The first step in executing any plug-in model is instantiating the model itself. This requires calling the model's constructor (CATIAModel::CATIAModel), which has been assigned the constant MODEL_INIT.

Back on the Java side of the plug-in (in CATIAPugin.java), we decide which generic function is appropriate for the operation at hand. This only requires that we know which type of object must be returned. When a new model or data object is instantiated, the callObjectFunc function is required:

```
long modelPtr;
Object[] arr = new Object[2];
arr[0] = file;
arr[1] = new Boolean(isVisible);
modelPtr = callObjectFunc(0, CATIAPuginCaller.MODEL_INIT, arr);
```

The first argument to `callObjectFunc` represents an existing object. Because the model is being created for the first time, no object is required and this argument is set to zero (or null). The second argument is the specific method to be called—in this case, the model constructor. The final argument is an array containing arguments required by the constructor. When the call has completed, a variable of type `long` will be returned and stored in `modelPtr`. The variable `modelPtr` represents a C++ pointer (casted to a `long`) to the newly created model object. It will be used in subsequent calls that act on the model.

Now we turn to the C++ side of the plug-in. Once `callObjectFunc` has been invoked, control is transferred to the corresponding function in `CATIAPuginCaller.cpp`. The plug-in author need not be concerned with the functions in `CATIAPuginCaller.cpp`. There is almost nothing in this file that requires modification. All that really matters is that the functions in `CATIAPuginCaller.cpp` invoke corresponding functions in `CATIADelegate.cpp`.

In this case, `CATIADelegate::callObjectFunctions` is invoked. Here, the constant identifying the specific function (the model constructor) to call is queried in a switch statement:

```
long callObjectFunctions (JNIEnv* env, jobjectArray args,
unsigned int iNumArgs, long iObjectPtr,
unsigned int iFunctionIndex)
{
    switch (iFunctionIndex)
    {
        case CATIAPuginCaller_MODEL_INIT:
            iObjectPtr = initializeModel (env, args, iNumArgs);
            break;
        .
        .
    }
    return iObjectPtr;
}
```

Assuming that the constant is `MODEL_INIT`, the `initializeModel` function will be invoked. As it unpacks the argument list passed in from Java, the `initializeModel` function performs necessary error checking. It ensures that the number of arguments is correct and that those arguments are of the proper type. Once satisfied, it instantiates a new `CATIAModel` object and returns the object pointer (casting it to a `long`) to be transported back to the Java side.

This is the basic pattern for each of the plug-in calls. An author can start on a new plug-in by copying the CATIA plug-in source files and renaming the files (`MyPluginCaller.cpp`, `MyPluginDelegate.cpp`, etc.) Comments in the source files indicate where modifications are required.

It is worth discussing briefly how the Java-side arguments are unpacked. The file `JNIHelper.cpp` contains a function called `getArgument` that is used to extract individual parameters from an array of arguments passed in from Java. Continuing with our example, the argument list for the model constructor contains two arguments: a string containing the model's file name and a boolean indicating whether the CATIA window should be displayed. These two parameters are extracted in the following calls to `getArgument`:

```
void* szFileName = NULL;
void* bIsVisible = NULL;
err1 = getArgument (env, args, 0, JAVA_STRING, szFileName);
err2 = getArgument (env, args, 1, JAVA_BOOLEAN, bIsVisible);
```

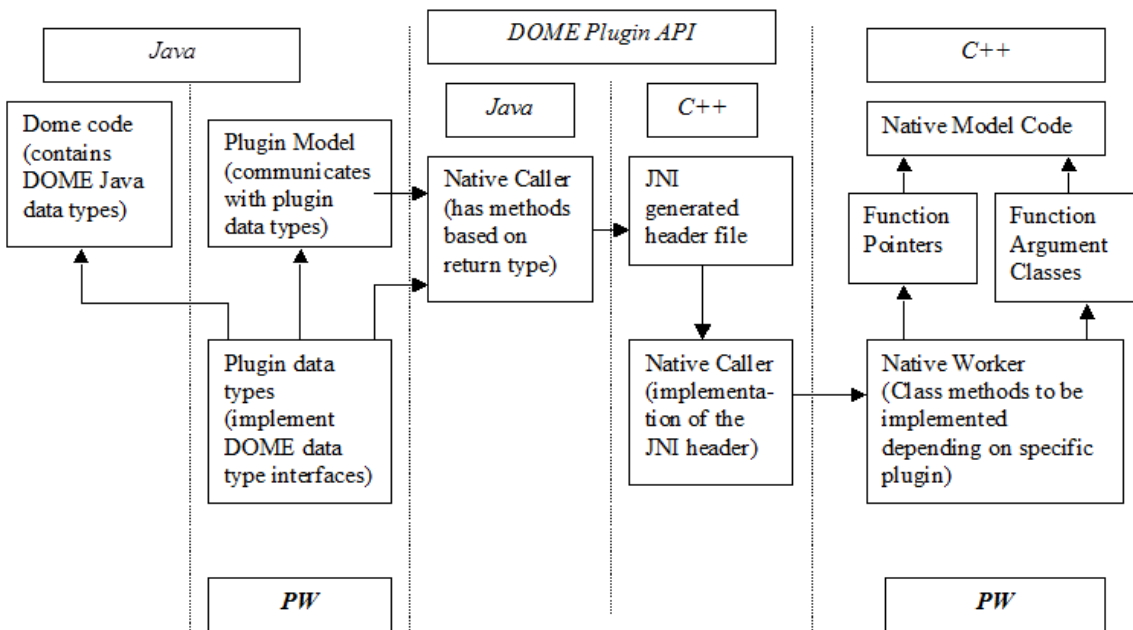
The variable `args` is the C++/JNI representation of the argument list. The parameters must be extracted in the order they appear in the list, which is determined in `CATIAPugin.java`, the Java-side caller (refer to the example on page 4). Note that `getArgument` is a generic function that returns all extracted parameters as heap objects (e.g., `new char[10]`). This

means that the resulting objects must be destroyed when they are no longer needed. The success of the `getArgument` operation is returned and should be analyzed before assuming the arguments were extracted properly.

Developing Plugin Models for DOME

Introduction This guide explains how to write plugin models for DOME using the plugin API. The plugin API encapsulates JNI (Java Native Interface) calls to the native code so that the plugin writer is not required to know JNI.

The architecture section of the guide explains what pieces of code are to be reused and what pieces need to be implemented by the plugin writer. The section, writing plugin code, gives a step-by-step procedure for plugin model code development and the last section in the document enlists a few possible pitfalls to be avoided during development.



Architecture

Note: PW indicates the parts of the code to be implemented by the plugin writer.

The above architecture block diagram is self-explanatory.

The plugin writer needs to implement the plugin model and the plugin data types in Java. The plugin model should implement the plugin interface and the plugin data types should implement the corresponding DOME data type interfaces. For example, a `PluginReal` class should implement the `DomeReal` interface and so on. Please see the Example Plugin code that accompanying the document for this purpose.

The plugin model and data types call methods in the Java side `NativeCaller` class, which will seamlessly call the native methods. The plugin writer should include the provided `NativeCaller` class's Java and C++ side implementation in their source code tree for the above-mentioned seamless native method calls.

On the C++ or the native side, the plugin writer is expected to write all the classes of her plugin model code, the function pointers to functions in this plugin code and classes encapsulating function arguments. She also needs to implement the `NativeWorker` classes' methods.

Writing plugin code The code snippets mentioned in this section are from the Example plugin code, which shows plugin writers how to write plugin models for DOME.

Following is recommended sequence of steps for the development of the DOME plugin models.

Step 1 Write the native code for the plugin model. It is recommended to write the native model code and test it thoroughly. In the Example plugin, ExampleModel.cpp, ExampleData.cpp files contain the native model and data types.

Step 2 Write the function pointers. Once the native model and data types in step 1 have been finalized, the plugin writer should write the function pointers for the native model and data types methods, which will be later called from Java. In the Example plugin code, file TFuncutor.h contains the definition of function pointers. The base class TFuncutor defines pure virtual (i.e. abstract) methods. Each subclass of TFuncutor then provides real implementation for one of the virtual methods. Other method implementations in a particular subclass are dummy and correct program flow will actually never reach inside those methods. Since the subclasses of TFuncutor uses template classes, all classes are put together in a header file TFuncutor.h.

It is up to the plugin writer whether she wants to use an inheritance scheme like one used in the Example plugin. It is not mandatory to write Functor classes but function pointers are required. Thus the plugin writer is free to provide the function pointers in a different way.

Step3 Write the argument holder classes. Two files, FuncArgument.h and FuncArgument.cpp, define two classes. They are Args and Argument. Plugin writer should include these files in their project without any modification. In order to pass the arguments to the model methods, plugin writers need to write a number of argument holder classes. In the Example plugin the argument holder classes are defined in MethodArgs.h file and are implemented in MethodArgs.cpp file. The argument holder classes written by plugin writer must extend from the Argument class defined in FuncArgument.h and provide implementation for the storeArgument method.

These argument holder classes are used by the NativeCaller to store argument values. NativeCaller handles JNI for the plugin writer and it should be used in the plugin development without any modification.

Step 4 Write NativeWorker method implementations. NativeWorker.h file defines the NativeWorker class. This file should not be modified as NativeCaller uses it. The plugin writer only needs to do the following in NativeWorker.cpp.

1. Include the Model header at the start of NativeWorker.cpp.
2. If the plugin writer wishes, she could define method name constants and enumeration constants for argument information classes. To generate an argument information object, use Args class in FuncArgument.h. Please see NativeWorker.cpp in the Example plugin.
3. Provide implementation for initializeArgInfoMap method. The ArgInfo Map is used by NativeCaller to gather information about the method arguments. Hence correct implementation of this method is crucial.
4. Provide implementation for addToArgMap method. In this method implementation, correct argument holder class should be instantiated depending on the function name passed in. This method is used by NativeCaller to create argument holder objects, which will be later used to store argument values.
5. Provide implementation for initializeFuncMap method. Here the function pointers need to be stored in the funMap data structure.
6. Finally, the plugin writer needs to provide implementation for the following methods:
callObjectFunction, callVoidFunction, callBoolFunction, callDoubleFunction and callIntFunction.

The method names are based on the return types. Inside these methods, use the function pointers and argument holder objects to invoke appropriate methods. Note that by the time program flow reaches in these methods, NativeCaller would have already populated the argument holder objects with argument values.

1. The implementation of the following methods should not be changed as they are crucial to correct operation of NativeCaller:

```
NativeWorker();
~NativeWorker();
Args* getArgInfo(const string function);
bool argInfoMapContainsKey(const string func);
void storeArgument(const string funcname, int index, void* argument);
```

This completes the native side implementation of the plugin.

Step 5 Write Java side plugin data type classes. These classes should in turn call the native methods using NativeCaller class on the Java side. Please see ExampleBoolean.java, ExampleReal.java in the Example plugin code given. In the example classes provided, each method e.g. getValue, setValue etc. is overloaded. One method operates on underlying Java data whereas other method invokes the actual native method. This way all the data is loaded on Java side only and transferred to native side only when the execute method on the plugin model is called.

Step 6 Write Java side plugin model. This class must implement the Plugin interface provided on the Java side. Each plugin method should call the corresponding native method. The execute method should load all the java data to native side before native side execution and transfer the results back to Java side after execution is complete. Please see ExamplePlugin.java in the Example plugin code for reference.

Step 7 Test the plugin code. It is advisable to test the plugin thoroughly after each step and after all the steps are completed.

Note that the C++ code should be compiled as a DLL (Dynamic Link Library) on Windows platform, as a LIB file on Solaris platform and so on. When this native library file name is passed to the plugin class constructor as an argument, do not include the file extension and preceding dot. Also this file should be included in system's path environment so that JNI (Java Native Interface) would be able to locate this file. If Java environment cannot find this file, java.lang.UnsatisfiedLinkError exception will be thrown.

Possible Pitfalls

- Function Pointers: function signatures should exactly match signatures in function pointers otherwise results can be unpredictable. E.g. although "string" STL type can be interchangeably used for char*, it may not work right in case of function pointers.
- EXCEPTION_ACCESS_VIOLATION error on Java side could be due to Null Pointers or unfilled array elements on the native side.
- If the native side does not throw Exceptions with explanatory messages and/or print debugging information, troubleshooting the plugin code can become difficult and time consuming process.

DOMÉ Reference

Topics

DOMÉ API

Package Descriptions

Topics

cadlab.dome3**Attention:** Sub folder indentations need work

Package Name	Description
api	Facade classes to browse DOME server and execute DOME Interfaces
config	Implementation of DOME class registry
gui	Classes for Run/Build/Deploy/Sever Mgmt GUI
help	Classes for help system and help contents
network	
client	
conference	Classes for Conference with playspace participants
connection	Connection management w/ XML-RPC
functions	Classes for DOME client to use functions provided by a DOME server through XML-RPC communication
handlers	XML-RPC handlers implementing functions provided by a DOME client
objectrecord	Data objects used in a Tree to keep track of Models and Model objects
server	
conference	Conference with playspace participants
db	Query/Insert/Update to DOME DB
functions	Classes for DOME server to use functions provided by other DOME servers or a DOME client through XML-RPC
handlers	XML-RPC handlers implementing functions provided by DOME server
objectmodel	
dataobject	Interface definition and Class implementation of DOME data objects: Real, Integer, Boolean, Matrix, Vector, File
exceptions	Exception classes used in DOME classes
model	Implementation of common behaviors of DOME Models and Plugin Model in the Runtime- and Build- Mode
modelcomponent	Classes for Model Components. Model Component is an item associated with a particular Model, including Filter
modelinterface	Classes for Model Interface. Model Interface is a collection of model objects – such as parameters and relations –
modelobject	
context	Classes for Context, used to organize model objects.
parameter	Classes for Parameter, representing a dataobject or set of dataobjects.
relation	Classes for Relation, a collection of ModelObjects and associated special behaviors
subscription	Classes for Subscription, a kind of relation defined by subscribed Model Interface
visualization	Classes for Visualization, which accepts data from parameters and transform them into a various graphical visual
plasyapce	Classes for playspace
project	Classes for Integration Object, which allows one to specify a set of resources to tie together via subscriptions in i
toolinterface	Classes for analysis tool interface and optimization tool interface
util	Classes for causality solving, mapping management, and ID generation
plugin	Classes for various plugin models: CATIA, ABAQUS, etc.
search	???
swing	Swing classes for DOME GUI. Some Swing classes are specialized for listener mechanism used in DOME, such
tool	Classes for DOME tool implementations such as the QMOO optimization tool
util	Utility classes used in DOME development: logging, units, and XML

Plugins

Contribute

The documentation on this site is auto generated from this repository on BitBucket. You can contribute by submitting pull requests to contribute content or post corrections on the issue tracker.