Weevil User Manual

Yanyan Wang

Software Engineering Research Laboratory
Department of Computer Science
University of Colorado
Boulder, Colorado 80309-0430 USA
ywang@cs.colorado.edu

1 Weevil

A particular experiment is related to three primary concepts: the system under experimentation (SUE), the testbed and the actor. An actor maps a client to its system access point and actuates the SUE as dictated by the client's workload. An experiment can be understood as a two-phase process in which (1) a workload is generated for the actors, (2) the SUE is deployed on the testbed with the actors actuating it based on the workload and the results are collected afterward. Weevil supports the two-phase process through a central-controller architecture in which a master script controls the whole process. Workload generation is conducted on the master's host as presented in Section 2. In the experiment deployment and execution phase (described in Section 3) the master generates all the controls, deploys them together with the workload and the SUE on the testbed, coordinates all the system components and the actors to execute the experiment, and gathers data after the experiment terminates. The two program *weevilgen* and *weevil*, are for workload generation and experiment deployment and execution respectively.

Throughout the following sections, we mainly refer to a Siena example included in the prototype distribution under *share/weevil-1.1.0/examples*. For some features not used in the Siena example, we refer to a Chord and a Squid/Apache example.

2 Simulation-Based Workload Generation

Weevil's simulation-based workload generation process supported by the package *weevilgen* is illustrated in Figure 1.

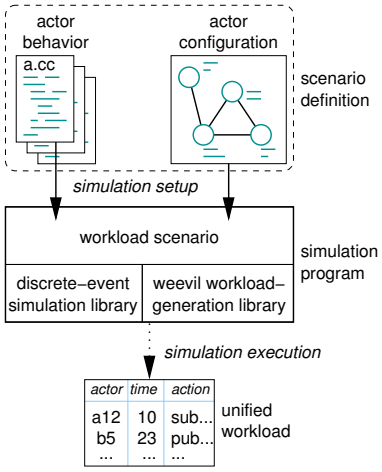

Figure 1: Simulation-Based Workload Generation

In detail, the following steps need to be taken in order.

2.1 Weevilgen Configuration (Workload Registration)

Create a name for the workload (refered to as <*workload name*> below), like "`MyWorkload`" in the Siena example; add the name into "*weevil.conf*" file after "`weevilgen_WORKLOADS:=`". You could have more than one workload registered in "*weevil.conf*" separated with space, as indicated in the apachesquid example. To start a new line, \ is needed to end the current line. Weevilgen only generates workloads that have been registered in "*weevil.conf*".

For example, in file "*weevil.conf*" of the Squid/Apache example we registered a series of workloads:

```
weevilgen_WORKLOADS := wkld_10_10_10 wkld_20_20_20 wkld_30_30_30 \
wkld_50_50_50 wkld_100_100_100 wkld_150_150_150 wkld_200_200_200 wkld_30 \
wkld_60 wkld_90 wkld_150 wkld_300 wkld_450 wkld_600 wkld_tp wkld_gsl
```

Then, to generate each workload, Weevil must be provided with programmed actor behavior models as illustrated in Section 2.2 and an actor configuration file named "<*workload name*>*.m4*" including all the actor configurations as illustrated in Section 2.3.

2.2 Actor Behavior Model Programming

One or more types of actor behaviors could be programmed in C++ with the support of Weevil's workload-generation library and SSim simulation library, and may therefore execute arbitrary functions and maintain arbitrary state. SSim is a discrete-event simulation library that supports message communication between simulated processes, so actor behavior programs may specify interactions with other actors. Weevil's workload-generation library extends the SSim's library by providing a workload-output method and convenient methods for dealing with processes by ids. Please refer to the *Weevil's workload-generation library* and the *SSim's simulation library* in the reference manual when programming your actor behavior model.

Basicly, actor behavior is encapsulated in a subclass of the `weevil::WeevilTProcess` class or the `weevil::WeevilProcess` class. `weevil::WeevilTProcess`, which is a *sequential process*, is the extension of the `ssim::TProcess` class. A sequential actor behavior is defined simply by programming the `main` method, which defines the body of the process directly. A process can use the `WeevilGen::broadcast_event`, `WeevilGen::self_signal_event` and `WeevilGen::signal_event` method defined in Weevil's library to signal itself or other processes, and the `wait_for_event` method to receive events.

In contrast, `weevil::WeevilProcess`, which is a *reactive process*, is the extension of the `ssim::Process` class. It is used when the process may have other reactive behaviors in parallel to its routine behaviors. A reactive actor behavior is defined by programming the `init` and the `process_event` methods. `Init` initiates the process just before the simulation starts. `Process_event` is called automatically by an event and defines the functions executed by the process in response to an event. Same as the `weevil::WeevilTProcess`, the event can be signaled using the `WeevilGen::broadcast_event`, `WeevilGen::self_signal_event` and `WeevilGen::signal_event` methods.

Both of the above two types of processes can use the method `workload_output()` to output actions to the workload. As examples of the usage of these two types, we consider the following two types of interdependent behaviors.

2.2.1 Intra-Actor Interdependent Behavior

One type of interdependent behavior is one where future requests need to incorporate data from a previous request. As an example, consider a distributed publish/subscribe system where actors use an access point to publish notifications and subscribe for notifications of interest. A common behavior for a subscriber actor could be to periodically subscribe and unsubscribe, with every unsubscription matching the previous subscription, then continue with the next subscription. We implemented this behavior in *examples/siena/subscriber.cc*. (It is used to generate workload *wkld_sequential.wkld*.) Notice that in this case, the behavior is represented as a sequential process class and is defined simply by programming the `main` method of that class.

Similarly, Weevil can easily generate workloads for various types of session-oriented protocols, where actors maintain a virtual connection with the system by providing an immutable session identifier throughout their interaction with the system.

2.2.2 Inter-Actor Interdependent Behavior

Another type of interdependent behavior is one where actors communicate and coordinate their interactions with the SUE. We consider a scenario in which three siena clients are involved. They have their routine behaviors. C1 subscribes 50 times, C2 publishes 40 times, and C3 publishes 20 times. Besides, they also communicate and may have reactive actions triggered by others.

To generate a workload, we consider each client as a reactive workload process `SienaClient`, the subscription, notification, and unsubscription as the possible actions of that process to be outputed to a workload file. The communication between clients is represented by a `Trigger` object. The configuration of each workload process consists

of the number of its routine actions, the intervals between actions, those parameters used to generate the content of subscriptions and notifications, and the list of clients to whom the current client will trigger after its own action.

The code to implement this behavior model could be found in *examples/siena/sienaclient.cc*. (It is used to generate workload *MyWorkload.wkld*.) Notice that in this case, the behavior is defined as a reactive process by implementing the `init` and `process_event` methods. The `Trigger` class represents an event signaled to a simulation process. The `init` method initializes workload processes just before the simulation starts but after the assignment of process properties. Notice also that the configuration of each workload process is implemented with the help of actor configurations described later in Section 2.3. Corresponding to each property declared in the actor configurations, a class variable and a class function need to be defined in the behavior program, such as,

```
//class variable
unsigned int m_constr_min;
//class function
void constr_min( unsigned int cmin )
{ m_constr_min = cmin; }
```

in the file "*sienaclient.cc*" corresponds to the property configuration

```
WVL_SYS_WorkloadProcessProp( 'C1', 'constr_min', '1')dnl
WVL_SYS_WorkloadProcessProp( 'C2', 'constr_min', '2')dnl
WVL_SYS_WorkloadProcessProp( 'C3', 'constr_min', '1')dnl
```

in the actor configuration file "*MyWorkload.m4*".

2.3 Actor Configurations

After programming the actor behavior models, you can populate a scenario consisting of many actor instances specified in the actor configuration file named "<*workload name*>*.m4*". The actor behavior models and the actor configurations make up a workload scenario definition.

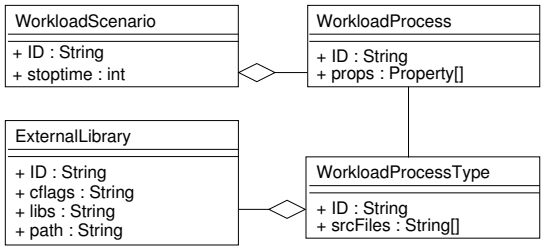

Figure 2: Workload Scenario Conceptual Model

Figure 2 shows the portion of the Weevil conceptual model concerning the definition of workload scenarios. You need to parameterize the following GNU m4 declaration macros in the actor configuration file. The order of the declarations does not matter. A macro can be used before it is defined as long as it is defined somewhere in the actor configuration file. Weevil supports the engineer during this activity by performing extensive checks on the syntax and consistency of the configurations and by providing detailed error messages about any problems it encounters.

- `WVL_SYS_WorkloadScenario(<ID>, <length>, <processes>)`

  A *WorkloadScenario* has an identifier and a collection of *processes*. The argument <`length` > defines the number of clock ticks in the simulation. If you define this argument, the simulation will end as long as the clock-tick number is reached even if the actors have not finished their work. For example,

```
define(‘clients’, ‘C1, C2, C3’)dnl
WVL_SYS_WorkloadScenario( ‘MyWorkload’, ‘10000’, ‘clients’)dnl
```

- `WVL_SYS_WorkloadProcessType(<ID>, <sourceFiles>, <libraries>)`

  A *WorkloadProcessType* defines an actor behavior model in a list of *sourceFiles* as described in Section 2.2, and may be associated with a collection of *libraries* that contain external dependencies of the implementation. Please note that none of the file names of the .cc files in the *sourceFiles* list can be the same as the workload name you registered in the *weevil.conf* since a <*workload name*>*.cc* will be generated in the workload generation process. The source files should list all the C++ files in the order of file inclusion. For example,

```
WVL_SYS_WorkloadProcessType( ‘SienaClient’, ‘gen_message.h, gen_message.cc, sienaclient.cc’, ‘’)dnl
```

  In this example, instead of adding the statement `#include ‘‘gen_message.h’’` in both *gen_message.cc* and *sienaclient.cc*, we only need to put *gen_message.h* before *gen_message.cc* and *sienaclient.cc* in the <*source files*> parameter of this declaration macro.

  The Squid/Apache's *wkld_gsl* example that generates workloads based on analytic models uses the GNU scientific library *libgsl* declared later:

```
WVL_SYS_WorkloadProcessType( ‘AnalysisBrowser’, ‘analysisbrowser.cc’, ‘libgsl’)dnl
```

- `WVL_SYS_ExternalLibrary(<ID>, <cflags>, <libs>, <path>)`

  This declaration defines the usage of an external library, including its cflags, its libs, and the path to add into the environment variable *LD_LIBRARY_PATH*. For example, the external library "*libgsl*" used above is declared as follows:

```
WVL_SYS_ExternalLibrary( ‘libgsl’, ‘-I/usr/include -L/usr/lib’, ‘-lgsl -lgslcblas -lm’, ‘/usr/lib’)dnl
```

- `WVL_SYS_WorkloadProcess(<ID>, <type>)`

  Each *WorkloadProcess* represents an actor, which is an instance of a *WorkloadProcessType*. For example,

```
define(‘clients’, ‘C1, C2, C3’)dnl
WVL_SYS_Foreach(‘i’, ‘WVL_SYS_WorkloadProcess( i, ‘SienaClient’)’, clients)dnl
```

  defines three instances of the *WorkloadProcessType* `SienaClient` (`C1, C2, C3`).

- `WVL_SYS_WorkloadProcessProp(<workloadProcessID>, <name>, <value>)`

  This declaration macro is used to configure each defined process. Corresponding to each property, a class variable and a class function need to be defined in the actor behavior program to assign the value of the property to the class variable. Besides the example we gave in Section 2.2, another example could be the property"`sub_neighbors`":

```
WVL_SYS_WorkloadProcessProp( ‘C1’, ‘sub_neighbors’, ‘"C2 PUB 3 0.02|C3 SUB 2 0.01|"’)dnl
WVL_SYS_WorkloadProcessProp( ‘C2’, ‘sub_neighbors’, ‘"C3 SUB 2 0.02|"’)dnl
WVL_SYS_WorkloadProcessProp( ‘C3’, ‘sub_neighbors’, ‘"C1 PUB 5 0.01|"’)dnl
```

corresponds to the class variable “`m_sub_neighbors`” and function “`sub_neighbors`” in program “*sienaclient.cc*”. The function “`sub_neighbors`” parses the string in the parameter <`value`> to assign the variable “`m_sub_neighbors`”.

2.4 workload generation

With the actor behavior models programmed and actor instances configured for a workload scenario, Weevil checks them for consistency and then translates them into an executable simulation program that is linked with the libraries and then executed to produce the desired workload named “<*workload name*>*.wkld*”. The workload consists of all interactions between actors and the SUE. These interactions represent service calls that must be applied to the SUE during experiment execution. This process is accomplished through the command “*weevilgen gen-*<*workload name*>”.

```
examples/apachesquid> ${execPath}/weevilgen help
Welcome to WEEVIL WORKLOAD GENERATOR.

The following commands are available:

  check-all            - checks all workloads.
  check-<workload>     - checks a workload’s settings.
  clean-all            - cleans files for all workloads.
  clean-<workload>     - removes all generated files for a workload.
  help                 - prints this help message
  gen-all              - generates all workloads.
  gen-<workload>       - generates a workload.
  version              - prints the version of weevil workload generator

The following workloads are available:

wkld_10_10_10 wkld_20_20_20 wkld_30_30_30 wkld_50_50_50
wkld_100_100_100 wkld_150_150_150 wkld_200_200_200 wkld_30 wkld_60
wkld_90 wkld_150 wkld_300 wkld_450 wkld_600 wkld_tp

examples/apachesquid> ${execPath}/weevilgen gen-wkld_10_10_10
```

Besides the “*gen-*<*workload name*>” option, you can also use the other options listed above for different functions.

3 Experiment Deployment and Execution

Weevil directly supports experiment deployment and execution. The overall process is depicted in Figure 3. Actions are represented by rectangles and are labeled by circled numbers. Input and output data for those actions are represented by ovals. Dark ovals represent input models provided by the engineer. White ovals represent control scripts and data files generated by Weevil. The cross-hatched ovals represent data generated by the SUE during an experiment. Solid arrows represent normal input/output data flow, whereas dotted arrows represent the execution of scripts.

The following steps need to be taken to setup and conduct an experiment.

3.1 Weevil Configuration (Experiment Registration)

Create a name for the experiment (refered to as <*experiment name*> below), like ”`experiment`” in the siena example; add the name into ”*weevil.conf*” file after ”`weevil_EXPERIMENTS:=`”. You could have more than one

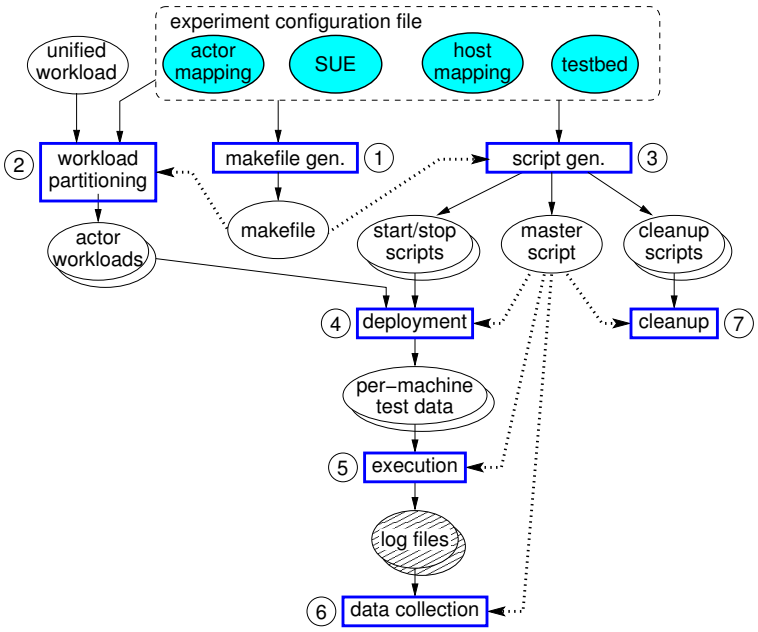

Figure 3: Weevil Experimentation Process

experiment registered in "*weevil.conf*" separated with space, as indicated in the apachesquid example. To start a new line, \ is needed to end the current line. weevil only conducts experiments that have been registered in "*weevil.conf*".

For example, in file "*weevil.conf*" of the Squid/Apache example we defined a series of experiments:

```
weevil_EXPERIMENTS := exp_10_10_10 exp_20_20_20 exp_30_30_30 \
exp_50_50_50 exp_100_100_100 exp_150_150_150 exp_200_200_200 exp_30 \
exp_60 exp_90 exp_150 exp_300 exp_450 exp_600
```

For each experiment, Weevil must be provided with a workload file (refer to Section 3.2) and an experiment configuration file named "*<experiment name>.m4*" including all the experiment configurations as illustrated in Section 3.3.

3.2 Workload File

You can generate the workload file with *weevilgen*. Of course, you can always use workloads from other workload generators or just use a real trace as the workload. But please note that the workload should be made up of workload lines with the following format:

```
event(<time stamp>, <workload process ID>, '<event content>')
```

3.3 Experiment Configurations

These configurations are represented by the dark ovals along the top of Figure 3. They are programmed in GNU m4 by parameterizing Weevil-defined declaration macros (Please refer to the "Weevil's Experiment Environment Declaration Macros") to instantiate elements of two conceptual models (SUE and testbed) and necessary mappings (Please refer to the "Weevil's Experiment Environment Conceptual Models"). In other words, these declaration macros will define a set of macros serving as properties of an experiment. (Please refer to the "Weevil's Experiment Environment Declaration Macros" and the "Some Other Weevil-Defined Property Macros") for these property macros.) The order of the declarations does not matter. A macro can be used before it is defined as long as it is defined somewhere in

the experiment configuration file. Weevil supports the engineer during this activity by performing extensive checks on the syntax and consistency of the configurations and by providing detailed error messages about any problems it encounters. These declaration macros are defined below:

Experiment `WVL_SYS_Experiment(<ID>, <workload>, <actors>, <testbed>, <sue>, <componentHosts>, <clean>), <parallel>, <monitors>, <timeout>))`

This declaration macro defines the experiment to be conducted, including its workload, actor mapping, monitor mapping, testbed, SUE, and host mapping. All of these properties will be specified using further declaration macros. Note that the `ID` of an experiment can be different from the experiment name you registered in the "*weevil.conf*" file. The last parameter *clean* specifies if the workspace of the experiment is to be removed after the experiment terminates. Since Weevil support parallel communication from master host to each testbed host, the parameter `parallel` defines the maximum number of remote call processes that can be safely sent out in parallel from the master host. The last parameter, `timeout`, defines the latency of the experiment expressed in seconds. After the timeout, the experiment will be terminated even if the workload has not finished yet. For example, the experiment with name "*experiment*" in Siena example is defined as:

```
WVL_SYS_Experiment('Exp_0', 'MyWorkload', 'D0, D1, D2', 'Bed0', 'Siena', 'CHMap0, CHMap1, CHMap2', 'Y', '30', '', '2000')
```

Workload `WVL_SYS_Workload(<ID>, <filename>, <processes>)`

This declaration macro further specifies the workload. It accociates the workload ID with a specific workload file. It also points out the workload processes included in the workload. For example, the workload in Siena example is defined as:

```
WVL_SYS_Workload('MyWorkload', 'MyWorkload.wkld', 'C1, C2, C3')dnl
```

Testbed Model Weevil configures the testbed as a set of hosts accessiable through user-level remote shell access, no matter the testbed is a local network, an emulated testbed like *Emulab*, or the wide-area network testbed like *PlanetLab*.

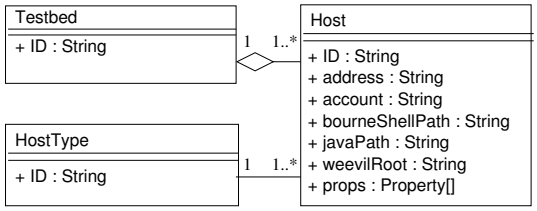

Figure 4: Testbed Conceptual Model

The declaration macros included in this model are:

- `WVL_SYS_Testbed(<ID>, <hosts>)`

  As shown in Figure 4, a *Testbed* has an identifier and a collection of *Host*s. For example,

  ```
  WVL_SYS_Testbed( 'Bed0', 'H0, H1, H2')dnl
  ```

- `WVL_SYS_HostType(<ID>)`

  The *hostType* attribute is used to partition the hosts into categories that each of them has its own binary package of the SUE. For example, a software may have different binary packages if it is compiled on different operating

systems. So you should just create an ID for the host type and define it through this declaration macro. Then this ID could be used later in the *WVL_SYS_Host* macro. For example,

```
WVL_SYS_HostType( 'FreeBSD' )dnl
WVL_SYS_HostType( 'Linux' )dnl
```

- `WVL_SYS_Host(<ID>, <address>, <account>, <bourneShell>, <java>, <weevilRoot>, <type>)`

  Each host in a testbed with the network *address* can be accessed through a user *account*. In PlanetLab the account is actually the PlanetLab slice name to which the engineer is assigned. Weevil uses the Bourne shell for experiment framework execution. Thus, each *Host* has the attribute *bourneShellPath* to provide its local path to the program *sh*. The *javaPath* attribute is needed if there is any Java command or Java implementation in component start/stop scripts or actor binary distribution, as described below. *weevilRoot* specifies the workspace on the host assigned for the experiment. At last, by assigning a host to a specific *HostType*, the proper software binary distribution could be used.

- `WVL_SYS_HostProp(<hostID>, <propertyName>, <propertyValue>)`

  Besides the above required properties, A *host* can also have an optional list of properties that can be used to contain any system-specific information that must be known for each host.

SUE Model The conceptual model of an SUE is shown in Figure 5.

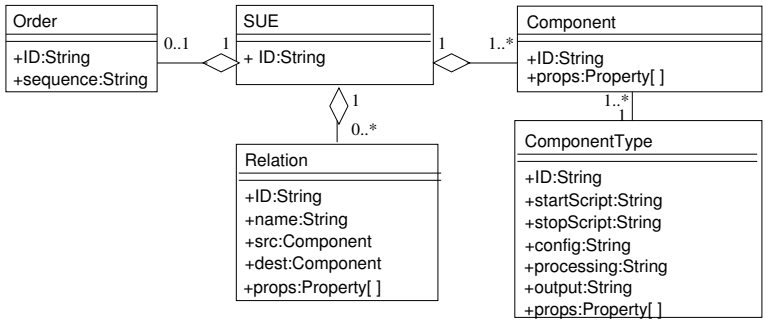

Figure 5: SUE Conceptual Model

The declaration macros included in this model are:

- `WVL_SYS_SUE(<ID>, <components>, <relations>, <order>, <prestart>)`

  An *SUE* is comprised of typed *Components*. They can be related through *relations*. For some experiments, some components are required to start before the others. The parameter, *order*, is used to specify the start-up order. The last parameter, *prestart*, specifies the components started before the actors begin to send out requests. It currently should be the same as the parameter *components* since this Weevil version does not support SUE changes at run-time. The *relations* and the *order* are optional arguments. For example,

```
WVL_SYS_SUE( 'Siena', 'S0, S1, S2', 'R10, R20', 'Order', 'Prestart')dnl
```

- `WVL_SYS_ComponentType(<ID>, <startScript>, <stopScript>, <startArgs>, <config>, <logs>)`

There could be different types of components in an SUE, such as Apache servers and Squid proxies in the Squid/Apache experiments. All the instances of a particular ComponentType share the same formats of start/stop commands, configuration files, and log file names. Thus, for each *ComponentType*, you should define its *startScript* and *stopScript*, and optionally a *config* attribute that contains the contents of a configuration file. This design allows common attributes to be shared among all components of a type. For different experiments targeting the same system, an engineer would typically need to make minor changes to other entities without having to modify *ComponentType* attributes. The attribute *startArgs* provides the value of some arguments to the start script at the beginning of the experiments. The value may change later if the SUE can be changed at run time. But for the current version, the arguments have the same value along the experiment. The last parameter, *logs* specifies the log file names to collect after the experiment.

All the Weevil-defined property macros can be referenced in these parameters, and are resolved during script generation using m4's macro expansion. (Please refer to the reference manual for the Weevil-defined property macros.)

The following example is for the experiments of Squid/Apache system. There are two types of components, `ApacheServer` and `SquidProxy`. Apache could be started under its *bin* directory through the command "*apachectl -k start*", while Squid could be started under its *sbin* directory through command "*squid -N -d 1*". Thus we defined `ApacheServer_startScript`, `SquidProxy_startScript` with some macros to customize to different components. Similar for their stop scripts. Apache and Squid are both configured through configuration files. Thus, we changed their default configuration files by adding macros to it to customize to different components. (Please refer to files *examples/apachesquid/httpd.conf* and *examples/apachesquid/squid.conf*.)

```
WVL_SYS_ComponentType(‘ApacheServer’, ‘ApacheServer_startScript’, ‘ApacheServer_stopScript’, ‘’, dnl
‘’include(‘httpd.conf’)’’, dnl
‘theCompPath/WVL_Experiment_Name‘_’WVL_Component_ID‘_error.log’, dnl
theCompPath/WVL_Experiment_Name‘_’WVL_Component_ID‘_access.log’’)dnl

WVL_SYS_ComponentType(‘SquidProxy’, ‘SquidProxy_startScript’, ‘SquidProxy_stopScript’, ‘’, dnl
‘’include(‘squid.conf’)’’, dnl
‘theCompPath/WVL_Experiment_Name‘_’WVL_Component_ID‘_access.log’, dnl
theCompPath/WVL_Experiment_Name‘_’WVL_Component_ID‘_cache.log’, dnl
theCompPath/WVL_Experiment_Name‘_’WVL_Component_ID‘_store.log’’)dnl

dnl
dnl -----ApacheServer Start/Stop Script-----
dnl
define(‘ApacheServer_startScript’, dnl
‘cd WVL_SYS_Echo(‘WVL_Host_’WVL_SYS_Echo(theHost)‘_apacheRoot’)/bin
./apachectl -k start -f theCompPath/WVL_Experiment_Name‘_’WVL_Component_ID.conf &’)dnl

define(‘ApacheServer_stopScript’, dnl
‘cd WVL_SYS_Echo(‘WVL_Host_’WVL_SYS_Echo(theHost)‘_apacheRoot’)/bin
./apachectl -k stop -f theCompPath/WVL_Experiment_Name‘_’WVL_Component_ID.conf’)dnl

dnl
dnl -----SquidProxy Start/Stop Script-----
dnl
define(‘SquidProxy_startScript’, dnl
‘rm -rf WVL_SYS_Echo(‘WVL_Host_’WVL_SYS_Echo(theHost)‘_squidRoot’)/var/cache
WVL_SYS_Echo(‘WVL_Host_’WVL_SYS_Echo(theHost)‘_squidRoot’)/sbin/squid -z
WVL_SYS_Echo(‘WVL_Host_’WVL_SYS_Echo(theHost)‘_squidRoot’)/sbin/squid dnl
-N -d 1 -f theCompPath/WVL_Experiment_Name‘_’WVL_Component_ID.conf &’)dnl

define(‘SquidProxy_stopScript’, dnl
‘read pid <theCompPath/WVL_SYS_Echo(WVL_Experiment_Name)‘_’WVL_Component_ID.pid
echo $pid
kill $pid’)dnl
```

- `WVL_SYS_ComponentTypeProp(<componentTypeID>, <propertyName>, <propertyValue>)`

  ComponentType allows system-specific properties to be assigned to all the instances of this type.

- `WVL_SYS_Component(<ID>, <Type>)`

  This declaration just defines each component of the SUE as an instance of a componenttype. For example,

```
WVL_SYS_Component(‘S0’, ‘ApacheServer’)dnl
define(‘Proxies’, ‘P0, P1, P2’)dnl
WVL_SYS_Foreach(‘i’, ‘WVL_SYS_Component(i, ‘SquidProxy’)’, Proxies)dnl
```

- `WVL_SYS_ComponentProp(<componentID>, <propertyName>, <propertyValue>)`

  Component allows system-specific properties to be assigned to it.

  For example,

```
WVL_SYS_ComponentProp(‘S0’, ‘Port’, 4321)dnl
WVL_SYS_ComponentProp(‘S1’, ‘Port’, 5432)dnl
WVL_SYS_ComponentProp(‘S0’, ‘Protocol’, ‘ka’)dnl
...
```

- `WVL_SYS_ComponentRelation(<ID>, <name>, <src>, <dest>)`

  The *Relations* contained in an SUE model are used to represent any binary associations between components. These are optional and entirely system specific. For instance, in the Squid/Apache example, the proxies works cooperatively. Thus, each proxy component has all the other proxy components as its siblings.

```
WVL_SYS_ComponentRelation(‘RP0P1’, ‘sibling’, ‘P0’, ‘P1’)dnl
WVL_SYS_ComponentRelation(‘RP0P2’, ‘sibling’, ‘P0’, ‘P2’)dnl
WVL_SYS_ComponentRelation(‘RP1P0’, ‘sibling’, ‘P1’, ‘P0’)dnl
WVL_SYS_ComponentRelation(‘RP1P2’, ‘sibling’, ‘P1’, ‘P2’)dnl
WVL_SYS_ComponentRelation(‘RP2P0’, ‘sibling’, ‘P2’, ‘P0’)dnl
WVL_SYS_ComponentRelation(‘RP2P1’, ‘sibling’, ‘P2’, ‘P1’)dnl
```

- `WVL_SYS_ComponentRelationProp(<componentRelationID>, <propertyName>, <propertyValue>)`

  This declaration macro defines any other properties related to a *Relation*.

- `WVL_SYS_ComponentOrder(<ID>, <sequence>)`

  The *order* contained in an SUE model are used to represent the necessary (or preferred) order to start all the components. This is optional and entirely system specific. Some SUEs require some components to be ready before other components, such as Siena. It requires a parent server to be ready before its children start (since they need to communicate with their parent server as soon as they start). Then you should specify the start order of all the components like,

```
WVL_SYS_ComponentOrder(‘Order’, ‘S0, 3, S1, 0, S2, 0’)dnl
```

  In this example, `S0` is the parent of `S1` and `S2`. Please refer to the format of the second argument of this declaration macro. Components are listed in the start order separated with commas. The number between successive component IDs represents the time to wait to start the next component. The number after the last component ID represents the time to wait to start actors. It is useful if a component needs some time to get ready. It can be zero if the start process is very quick or the order of the successive components does not matter (like `S1` and `S2` in this example).

But for other SUEs, the order does not matter. For instance, in the Squid/Apache example, the proxies works cooperatively. But it does not require a component's sibling to start before it. Then you need not declare the order with this declaration macro. Weevil will start all the components in the order they are defined. Of course, you can always specify a preferred order in such cases.

- `WVL_SYS_ComponentPreStart(<ID>, <components>)`

  The *prestart* contained in an SUE model are used to specify the components to be started before the actors begin to send out requests to the SUE. But since the current released version of Weevil does not support dynamic changes to the SUE during experiment execution, the *components* list should be exactly the same as the list of all the SUE components in an experiment.

```
WVL_SYS_ComponentPreStart('Prestart', 'S0, S1, S2')dnl
```

Mappings The two models described above and the workload are largely independent. This independence means that they can be composed together in many different combinations by providing two mappings between them.

The first mapping simply associates each *component* in the SUE with a *host* in the testbed.

- `WVL_SYS_ComponentHost(<ID>, <component>, <host>)`

  For example,

```
WVL_SYS_ComponentHost('CHMap0', 'S0', 'H0')dnl
```

- `WVL_SYS_ComponentHostProp(<componentHostID>, <propertyName>, <propertyValue>)`

  If there is any extra limitation related to a component-host mapping. You could specify it here.

- `WVL_SYS_ComponentHostType(<componentTypeID>, <hostTypeID>, <binaryDistDir>)`

  This declaration macro maps a ComponentType to a HostType. Since the same type of components share the same binary distribution for the same *HostType*. Thus, For each ComponentType-HostType pair, if the software is installed through copying its binary distribution from the master machine to the testbed hosts, this macro needs to be called to assign a different binary distribution for each ComponentType-HostType pair. For example,

```
WVL_SYS_ComponentHostType( 'SienaServer', 'FreeBSD', '/scratch/software/Siena')dnl
WVL_SYS_ComponentHostType( 'SienaServer', 'Linux', '/scratch/software/Siena')dnl
```

  The values of `<binaryDistDir>` are the same since Siena has the same binary distribution for FreeBSD machine and Linux machine.

- `WVL_SYS_ComponentHostTypeProp(<componentHostID>, <hostTypeID>, <propertyName>, <propertyValue>)`

  This declaration macro defines any other properties related to a ComponentType-HostType mapping.

The second mapping associates each workload process with a component through an actor. Each workload process must be represented by a single component, but any number of workload processes can be associated with a particular component through actors.

- WVL_SYS_ActorProgram(<ID>, <style>, <binaryDistDir>, <program>, <argument>, <classpath>, <logs>, <receiveProgram>)

  An actor is implemented as a system-specific program that understands how to actuate the SUE as dictated by the workload line. In other words, you are expected to implement the function of parsing a workload line and sending out a corresponding system command. You can use defined macros in the program to customize for different actors. The program can also receive arguments as configured in the <argument> parameter. You can choose from Java and Shell for the <style> parameter to implement the program in Java or in shell script. Weevil provides a library to support its implementation in Java. (Please refer to the reference manual.) Otherwise, if you are implementing a shell script actor program, and you have *n* arguments, then *$i (i=1,...,n)* represents the *i*th argument. $(n+1) represents the workload line to parse. (Please refer to the *examples/apachesquid/actor/apachesquidactor.sh* as an example.) The <binaryDistDir> is to specify the location of the actor program. If it is a *Java* actor, then you should put the Java classname in the <program> parameter; otherwise, you should put the shell script file name in it. The parameter <classpath> is used when the execution of the Java actor program needs extra classpaths. For example, the shell script actor program declaration in Squid/Apache example is like:

```
WVL_SYS_ActorProgram('DrProgram0', 'Shell', './actor', 'apachesquidactor.sh',
'ApacheSquidActor_args', '', 'WVL_Experiment_Name'_'WVL_Actor_ID'.log'')dnl
dnl
define('ApacheSquidActor_args', 'dnl
WVL_SYS_Echo('WVL_Host_'theHost'_executablePath')/bin dnl
WVL_SYS_Echo('WVL_Host_'theHost'_address') WVL_SYS_Echo('WVL_Component_'theComponent'_Port') dnl
theActorPath')dnl
```

  The Siena's Java actor program needs the class definition in Siena software, thus:

```
WVL_SYS_ActorProgram('DrProgram1', 'Java', './actor', 'SienaActor', 'SienaActor_args',
'SienaActor_classpath', 'WVL_Experiment_Name'_'WVL_Actor_ID'.log'')dnl
dnl
define('SienaActor_args', 'WVL_SYS_Echo('WVL_Component_'theComponent'_Protocol') dnl
WVL_SYS_Echo('WVL_Host_'theHost'_address') WVL_SYS_Echo('WVL_Component_'theComponent'_Port') dnl
WVL_SYS_Echo('WVL_Actor_'WVL_Actor_ID'_Port')')dnl
dnl
define('SienaActor_classpath', 'theSoftwarePath/siena-1.5.1.jar')dnl
```

  We support run-time communication between distributed actors. The weevil-provided remote call commands are:

  theActorPath/WVL_Experiment_Name'_communicate.sh' theActorPath <a file of target actors>

  theActorPath/weevil-sendFile.sh theActorPath <a file of target actors><file to send>

  theActorPath/weevil-sendMessage.sh theActorPath <a file of target actors><arguments to send to the receiveProgram>

  The last parameter, *receiveProgram*, specifies the program that receives and handles messages from other actors.

- WVL_SYS_Actor(<ID>, <workloadProcess>, <program>, <component>, <dir>)

  This declaration macro maps a <workloadProcess>to a <component> through an actor <ID> implemented as the actor program <program>. The last parameter <dir> is the source directory of all the other files the actor needs for experiment execution. For example, the Chord actor *D0* can be defined as,

```
WVL_SYS_Actor('D0', 'C1', 'DrProgram1', 'S2', 'files/C1')dnl
```

The directory `files/C1` is the source directory where all the files to be inserted by `C1` are located.

- `WVL_SYS_ActorProp(<actorID>, <propertyName>, <propertyValue>)`

  You can specify extra actor properties through this macro. For example,

```
WVL_SYS_ActorProp('D0', 'Port', '4322')dnl
```

The last mapping associates each actor with an experiment entity (either a component or a host) through a monitor. This mapping is a many-to-many mapping. In other words, there can be more than one monitors for each actor or for each experiment entity. The monitor mapping is necessary when an actor needs to decide on its behavior based on current execution status of the experiment. The monitors can analyze the experiment execution log files and return information to the actor at run-time.

- `WVL_SYS_MonitorProgram(<ID>, <binaryDistDir>, <program>, <argument>)`

  A monitor should be implemented by experimenter as a system-specific, stand-alone shell script that parses the log files of the entity and returns information to an actor (actors). The `<binaryDistDir>` specifies the location of the monitor program. You should put the shell script file name in the parameter `program`. The program can receive arguments as configured in the `<argument>` parameter.

- `WVL_SYS_Monitor(<ID>, <program>, <entity>)`

  This declaration macro declares a monitor instance `ID` in an experiment. It runs the `program` against an experiment (entity). The actors are free to access information from it.

- `WVL_SYS_MonitorProp(<monitorID>, <propertyName>, <propertyValue>)`

  You can specify extra monitor properties through this macro.

Others Besides the above configurations of conceptual models, there may be some other configurations, such as the macro extension included in the workload. For example, in the Squid/Apache examples, their workloads have the format like

```
event(10,br10,GET(file76))
...
```

The file IDs like `file76` included in the workload are not real filenames. It can be instantiated through the configuration

```
define('files', 'WVL_SYS_Range('', 1, 200)')dnl
WVL_SYS_Foreach('i', 'define('file'i, dnl
'http://''WVL_SYS_Echo('WVL_Host_'WVL_SYS_Echo('WVL_ComponentHost_'S0'_host')'_address')':''dnl
WVL_SYS_Echo('WVL_Component_'S0'_Port')'/test/'i'.txt')', files)dnl
```

in the experiment configuration file.

3.4 Setup and Script Generation

The goal of the setup phase is to "compile" the experiment configurations into the control scripts that will be used for experiment deployment and execution. Initially, the configurations are checked for consistency, and a per-experiment Makefile is generated to control the rest of the process, which consists of actions 2 and 3 in Figure 3.

Action 2 tailors the workload based on the experiment configurations (such as the file instantiation discussed above) and partitions the unified workload to per-actor workloads. Following that, in action 3, three tailored scripts, a start script, a stop script, and a cleanup script, together with a tailored configuration file are generated for each component in the SUE by applying macro expansion to the contents of the relevant attributes of *ComponentType*. Additionally, a single master control script is created. This whole process is accomplished through the command "*weevil setup-<experiment name>*".

```
examples/apachesquid> ${execPath}/weevil help
Welcome to WEEVIL.

The following commands are available:

  check-all            - checks all experiments.
  check-<experiment>   - checks an experiment's settings.
  clean-all            - cleans all experiments.
  clean-<experiment>   - removes all generated files for an experiment.
  help                 - prints this help message
  run-all              - runs all experiments.
  run-<experiment>     - executes an experiment.
  setup-all            - sets up all experiments.
  setup-<experiment>   - generates all files for an experiment.
  version              - prints the version of weevil

The following experiments are available:

exp_10_10_10 exp_20_20_20 exp_30_30_30 exp_50_50_50 exp_100_100_100
exp_150_150_150 exp_200_200_200 exp_30 exp_60 exp_90 exp_150 exp_300
exp_450 exp_600

examples/apachesquid> ${execPath}/weevil setup-exp_10_10_10
```

3.5 Deployment and Execution

```
examples/apachesquid> ${execPath}/weevil run-exp_10_10_10
examples/apachesquid> ${execPath}/weevil resume-exp_10_10_10
```

At this point, the engineer can perform an experiment by simply executing the master control script with the above `run` command. If error happens during experiment deployment, the engineer can resume from the failure point with the above `resume` command. The master control script deploys the components of the SUE, per-actor workloads, actors, and control scripts to the hosts (action 4), then starts all the components and actors (action 5). By estimating the round-trip time between the master host and testbed hosts, the master script intelligently decides when to have actors begin processing their workloads. The master script waits for all actors to complete processing their workloads and then causes execution of the stop scripts for each of the components. After all the components terminate, post-processing scripts are executed and output is copied back to the master machine (action 6). Finally, the testbed machines are cleaned up as necessary (action 7).

The directory structure on each testbed host when weevil executes an experiment is shown in Figure 6. The property macros *theWeevilRoot*, *theCompPath*, *theSoftwarePath*, and *theActorPath* are defined correspondingly. (Please refer to the "Some Other Weevil-Defined Property Macros" reference manual.)

3.6 Clean up

After the experiment terminates, it will clean up its workspaces either on all the testbed hosts automatically specified by the value *'Y'* of the parameter *clean* of the SUE, or one by one with the generated clean scripts like,

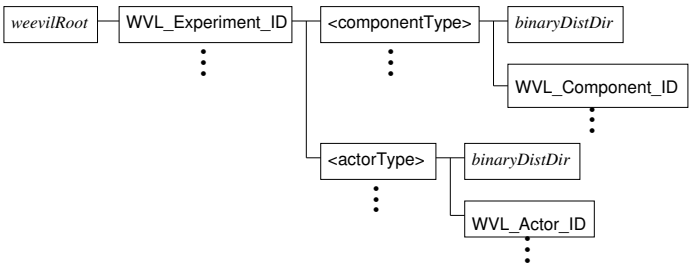

Figure 6: Run-Time Directory Structure

```
examples/apachesquid> sh exp_10_10_10_S0_clean.sh
examples/apachesquid> sh exp_10_10_10_S1_clean.sh ...
```

Besides, the master directory could be cleaned up through

```
examples/apachesquid> ${execPath}/weevil clean-exp_10_10_10
```