DEW: Distributed Experiment Workflows

Jelena Mirkovic, Genevieve Bartlett and Jim Blythe
USC Information Sciences Institute
{*sunshine, gbartlett, blythe*}*@isi.edu*

Abstract

Current testbed experiments are often ad-hoc, manual, complex and hard to repeat and reuse. This is due mostly to our current inability to capture, standardize and encode experiment behavior. We propose DEW—distributed experiment workflows. Unlike current experiment representations, which focus mostly on topology, DEW encodes experiment *behavior* and topological *constraints*, which help with realization on testbeds. We show how DEW enables easier experiment design, management, sharing and reuse, and how it can facilitate automated generation of topologies and runnable scripts.

1 Introduction

Progress in systems, networking and security fields today occurs at lightning speed, and is driven by industry and academic research. This produces a myriad of research solutions, which must be validated and evaluated for their performance in realistic working conditions. Network testbeds are often used in this process, offering free, distributed resources, which can be configured and organized into complex experiments.

In spite of the growing sophistication, number and diversity of testbed infrastructures, the way we use them remains largely ad-hoc and manual. Testbed experiments are often built from scratch, by painfully stacking and restacking pieces into a coherent whole. Experiments are run, reconfigured, and re-run hundreds of times over long time periods, using scripts that continuously evolve. Much of the knowledge needed to create an appropriate experiment for a given experimentation goal remains in the minds of the researchers. This makes testbed experiments hard to repeat, reproduce or reuse, and it limits the progress in the scientific fields that need testbeds for evaluation.

We contend that much of the current situation stems from the failure of experiment representations to capture an *experiment's behavior*. Current testbeds represent experiments mostly as a collection of resource demands, a.k.a. *topology*. The topology describes what types of resources are needed from the testbed, and how to connect and configure these resources. Once the resources are successfully allocated to the experiment, what the researcher does with them is not recorded nor structured into any meaningful representation. This leaves defining and orchestrating the *behavior* of allocated testbed nodes as a separate step, often done with a separate set of tools and custom written scripts, from the ground up and over the course of a long time.

Separating the experiment's topology from its behavior has two main drawbacks. First, this two-step process is not aligned with how researchers design experiments. A researcher starts from an experimental goal and iteratively designs a scenario, or a set of behaviors, to meet this goal. Like a screenwriter, the researcher envisions what will happen in the experiment, and how these actions will be interrelated. Our failure to encode this process means the screenplay remains buried in the researcher's mind, reduced to a set of actors (testbed nodes) with their features and locations on the set, but without the reasoning for their presence and without the lines, which they have in the play. This misalignment burdens the experiment design process, where topology is encoded by the testbed, but behavior must be manually encoded by the researcher in the form of scripts, which are then manually invoked on the topology. As experimentation progresses and the goals change, new topologies may be created, and scripts may be adjusted or reused, without any record of their interdependencies. Porting to a new testbed, scaling an experiment up or down, or repeating an existing experiment on new hardware, all require the researcher to recall these missing links, and manually address minute details of the experiment to make it valid and consistent with prior runs. This is a high cognitive burden placed on imperfect human memory, and it often leads to a lack of sharing of experiments, reuse difficulties with important pieces missing, and overly simplistic experiments, which do not sufficiently advance our science.

Second, the disconnect between topology and behavior means running experiments is a very manual and error-prone process. Even when most of the experiment's behavior is scripted, the researcher may run sequences of scripts manually, reformatting data in between, or changing portions of the scripts, to meet some immediate goal. This manual process is quickly forgotten and makes it hard to reproduce experimental results, even by their own creators. Some basic services, which are necessary for repeatability and reproducibility, are very challenging to accomplish if we do not encode the desired experiment behavior. Failure detection and failure diagnosis are difficult, because there is no record of what an experiment should do and how to measure success of these actions. Repeatability and reproducibility are hard, because scripts, created by researchers, capture only some actions taken in the experiment. Saving versions of scripts and topology, so we could refer back to them later, is problematic because we do not understand how these artifacts fit together, nor which pieces are important enough to be saved. Building community libraries of common experimental actions, or of representative experiments, is challenging, because we do not understand the high-level purpose of low-level actions (e.g., OS commands), nor how to detect if two experiments are similar.

It is, however, not enough to represent experiment behavior as merely a flat record of everything that transpired. This would create an avalanche of data, which would be very hard for humans to interpret or make use of. We must also impose some structure on the data, to make sense of the underlying actions and enable researchers to manipulate them at a high cognitive level and to drill down to details when necessary.

In this paper we propose a new representation for testbed experiments, called a Distributed Experiment Workflow, or DEW. DEW enables the researcher to abstract the definition of an experiment from its realization. It encodes the desired behavior of an experiment at a high-level as a *scenario* (e.g. "generate attack from A to B, wait 10 seconds, turn on defense at C"), and provides sufficient details as to how each action in a scenario can be realized on the testbed, via *bindings* (e.g. use script `attack.py` with specific parameters for the attack action). DEW further encodes only those features of testbed topology, which matter for the experiment, via *constraints* (e.g. "use Ubuntu OS on C").

When an experiment is to be realized on the testbed, the constraints section of DEW is used to generate a resource request for the testbed. Once the experiment is allocated—physical nodes are reserved and loaded with the operating system—the scenario and bindings are used, along with allocation details, to produce scripts, which run on the nodes.

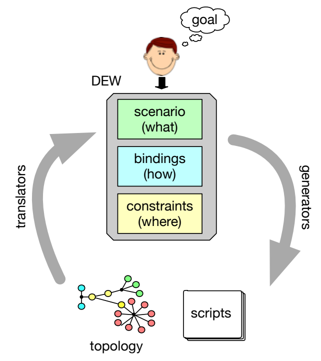


Figure 1: New experiment representation—DEW.

When the researcher runs the experiment on the testbed they parameterize and run these scripts, possibly interspersed with manual actions, which produces a *run history*. Together, the DEW representation, node allocations and the run history represent a complete record of an experiment, which can be shared and reused by others.

We first present the structure of DEW and our design decisions (Section 2). We next discuss lessons we can learn from past related work in representing experimental behavior (Section 3) and describe a plan forward for adoption of DEW (Section 4). We discuss the benefits of DEW in Section 5, and how it can help simplify experiment design, running, sharing and reuse, making experiments more robust and scientific. Our UI and preliminary companion tools are available publicly at [2].

2 DEW

In this section we describe DEW—our proposed experiment representation. DEW unifies behavior and topology of an experiment, but captures only relevant topological details. We had several design goals for DEW:

High-level representation. DEW should be a human-readable, short description of *what* the experiment is supposed to do. Such a description facilitates reuse, because researchers could, at a glance, understand the experiment's actions and judge if this experiment is useful to them. Moreover, expressing what an experiment does, in broad strokes, captures the design process, which so far has existed only in the researcher's mind.

Generic language. DEW should support many diverse experiments, and thus its language must be expressive enough to be broadly applicable, regardless of the experiment's goal or testbed infrastructure.

Self-contained representation. We wanted DEW to contain sufficient details to facilitate automated generation of experimental topologies and scripts, which would run on these topologies. If we could achieve this, then researchers could work with experiments at a high level, delegating detail-oriented and error-prone tasks to machines. This would further facilitate generation of topologies and scripts in many different languages, and for many different testbed infrastructures, enabling portability.

Decouple behavior from topology. We wanted to decouple the intended behavior from the topology where it will be realized, thus enabling the same experiment to be scaled up and down easily, by changing a few lines of DEW. Our goal was to capture only the necessary topology details in the form of constraints.

Structured representation. We wanted to impose some natural structure on DEW, enabling researchers to easily locate and focus on the important pieces for their goal. This structure should facilitate experiment reuse.

We illustrate DEW in Figure 1. DEW consists of three parts: (1) The *scenario*, which encodes at the high-level *what* the experiment is supposed to do. (2) The *bindings*, which encode *how* each action in the scenario is realized, i.e., which software is used, and how it is parameterized. Together, scenario and bindings create a self-contained representation of the experiment's behavior. (3) The *constraints*, which decouple behavior from topology, and together with scenario enable automated topology generation. DEW's structure enables a researcher to focus only on relevant, small portions of the experiment. For example, adding a new behavior would require modifications to the scenario and bindings. Changing software, which produces a certain action, would only require changing one line in the bindings. Scaling up or down would lead to changes in just the constraint section.

Figure 2 depicts our preliminary design of DEW's syntax in Extended Backus-Naur form. DEW primarily expresses what the researcher wants to accomplish in the experiment, and secondly how and where this will be accomplished. The main sections of DEW are "scenario", "bindings" and "constraints", with each section using its own language to describe the experiment.

The *scenario* consists of statements, optionally organized into sections preceded by a label. Figure 2(b) illustrates a scenario with five labels: configure, monitor, run, analyze and store. Labels could be used by the user or an experiment orchestration system to run only a subset of statements, for example several run cycles with different parameters. Each statement starts with an optional trigger, where it can wait for some time period to elapse, or for a given event(s) to occur, or both. The statement then includes an actor (usually, this would be a node in an experiment) and the action to take. The statement could end with an optional emission of one or more events, which could be used to trigger later statements.

Triggers and events enable us to specify both serial and parallel sequences of statements, as well as conditional sequences. In a parallel sequence (e.g., "configure" section in Figure 2(b)) , the statements all start without triggers, or on the same trigger. A serial sequence (e.g., "run" section in Figure 2(b)) contains statements where earlier ones emit events, which trigger later ones.

The *bindings* part consists of statements, which link each action and event to the tools/scripts and their inputs, which will realize this action or event on the testbed. Each statement specifies the action or event, the executable which realizes it, and the input parameters for this executable. Some parameters may be fixed at the experiment design time, and could be specified as constants (see line 26 in Figure 2(b)). Other parameters may be variables, whose values will be specified at runtime. We use large font size for them in Figure 2(b). The example also includes some built-in functions, such as ETH, IP, etc., which need to be evaluated at runtime, but are commonly needed by experiments, and thus should be provided by the testbed.

Current testbeds require users to fully specify their topologies, when running an experiment. This approach leads to over-specification, for example by requesting specific hardware that is known to have a fast CPU, rather than specifying CPU speed as a desired feature of a given actor. Over-specification limits the size of the candidate pool for allocation and increases probability of allocation failures [12]. It also requires user action to adapt the experiment to new resources as testbeds evolve, or to use the experiment in a different testbed environment. For these reasons DEW does not include a full topology specification, but rather specifies the *constraints* on the allocation, which talk about desired features of actors and their connections. The constraints section consists of a sequence of statements. Each statement specifies the constraint type, and the actor or several actors, to which this constraint applies. Our current design provides constructs to specify the number of nodes in a given actor role (`num`), the OS, hardware type and number of interfaces for a given actor (`os`,`type` and `interfaces`), and nodes that are connected by a link (`link`) or a LAN (`lan`). We expect that we will need to extend this set of constructs in the future.

If a constraint is not specified for a given actor, default settings are assumed. By default an actor is a single node in the topology (no `num` constraint) on any hardware (no `os`,`type` or `interfaces` constraints) and it would be connected to other nodes via a LAN (no `link` or `lan` constraints). In our example in Figure 2(b), the server runs the latest version of Ubuntu, but other nodes run any OS. We have also requested two attacker nodes, while there will be only one server and one client node. Further, there is an additional router node, which appears only in the constraints. This node has a link to the server and is on the same LAN as the attackers and the client.

```
dew          = scenario, bindings, constraints
scenario     = 'scenario:', {statement}
label        = 'configure' | 'run' | 'monitor' | 'analyze' | 'store' | string
statement    = [label,':'] , [trigger] actor, {actor}, action, [ 'emit', event ]
trigger      = [ 'when', event, {event} ] , [ 'wait', time ]
event        = string
time         = string
actor        = string
action       = string
bindings     = 'bindings',':', {statement}
bstatement   = action | event ,':', command, {command}
command      = string
constraints  = 'constraints',':', {cstatement}
cstatement   = 'num', actor, number | 'lan', actor, {actor} | 'link', actor, actor |
                 'os', actor, osname | 'type', actor, typename |
                 'interfaces', actor, number
osname       = string
typename     = string
```

(a) DEW in EBNF. For readability, we do not define strings and numbers.

```
scenario:
configure:
server install_iperf
client install_iperf
client install_flooder
server install_tcpdump

monitor:
server start_measure emit mstarted
server start_server emit sstarted

run:
when mstarted, sstarted client start_traffic emit cstarted
when cstarted wait t1 attacker start_attack emit astarted
when astarted wait t2 attacker stop_attack emit astopped
when astopped wait t3 client stop_traffic emit cstopped

analyze:
when cstopped server stop_measure emit mstopped
when mstopped server calculate_entropy

store:
server COPY_TO_GITHUB

bindings:
install_iperf apt-get install iperf -y
install_tcpdump apt-get install tcpdump -y
install_flooder /scripts/common/install_flooder.sh
start_measure tcpdump -i ETH(IP(server)) -w DUMP_OUTPUT
start_server iperf -s
start_traffic iperf -c IP(server)
start_attack flooder --dst IP(server) --proto 6 —rate RATE
stop_attack pkill -9 flooder
stop_traffic pkill -9 iperf
stop_measure pkill -9 tcpdump
calculate_entropy /scripts/entropy_calculator.sh ANALYSIS_OUTPUT
mstarted EXIST_PROCESS(tcpdump)
sstarted EXIST_PROCESS(iperf)
cstarted EXIST_PROCESS(iperf)
astarted EXIST_PROCESS(flooder)
astopped NOT EXIST_PROCESS(flooder)
cstopped NOT EXIST_PROCESS(iperf)
mstopped NOT EXIST_PROCESS(tcpdump)

constraints:
server os Ubuntu-latest
attacker num 2
lan client attacker router
link router server
```

(b) Example entropy-measurement experiment in DEW.

Figure 2: DEW's syntax in EBNF and an example experiment represented as DEW

3 Related Work

There are several related works on new experiment representations to improve sharing and reuse. Experimentation Workbench by Eide et al. [9] proposed new constructs in the NS file (Emulab testbed's topology description file) to describe experiment behavior in addition to topology. The authors also proposed keeping track of experiment resource allocations and the behavior in an "experiment record" (similar to our experiment history), and allowing users to tag manual commands that should be included in the record. This work was an inspiration for our work, but it lacks sophistication and user-friendliness that we hope to achieve. First, its experiment representation combines topology and behavior, while DEW keeps these separate by using actor roles as an abstraction for resources, and constraints to tie actor roles to topology based only on important features. This makes DEW more broadly useful, as the same scenario can be used with many different topologies. Second, Eide et al's representation of behavior includes the actual commands to be run during the experiment, while ours keeps the high-level description of behavior (scenario) separate from the commands that realize it (bindings). This makes DEW more structured and more user-friendly.

GPLMT [15] proposes a new experiment-description language. GPLMT produces experiment representations, which are much less readable by humans, more verbose and less structured than those written in DEW. For instance, their example experiment from Listing 1.1. includes 46 lines in XML. In DEW these would become six lines, with human-readable format without XML mark-up. A similar critique holds for works by Buchert et al. [5], Hussain et al. [3], Baxley et al. [4], Vrijders et al. [14], Seidel et al. [13], and Cherrueau et al. [6]. In general, we view other experiment representations—many of which are also coupled with a workflow engine—not as competitors but as potential partners in our development of DEW. First, we can easily envision building translators and generators (see Section 4.1 and 4.2) that translate their experiment representations into DEW and vice versa. Second, these works are good examples of community-specific tools, which can be built on top of our proposed experiment representation, to make it more useful for specific communities of experimenters.

Labwiki [17] and GENI desktop [11] provide an IDE-like environment for experiment design and running. The main goal of these approaches is integrating design, running and data analysis of experiments into a single environment and thus easing lifecycle management and sharing/reuse. Our GUI provides more assistance to users during experiment design, as illustrated in Section 4.3, but it otherwise shares the design goals with these prior works. One major drawback of these prior works, which design a new way to build and run experiments, is that they are slow to be adopted by existing testbed users. We discuss this problem in the next section, and how we plan to overcome it for DEW.

While experiment lifecycle support shares many goals with scientific workflows (e.g., Pegasus [7], VisTrails [10]), such as portability, repeatability, provenance, etc., there are some notable differences, which necessitate new solutions for representation and later orchestration of testbed experiments. First, scientific computation consist mostly of computation and data movement/transformation, while testbed experiments include many other tasks such as traffic generation, node and switch configuration, etc. Second, scientific experiments usually involve many nodes performing the same computation tasks, while testbed experiments involve many nodes playing different roles (e.g., server vs client vs attacker). Third, testbed experiments may require specific node features, such as connectivity, NICs, hardware, etc.

4 Easing Adoption

The main challenge when proposing a new experiment representation or a new experimentation process (e.g., via a new IDE) is how to motivate wide adoption. Novice testbed users are likely to adopt any framework easily, because they are new to experimentation and expect to have to put in time to learn new tools. But existing users, which are accustomed to specifying topologies and experimentation process manually, are very reluctant to invest time into learning new tools. Fundamentally, users prefer to continue to use what they have already built—their existing scripts and topologies.

As we discuss in the following sections, we plan to address this adoption problem by building *translators*, which can ingest existing scripts, and automatically produce DEW representation. We also plan to build *generators*, which can ingest DEW and automatically produce topologies and scripts, runnable on today's testbeds.

Another related issue, which can hinder adoption, is that different users prefer different modes of experiment design. Some like to produce textual representations of experiments, while others may want to draw them, or assemble an experiment out of smaller building blocks, or start with an existing experiment and extend it. To be successful, we must support many different experiment design approaches, as we discuss in Section 4.3.

4.1 Translators

Translators are tools that can produce DEW from the currently used scripts for testbed experimentation. Such scripts are usually over-specified with regard to parameters and the topology, associated with the script. This makes it hard for researchers to reuse scripts in a different setting. Translating scripts into DEW brakes this tendency and allows for more flexible and portable experiment representation.

We illustrate this using one of our scripts, which was used for running a flash-crowd DDoS experiment to test one of our research solutions, called FRADE. Figure 3(a) shows our original script, minus some variable definitions. There are 26 lines, many of them hard to read and interpret quickly by a human user. Further, the script is coupled with a topology—notice that it assumes existence of N nodes, N-1 of which are called "attacker$_i$", with i=0..N-1, and one whose name is specific to the experiment. All these nodes are in the specific project on the testbed. The script thus must be manually changed when we want different node names, or if we are reusing the experiment in a different project. Our script is more loosely coupled with a topology than a usual testbed script—notice that it has variable experiment name $EXP and a flexible number of attacker nodes, controlled by command line parameter $3. This is because we had to run this experiment on multiple topologies, which required multiple experiments, and we had to manually evolve our script from a simple one, runnable on one topology, to a more complex one, runnable on multiple topologies. Using DEW and our generators a researcher would get multiple instances of scripts automatically, by simply adjusting constraints in DEW.

Figure 3(b) shows the DEW corresponding to the script in Figure 3(a). We built a simple translator to generate DEW from our bash script. While translating arbitrary scripts in any language to another language is challenging, testbed scripts usually use just a subset of bash functionalities. This gives us hope that we can process them with a moderately complex translator. We were able to generate all but line 14 in Figure 3(b) automatically. We have color coded actors orange, actions green, triggers blue and purple, and events red. Bindings are shown as black. Our translator identified lines starting with `ssh`, extracted the location and the command being executed, turned the location into an actor, and the command into an action, and inferred which actions occur serially by looking if their commands were executed from the main script without spawning a new process. Such serial actions had to wait for a trigger event, generated by the preceding action. The translator also identified `sleep` commands in the original script and converted them into wait triggers.

In our generated DEW, there are three distinct actors with generic names "actor$_i$". By default, since no constraints are specified, each actor will have only one instance. If we wanted to create three attackers, we could specify this as a constraint: `num actor2 3`. Translating our script into DEW had an immediate benefit! We noticed a flaw in our original script's design. Actions in lines 1–4 of DEW all occur in parallel (in separate processes, spawned off of the main script), but starting a detector (line 4 of DEW) should not happen in parallel with restarting a server. We should have waited for the server to start instead. Conversely, actions 8–12 do not depend on each other, and could occur in parallel.

In our future work, we plan to develop translators from other scripting languages, such as MAGI [3], GPLMT [15], Baxley et al. [4], XPFlow [5], etc.

4.2 Generators

We have yet to implement generators, which produce scripts from DEW. We expect that, by reversing the process for translators, we will be able to produce relatively complete scripts, which the user will then be able to improve further manually. We plan to produce generators for many languages, such as bash, MAGI [3], GPLMT [15], Baxley et al. [4], XPFlow [5], etc.

```
ssh  -o StrictHostKeyChecking=no $1.$EXP.frade "cd ~/frade/experiments/run/;
sudo bash start_log.sh flood.$1.$2.$3.$ms" &
ssh -o StrictHostKeyChecking=no  attacker0.$EXP.frade "cd ~/frade/experiments/run/;
sudo bash start_log.sh legitimate.$1.$2.$3.$ms" &
ssh  -o StrictHostKeyChecking=no $1.$EXP.frade "cd ~/frade/experiments/run/;
sudo bash restart_server.sh" &
ssh  -o StrictHostKeyChecking=no $1.$EXP.frade "cd ~/frade/experiments/run;
sudo bash start_detector.sh \"$modules $m4\"" &
sleep 15
ssh -o StrictHostKeyChecking=no  attacker0.$EXP.frade "sudo python3.4
~/frade/traffic/smart_attacker/legitimate.py -s $1 --sessions 100 --logs
/proj/FRADE/MTurk/Normalized-logs/$1-new.log 2>&1> output &" &
sleep $INT
 j=1
while [ $j -le $3 ] ; do
   ssh -o StrictHostKeyChecking=no  attacker$j.$EXP.frade "cd ~/frade/traffic/flood_attacker/;
   sudo python3 attack.py -s $1 -n $2 -u ../urls/urls-$1.txt & " &
   j=$(($j+1))
done
sleep $DURATION
j=1
while [ $j -le $3 ] ; do
    ssh -o StrictHostKeyChecking=no  attacker$j.$EXP.frade "sudo pkill -9 python3"
    j=$(($j+1))
done
sleep $INT
ssh -o StrictHostKeyChecking=no  attacker0.$EXP.frade "sudo killall python3.4"
ssh  -o StrictHostKeyChecking=no $1.$EXP.frade "sudo pkill -9 python"
ssh  -o StrictHostKeyChecking=no $1.$EXP.frade "sudo ipset list blacklist > /zfs/FRADE/blacklist.
$1.$2.$3.$ms"
ssh  -o StrictHostKeyChecking=no $1.$EXP.frade "sudo pkill -9 tcpdump"
ssh  -o StrictHostKeyChecking=no attacker0.$EXP.frade "sudo pkill -9 tcpdump"
```

(a) Example bash script for our FRADE experiment

```
scenario:
  actor0 start_log
  actor1 start_log
  actor0 restart_server
  actor0 start_detector
  wait t0 actor1 legitimate
  wait t1 actor2 attack
  wait t2 actor2 stop_attack emit stop_attack_done
  when stop_attack_done wait t3 actor1 stop_legitimate emit stop_legitimate_done
  when stop_legitimate_done actor0 stop_python emit stop_python_done
  when stop_python_done actor0 ipset emit ipset_done
  when ipset_done actor0 stop_tcpdump emit stop_tcpdump_done
  when stop_tcpdump_done actor1 stop_tcpdump emit stop_tcpdump_done

bindings:
  stop_python sudo pkill -9 python
  stop_tcpdump sudo pkill -9 tcpdump
  ipset sudo ipset list blacklist > /zfs/FRADE/blacklist.$1.$2.$3.$ms
  attack cd ~/frade/traffic/flood_attacker/; sudo python3 attack.py -s $1 -n $2 -u ../urls/urls-$1.txt &
  start_detector cd ~/frade/experiments/run; sudo bash start_detector.sh \"$modules $m4\"
  stop_attack sudo pkill -9 python3
  restart_server cd ~/frade/experiments/run/; sudo bash restart_server.sh
  stop_legitimate sudo killall python3.4
  legitimate sudo python3.4 ~/frade/traffic/smart_attacker/legitimate.py -s $1
  --sessions 100 --logs /proj/FRADE/MTurk/Normalized-logs/$1-new.log 2>&1> output &
  start_log cd ~/frade/experiments/run/; sudo bash start_log.sh legitimate.$1.$2.$3.$ms
  stop_attack_done NOT EXIST_PROCESS(attack.py)
  stop_legitimate_done NOT EXIST_PROCESS(legitimate.py)
  stop_python_done NOT EXIST_PROCESS(python)
  ipset_done ipset list blacklist | wc = 4
  tcpdump_done NOT EXIST_PROCESS(tcpdump)
```

(b) FRADE experiment translated into DEW.

Figure 3: Our example FRADE experiment in bash and its translation into DEW. We have color coded actors orange, actions green, triggers blue and purple, and events red. Bindings are shown as black.

Part of the power of DEW is that one experiment representation can be used to generate many topology descriptions. Currently, starting from actors and constraints, we can automatically produce NS topologies for networking testbeds like Emulab [16] and Deterlab [8].

4.3 User Interface

Users have different preferences for the way they interact with tools, and we expect this is especially true for experiment design tools. We expect we will need a range of user interfaces (UI) to achieve wide adoption of DEW.

We have started with three input UIs for DEW which help guide users through designing an experiment and producing its DEW representation using: (1) text-based and (2) NLP-based input, and (3) DAG-based input.

Our **assisted text UI** is an augmented text editor which performs text prediction and suggests complete and partial statements a user can select to construct their DEW. It consists of four panes, as shown in Figure 4, with the left panes showing actors, behavior and constraints, and the right pane showing suggested elements to complete the current statement that the user is working on. The panes are synchronized in real time. As the user types in scenario statements, the UI code automatically mines actors, actions and events, and populates the actor pane and the suggestion panes. This way users, which are not very familiar with DEW syntax, can easily create scenarios from scratch. In the Figure, the user has typed two statements into the scenario pane, and is now being suggested the constructs they can use for the third statement.

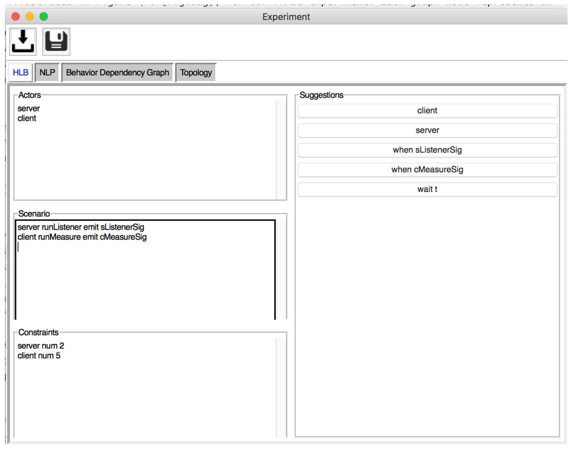
Our **Natural Language Processing (NLP) UI**, enables users to write free form English sentences. These are then processed to extract actors, actions and dependencies. For example, a user can type
After the server starts the listener and the measure script, the client will start its traffic.
and the UI would produce three lines of DEW:

```
server runListener emit sListenerSig
server runMeasure emit sMeasureSig
when sListenerSig, sMeasureSig client
     start_traffic emit sTrafficSig
```

We do not expect the NLP UI to be able to fully parse abstract descriptions, such as descriptions in the evaluation section of a technical paper, but we expect this UI can be helpful to users learning the DEW syntax.

Our **Directed Acyclic Graph (DAG)-based** method is under development and enables users to drag and drop graph nodes and connect these nodes with directed arcs in a drawing area. This is illustrated in Figure 5(a), for our original FRADE experiment. Each graph node represents an action and is annotated via color or text, with the actor that performs this action. In the figure, we use color-based indication of the actor. An arc between two nodes A and B, where $A \rightarrow B$, indicates that action A will emit an event, which will be used as a trigger for action B. If B should start after a time-based trigger, this is noted via a label on the arc between nodes. In situations where an action should start after some delay, but there is no event-based trigger, this node is connected to the special "start node" with an arc, which is labeled with the delay. Figure 5(a) illustrates this for action "legitimate".

As we discussed, converting our original FRADE experiment script into DEW helped us realize that its workflow did not properly capture dependencies between ac-



Actors:
server
client

Scenario:
server runListener emit sListenerSig
client runMeasure emit cMeasureSig

Constraints:
server num 2
client num 5

Figure 4: Screen shot of Assisted Text-Based UI

tions. This is also clearly visible from its DAG in Figure 5(a), where the legitimate action starts after an arbitrary time $t0$, and does not depend on the preparatory stages of setting up logging, server, and attack detection. We display the corrected DAG in Figure 5(b): the setup stages on the blue actor depend on each other, the start of the legitimate action depends on the completion of the setup stages, and the wrap-up stages occur in parallel.

Once the researcher is satisfied with their scenario and constraints, they click the "save" icon and our UI prompts them for the binding for each action and event. Scenario and constraint panes, along with the bindings, are saved as this experiment's DEW representation.

5 Benefits of DEW

Expressing experiments as DEW representations has multiple benefits for managing the overall experiment lifecycle, as we discuss below.

More robust design. Using DEW representation enables researchers to reason about experiments at a high level. This enables them to focus on action ordering and dependencies first, and then consider how these will be realized on the testbed, and specify the bindings and constraints. By thinking explicitly about ordering and dependencies, researchers can arrive at more robust scenarios, where actions wait for explicit triggers, and do not rely on timing and luck during execution.

Run histories. When the researcher runs the experiment on the testbed they would allocate a given topology, and then parameterize and run their scripts. These scripts are produced from a DEW or they are manually created by researchers, and passed through a translator to produce a DEW. Testbeds can build accompanying services to record run histories, and interpret their segments as sections of scenarios in DEW. For example, a run history for FRADE experiment may look like this: configure(wiki), run(wiki, 3), analyze(wiki).

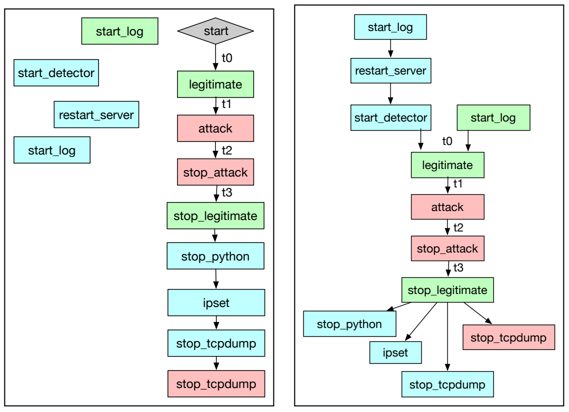
Experiment orchestration. Experiments are often orchestrated by researchers themselves, by invoking certain scripts on certain nodes, via SSH. Testbeds can use a scenario and bindings from DEW to build robust orchestration mechanisms. These mechanisms can detect when events occur and use them to trigger subsequent actions. Rather than build a custom orchestration, or select a single orchestration platform, we plan to build translators from DEW into inputs for multiple orchestrations platforms (e.g. MAGI [3] or Ansible [1]) and allow users to pick the output, which best suits their needs.

Easier failure detection. The scenario and bindings from DEW serve as input to a testbed, communicating what should happen in an experiment. This could be leveraged to detect failures during experiment runs in a timely manner. For example, an excessively long wait before a specific event could trigger a failure alert. We leave this for future work.

Hypothesis testing. Testbeds can build languages and services on top of DEW that help researchers express hypotheses. Testbeds can then orchestrate runs for hypothesis testing. For example, a researcher may request to execute the "run" stage of the scenario 100 times or until the average result of the "analyze" stage stabilizes. The testbed could then engage our generators to produce the appropriate scripts, and execute the run and analyze stages repeatedly, saving the results in separate files and evaluating stop conditions.

Flexible and explicit constraints. Expressing topology constraints explicitly enables negotiation between user and testbed, and possible constraint relaxation. We expect that users may under- or over-constrain their experiment, and so we can engage constraint solvers to provide real-time feedback to researchers as to how their constraints can be realized on the currently available testbed nodes. Using this feedback the researcher may decide to relax their constraints to increase their chances of experiment allocation success.

Sharing and reuse. Run histories would enable easy sharing of experiments. Testbeds could help bundle the experiment's DEW, resulting scripts and topologies, the executables used in bindings, as well as the record of the actual physical resource allocations, run history and any resulting data files, into a single archive for sharing. Researchers that attempt to reuse elements of this bundle could leverage DEW and run history to quickly understand how useful the bundle's elements are for their purpose. For example, the researcher may decide to reuse DEW and change the order of actions or add new ones. Or the researcher may decide to repeat the experiment on a new testbed, and change the constraints regarding a node feature, if the new testbed has too few nodes with a given feature (e.g., fast CPU).



(a) Original (b) Corrected

Figure 5: Our example FRADE experiment in DAG, both in its original and corrected form. The color of each node represents which actor role carries out each action: red for DDoS attacker(s), blue for the victim server and green for the legitimate client(s).

6 Conclusions and Future Work

Testbed experiments today are often ad-hoc, manual and hard to repeat and reuse. We have argued that this occurs mostly because testbeds fail to capture experiment *behavior* and leverage behavior to support the experiment lifecycle. We have proposed DEW—Distributed Experiment Workflow representation. DEW encodes experiment behavior and constraints, and uses them to automatically generate topologies and scripts. DEW enables researchers to create and manipulate their experiments at a high cognitive level, and enables creation of other services for experiment lifecycle management, such as failure detection, hypothesis testing, constraint negotiation, sharing and reuse.

We have much future work. First, we anticipate we can improve DEW's syntax and look forward to feedback as we move forward with testing DEW's expressiveness. We expect to greatly expand the expressiveness and handling of constraints to allow users to be detailed and explicit about their resource needs. Second, we plan to improve our bash-to-DEW translator and add new translators for other scripting languages. As we move forward, we will build generators to provide a scripting framework for users designing experiments directly with DEW. Last, we will continue improving our UIs and explore additional mechanisms to help users design experiments using DEW.

7 Acknowledgments

This project was sponsored in part by the Science and Technology Directorate of the United States Department of Homeland Security under contract number HSHQDC-16-C-00024. The views and conclusions contained herein are those of the authors and should not be interpreted necessarily representing the official policies or endorsements, either expressed or implied, of the Department of Homeland Security or the US Government.

References

[1] Ansible. `https://www.ansible.com`.

[2] DEW: Distributed Experiment Workflows Project. `https://github.com/gbartlet/DEW.git`.

[3] Montage AGent Infrastructure. `https://montage.deterlab.net`.

[4] Baxley, S., Bastin, N., and Gurkan, D. Analysis of In-order Packet Delivery Network Policy Enforcement Function. Second Place, GENI Experimenter Contest.

[5] Buchert, T., Nussbaum, L., and Gustedt, J. A Workflow-inspired, Modular and Robust Approach to Experiments in Distributed Systems. In *CCGRID - 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (May 2014).

[6] Cherrueau, R.-A., Simonin, M., and van Kempen, A. EnosStack: A LAMP-like stack for the experimenter. In *CNERT* (2018).

[7] Deelman, E., Blythe, J., Gil, Y., Kesselman, C., Koranda, S., Lazzarini, A., Mehta, G., Papa, M. A., and Vahi, K. Pegasus and the Pulsar Search: From Metadata to Execution on the Grid. In *Applications Grid Workshop, PPAM 2003* (2003).

[8] DETER. DeterLab Web page. `http://www.deterlab.net`.

[9] Eide, E., and Stoller, L. An Experimentation Workbench for Replayable Networking Research. In *Symposium on Networked Systems Design & Implementation (NSDI)* (2007).

[10] Freire, J., and Silva, C. T. Making Computations and Publications Reproducible with VisTrails. *Computing in Science Engineering 14*, 4 (July 2012), 18–25.

[11] GENI Project. GENI Desktop tutorial at GEC23, 2015.

[12] Mirkovic, J., Shi, H., and Hussain, A. Reducing Allocation Errors in Network Testbeds. In *Proceedings of the Internet Measurement Conference (IMC)* (2012).

[13] Seidel, S., Colbourn, C. J., Syrotiuk, V. R., Mehari, M., Poorter, E. D., and Moerman, I. An Intuitive Drag and Drop Framework for Wireless Network Experimentation. Demo at CNERT 2018.

[14] Vrijders, S., Staessens, D., Capitani, M., and Maffione, V. Rumba: a Python Framework for Automating Large-Scale Recursive Internet Experiments on GENI and FIRE+. In *CNERT* (2018).

[15] Wachs, M., Herold, N., Posselt, S., Dold, F., and Carle, G. GPLMT: A Lightweight Experimentation and Testbed Management Framework. In *PAM* (2016), vol. 9631 of *Lecture Notes in Computer Science*, Springer, pp. 165–176.

[16] White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the Operating System Design and Implementation* (2002), pp. 255–270.

[17] Zink, M., Bhat, D., Wang, C., and Rakotoarivelo, T. GIMI/LabWiki Tutorial, 2014.