The ISC Int'l Journal of Information Security
ISeCure

January 2025, Volume 17, Number 1 (pp. 59–73)
http://www.isecure-journal.org

CST-SDL: A Scenario Description Language for Collaborative Security Training in Cyber Ranges

Navid Shirmohammadi [1] and Behrouz Tork Ladani [1,*]
[1] *Department of Software Engineering, Faculty of Computer Engineering, University of Isfahan, Isfahan, Iran*

ARTICLE INFO.

Article history:
Received: June 7, 2024
Revised: November 12, 2024
Accepted: December 25, 2024
Published Online: December 31, 2024

Keywords:
Cyber Range, Cybersecurity, Training, Model-Driven Engineering, Scenario Description Language

Type: Research Article

doi: 10.22042/isecure.2024.462008.1132

ABSTRACT

As cyber threats grow increasingly sophisticated, the importance of security training as an effective means of prevention will become even more critical. Cyber Range (CR) is a platform for creating cyber training programs using virtualization and simulation technologies to create a realistic training environment. The main challenge for utilizing a CR is the specialized human resources required to design and maintain training sessions. To tackle this challenge, several high-level languages, known as Scenario Description Languages (SDLs), have been developed to enable the specification of training environments as models. These models can then be automatically transformed into deployment artifacts. Our studies showed that the existing SDLs could not address requirements when designing complex scenarios where multiple trainees should collaborate to reach a desired goal through various acceptable solutions. We present the Collaborative Security Training SDL (CST-SDL) for creating multi-trainee and multi-solution scenarios. CST-SDL uses an acyclic directional graph for specifying the scenario's solution routes and allows defining trainees with unique tasks, goals, and solution routes during the training session. To evaluate the CST-SDL's capabilities, we have implemented and integrated it into the KYPO cyber range.

1 Introduction

Cyber range (CR) is a platform that allows individuals to gain knowledge and hands-on experience about cyber threats in a low-cost and effective manner without endangering their data and IT assets [1–3]. CRs have been proven to be an effective solution for cyber security training and improving the security position of organizations [4–7]. Organizations can utilize CR to design training environments that correspond to their requirements and decrease the time employees need between learning new skills and utilizing them in the working environment [8].

One of the main factors determining the effectiveness of training is that it should correspond to the existing challenges of the industry [9–11]. Rapid changes in the technology stack and cyber threats can signify this factor in the field of cyber security even more than in any other context [12]. Designing new training sessions (scenario) is the obvious solution for keeping up the skills of the trainees relevant to new requirements. However, designing a new scenario is rather challenging and requires solid knowledge about the requirements of the intended industry sector, re-

* Corresponding author.
Email addresses: n.shirmohammadi@comp.ui.ac.ir, ladani@eng.ui.ac.ir
ISSN: 2008-2045

lated cyber threats, and skill in designing appropriate environments accordingly [9, 13, 14].

Model-driven engineering (MDE) can be used as an alternative for manually designing and deploying scenarios [11]. MDE suggests that by developing a high-level language (metamodel) for describing the requirements of the target environment, one can benefit from well-developed solutions for integrating verification and code generation requirements [15, 16]. Scenario description languages (SDL) are the outcome of implementing MDE in CRs. SDLs enable the scenario designer to describe their intended training environment and benefit from the automated verification and code generation processes implemented based on the SDL's structure [11, 13].

A few SDLs have been developed to address the different requirements of scenario designers. Existing SDLs mainly focus on providing utilities for describing the details of infrastructure used for training, specifying the software vulnerabilities and the linear path for solving the scenario [10, 11, 13, 17]. When analyzing existing SDLs, we identified the inability to describe complex trajectories for solving the scenario as the main limitation of these languages. In other words, existing SDLs were developed assuming that a single trainee (or a group of trainees with similar starting conditions) will take advantage of one of the vulnerabilities in the training environment to solve the scenario. This assumption differed when we tried to model complex scenarios with multiple trainees and acceptable solutions.

In this research, we present the CST-SDL, an SDL that focuses on describing trainees' collaboration in scenarios with multiple solutions. CST-SDL allows the scenario designer to describe the solutions as a DAG [1] in which the nodes have a set of pre- and postconditions based on the status of the infrastructure entities (like files, servers, and networks) and permissions. Scenario designers can use CST-SDL to specify the DAG's roots as the starting point of individual trainees and explicitly define the trainees' collaboration as a training group. We process this information to automatically verify the feasibility of the scenario by checking if all training groups can reach the desired nodes of DAG and solve the scenario. The main contributions of our SDL are:

- CST-SDL can be used to specify all the possible routes for solving the scenario using the DAG.
- CST-SDL can be used to design scenarios for red (offensive), blue (defensive), and purple (offensive and defensive) teams.
- CST-SDL allows the scenario designer to define a unique set of questions for each trainee (for gamification purposes) in the model and couple them with DAG nodes to specify their unique solution route.

The paper is organized as follows. Section 2 describes related works on SDL solutions and compares their capabilities with CST-SDL. We explain why a new SDL is needed in Section 3 by listing the shortcomings of existing solutions. Section 4 will present the CST-SDL and the feasibility verification algorithm, while in Section 5, we use the CST-SDL to create a scenario and demonstrate our SDL's capabilities compared to existing solutions. Finally, we conclude this paper in Section 6.

2 Related Work

Model-driven engineering (MDE) has been used in earlier cyber ranges to design domain-specific languages (DSL) for modeling arbitrary scenarios. The main reasons for using this MDE in the cyber range can be summarized as follows [7]:

- **Verifiability**: It is possible to automatically check the presence of intended features in the scenario model.
- **Expressiveness**: The designer can use the language (metamodel) to describe an arbitrary scenario.
- **Compositionality**: Existing scenarios can be combined to generate a new scenario.
- **Integration**: Arbitrary output (like deployment artifacts) can be generated by parsing the model.

Cheung *et al.* [18] made one of the early efforts to design a cybersecurity-related DSL by introducing the Correlated Attack Modeling Language (CAML). CAML tries to Identify the logical steps during an attack and specify their relationship. A cyber attack is a set of modules in which each has specific activation conditions alongside the pre- and postconditions. The observed events received from sensor reports are the source of activation conditions. Preconditions specify the constraints on the system and service configurations. When the activation and preconditions of a module are met, the postconditions of that module will be considered to be satisfied and used for future inferences. CAML is not an SDL, but its solution for modeling Attacks inspired CST-SDL's development.

One of the recent (and few) efforts to design a DSL was made by Russo *et al.* [9] and continued in [10], which offered the Cyber Range Automated Construction Kit (CRACK) SDL. CRACK SDL was developed with a focus on design, verification, code gener-

[1] directional acyclic graph

ation, and automatic scenario testing. This SDL can describe the infrastructure and vulnerable services running on the servers, user accounts and their privileges, and basic vulnerabilities on different services. CRACK SDL uses the pyDatalog [2] module to verify the scenario models by checking if the trainee can use their knowledge about the accounts credentials and vulnerable services to reach a goal. Automated deployment and testing are other notable features of CRACK that allow scenario designer to examine their designs on demand.

Costa *et al.* [11] presented the Virtual Scenario Description Language (VSDL) for "automating the definition and deployment of arbitrarily complex cyber range scenarios." Scenario designers can use VSDL to create a high-level model of their intended environment's infrastructure and make time-based modifications to the deployed scenario. A model generated using the VSDL contains expressions that determine the changes in hardware resources based on a time-based trigger, and the VSDL will apply these changes when the conditions are met.

Somarakis *et al.* [14] introduced a metamodel called the Security Assurance Model (SAM) for a cyber training program. This metamodel records a company's cyber assets, corresponding threads, vulnerabilities, and security controls. Also, three sub-models were introduced to cover the cyber range, simulation, and emulation information. The cyber range sub-model is the one we are concerned with, as it is used to describe information related to training programs. This sub-model can define the actors, the organization's software assets, and training program phases.

Braghin *et al.* [13] introduced the Cybersecurity Training Language (CTL). Scenario designers can use CTL to model an arbitrary scenario by defining the characteristics and relationships of IT assets and specifying the network devices, servers, software, and network connections. Code generation functions consume the CTL-generated models to produce OpenStack HEAT [3] templates for deployment. This SDL can be used to create red and blue team scenarios.

Yamin *et al.* [17] presented an SDL capable of modeling arbitrary scenarios by defining nodes, routers, services, vulnerabilities, teams, challenges, and agents. Besides the notable details in the metamodel, which allows the designer to model the different aspects of infrastructure and training process, the provided graphical user interface is a noticeable contribution made by the authors. This SDL can be used to create red and blue team scenarios.

Table 1. comparison of the SDLs' capabilities

SDL	Red Team Scenario	Blue Team Scenario	Purple Team Scenario	Multi-solution Design	Non-linear Solution Routes	Collaboration	Gamification	Testing
CRACK SDL [10]	*	*		*				*
VSDL [11]	*	*						
CTL [13]	*							
SAM [14]	*	*	*					
- [17]	*	*	*					
CST-SDL	*	*	*	*	*	*	*	

We compare the SDLs based on multiple criteria, such as their capability to describe red, blue, and purple team scenarios, define scenarios with multiple solutions, consider gamification requirements, and offer a solution for modeling trainees' collaboration. Table 1 shows the result of our comparison according to these criteria. Most of the reviewed SDLs can design red and blue team scenarios but are not suitable for designing purple team scenarios. Our proposed SDL and CRACK SDL are the only SDLs capable of describing multiple solutions in their model. However, CRACK SDL only allows the scenario designer to describe the goal and vulnerabilities, not the solution. CST-SDL can define non-linear (graph-based) solution paths, describe trainees' collaboration, and embed the gamification aspect of the scenario within the model. In CST-SDL, we did not implement the automated testing of deployment environments like CRACK SDL since there are reliable solutions, like Ansible [4] and Terraform [5], for configuring the environment based on the model criteria.

3 Motivation

Existing SDL solutions allow the scenario designers to specify their intended training environment based on the hardware specifications and software configurations. The designer can describe the network topology, select the software running on each server, manage files, and make changes at the run time if a trigger happens (see VSDL [11]). Other capabilities of existing SDLs are specifying the vulnerabilities and training goals, defining trainees, and creating primary training groups.

To demonstrate the need for a new SDL, let us consider the workflow of designing a scenario using existing SDLs. First, the scenario designer uses the SDL to define the infrastructure they intend to use during

[2] https://sites.google.com/site/pydatalog/home
[3] https://wiki.openstack.org/wiki/Heat
[4] https://www.ansible.com/
[5] https://www.terraform.io/

training. Depending on the SDL's capabilities, this includes the network topology, vulnerable software, and other related aspects. Depending on the SDL, designers can specify the training goal as infiltrating a particular account using a chain of vulnerabilities. Designers can define time-based triggers to modify the hardware depending on the training environment and their expectations of trainees' progress. After designing the scenario infrastructure, the designer has to write questions to guide the trainees through the scenario and provide them with the required knowledge about their targets and steps (gamification). Cyber ranges usually provide the designer with basic facilities for gamification. Finally, the designer should deploy and check their design to ensure the trainees have the information and resources to play the scenario (feasibility). It requires playing as each trainee and considering the collaboration of different training teams. In many training scenarios, designers can only go up to one trainee or a group of similar trainees as the time required for testing will grow.

The scenario design workflow requires a considerable amount of manual processing, interacting with cyber range after designing the infrastructure for deploying gamification and testing the feasibility of the scenario. Reusing the scenario or modifying it, in the best case, requires going through the test stage again, which can be time and resource-consuming. All these issues can be addressed by designing an SDL that integrates gamification into the models and allows the designer to specify the resources required for each step individual trainees should take during the session. If an SDL contains such information, the testing process can be automated by checking the resources of training groups, which reduces the testing time after designing or reusing the model to a few seconds. Such automation allows the designers to create complex scenarios with different routes for each trainee and to verify their design to ensure the availability of information and resources for every training group.

4 The Proposed Approach

This paper offers a new scenario description language (SDL), called CST-SDL, that aims to verify scenarios in which a group of trainees works together toward a goal. In this type of scenario, the initial resources of each trainee and the status of different infrastructure components can play a role in the achievable outcome of a trainee group. The following are our intentions for designing a new SDL:

- **Custom infrastructure**: SDL should be capable of describing arbitrary infrastructures.
- **Arbitrary scenarios**: SDL should be capable of describing Red, Blue, and Purple team scenarios.
- **Multiple solutions**: SDL should offer practical solutions for specifying all the approaches one can use to solve a scenario.
- **Non-linear solution**: SDL must not limit the progress definition to a single sequence of actions.
- **Collaboration**: It is desired to group the trainees to solve the scenario. SDL should be capable of describing trainees' collaboration.
- **Gamification**: SDL should offer facilities to define each trainee's unique questions and progress paths.

4.1 SDL's Anatomy

Our studies of existing SDLs and experiments while designing multiple security training sessions showed that an SDL should address four topics: infrastructure, instructions, collaborations, and gamification. CST-SDL was developed by absorbing existing SDLs' concepts and extending them by applying our findings. One can better understand the language structure and novelty by separately analyzing these topics in the CST-SDL. Figure 1 shows the left half of the metamodel diagram, while Figure 2 shows the right half.

- **Infrastructure:** SDL should offer facilities to describe arbitrary infrastructures. CST-SDL offers classes representing networks, routers, servers, accounts, software, services, and files in our SDL. Each class provides a unique set of properties for addressing the relevant information. An abstract class, called ITEntity, acts as the parent on which the preceding infrastructure classes are built and offers common properties, like initial permissions and status. Table A.1 (Appendix A) shows the definition of ITEntity and infrastructure classes in our SDL. all the classes end with Options inherit from ITEntity and offer related properties of the entity in their name. The NetworkMapping, Route, and RouteMapping classes use basic data structures within the NetworkOptions and RouterOptions classes. The ITEntityPermission and ITEntityStatus are data structures to hold information about resource permissions and status when required.
- **Instructions:** In complex training scenarios, multiple valid approaches might exist for reaching a desired outcome. CST-SDL offers facilities for describing different methods for solving the scenario. We considered modeling the solutions for solving a scenario as a directional acyclic graph (DAG) in which every node has a set of pre- and postconditions. This choice lets the designer describe complicated solution patterns

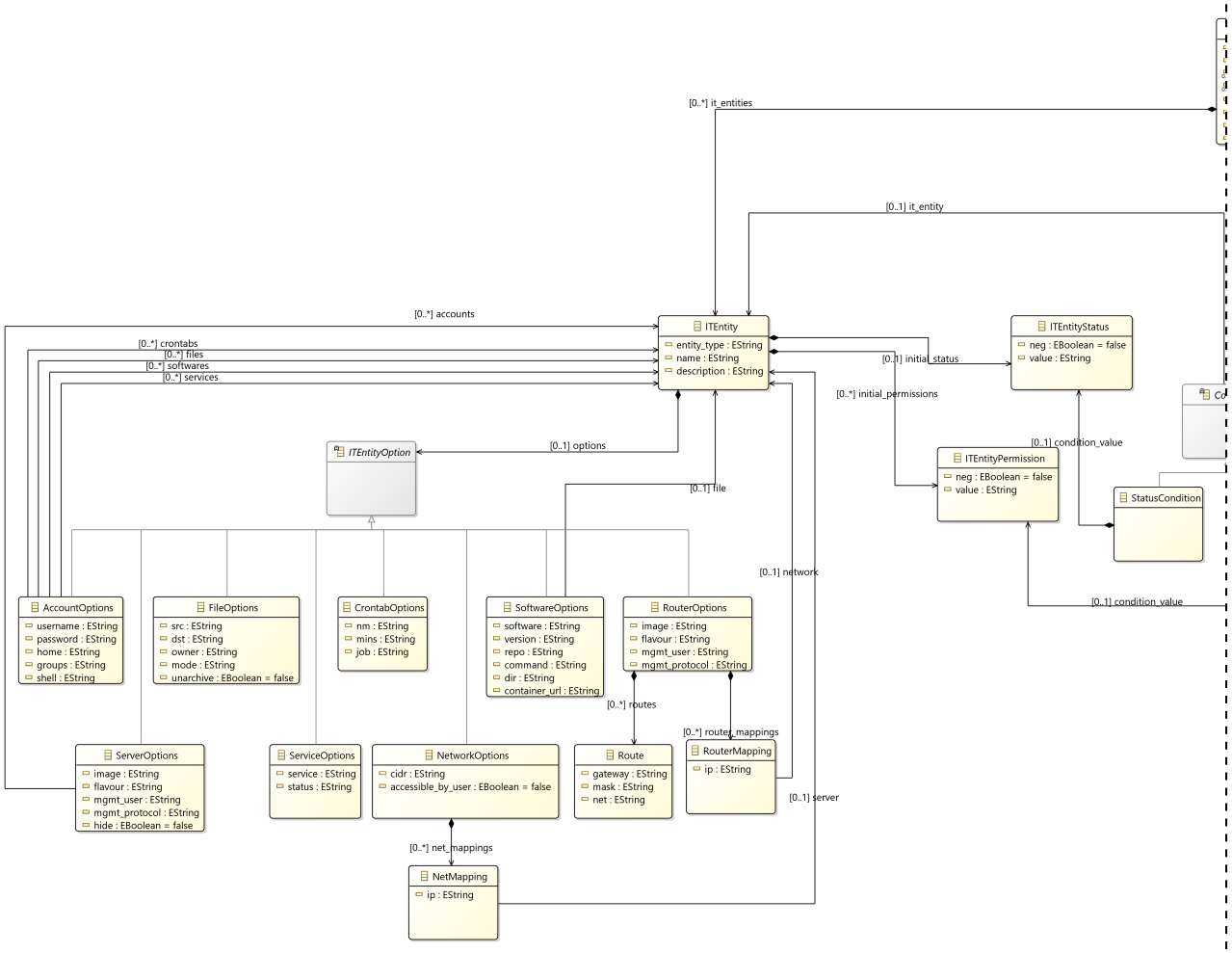

Figure 1. CST-SDL's metamodel diagram (left half)

with minor duplication. In CST-SDL, precondition describes a set of permissions, tags, or configurations a trainee should gain or cause before entering a node. The postcondition holds the same data structure as the precondition and specifies the new permissions, tags, and configurations that a trainee might gain or cause before leaving the node. Table A.2 (Appendix A) shows the State class, representing the DAG node. State class uses the Condition class for its pre- and postconditions.

- **Collaborations:** As mentioned earlier, our SDL aims to verify scenarios where trainees work together toward a goal automatically. Each trainee might initially have unique resources and privileges against the scenario infrastructure. CST-SDL allows the designer to specify the trainees' route independently, using the mapping property and assumes the first state's preconditions in the trainee's mapping as his/her initial privileges and resources. Table A.3 (Appendix A) shows the Trainee and TrainingGroup classes, which are used to specify the trainees and their collaboration as a group. The Path class is used as a data structure to match the trainee's path with levels defined in gamification.
- **Gamification:** CST-SDL allows embedding the gamification requirements within the model. A question-based training session is a common practice for gamification in many cyber ranges, such as KYPO CR, and we used this method as the gamification solution. Table A.4 (Appendix A) shows the classes used for gamification in our SDL. We modified the KYPO CR's Training Definition by replacing some platform-specific properties with general-purpose alternatives. We also added a callback property to our Level class, allowing the scenario designer to make a GET request to an API endpoint when trainees reach a node in DAG, making integrating custom solutions with CST-SDL easier. We use Level class to provide trainees with information and questions. The Question and Choice

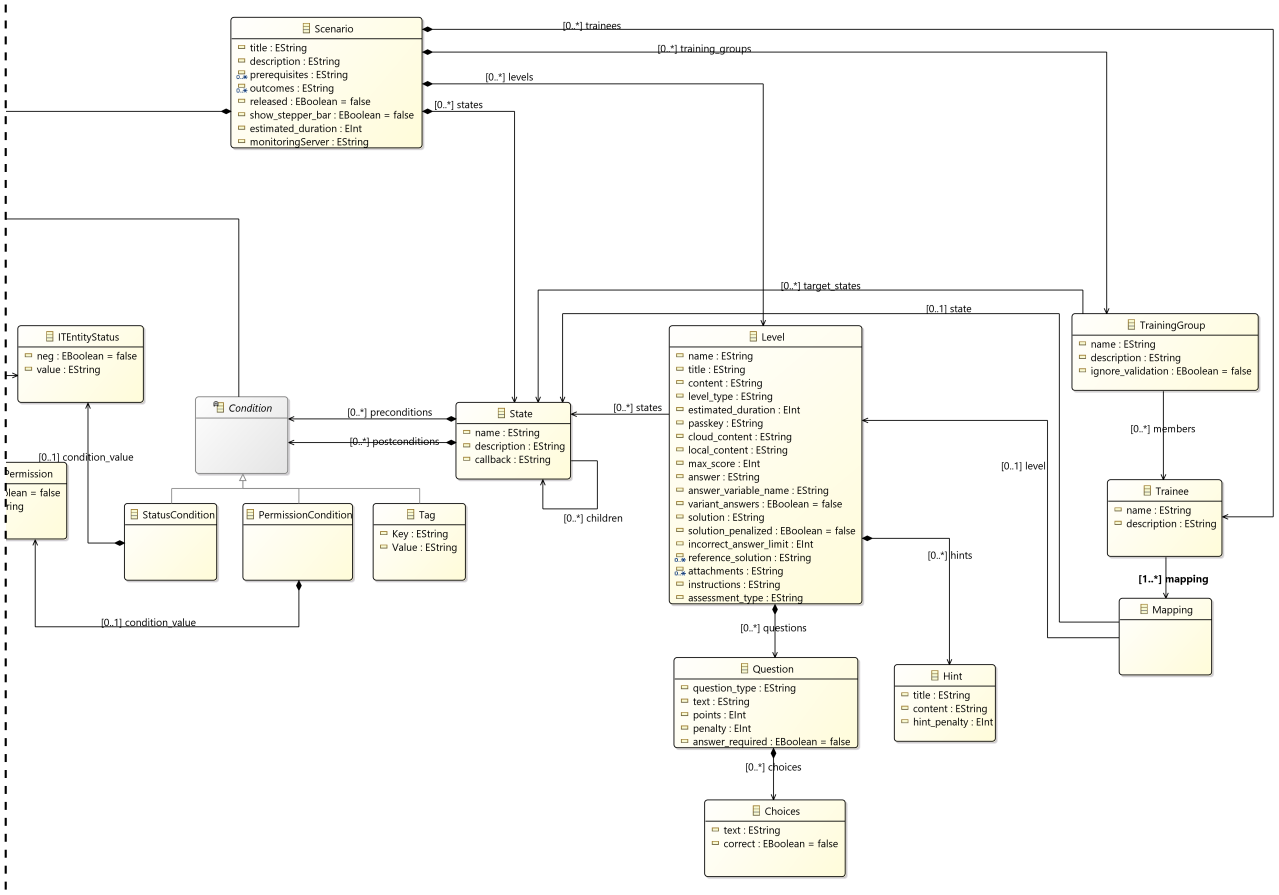

Figure 2. CST-SDL's metamodel diagram (right half)

classes allow the designer to design multiple-choice questions if needed, while the Hint class provides trainees with basic aids during their exercise.

4.2 SDL's Verification

The scenario needs to be verified to ensure it is compatible with the constraints enforced by the deployment tools and the requirements of scenario designers. In CST-SDL, we check the model before generating deployment artifacts, mainly to prevent errors related to the tools we use for deployment (Ansible and Terraforms). We also verify if the training teams can reach the desired nodes of DAG based on their members' resources (feasibility). We implemented our SDL using the Xtext [6], a framework for development domain-specific languages, which allows us to write our verification logic using Java and Xtend [7] while offering designers with a real-time verification as they are making their model. We implemented multiple constraints for different CST-SDL classes based on their usage. Some of these constraints are as follows:

- server's IP should match the network.
- roots of DAG must have valid preconditions (if provided) based on the infrastructure instances' status.
- file paths and repositories should exist.
- time allocated for answering levels must be a positive number.
- there should be no cycle in DAG.

Verifying the feasibility checks whether a group of trainees can reach desired nodes of DAG by checking their initial resources and assuming complete collaboration between them. This assumption means the trainees will share their resources and knowledge during the scenario with teammates. Each trainee begins at a specified initial node, and the algorithm checks whether they can proceed through the graph by meeting the preconditions and postconditions of the connected nodes.

The core idea is that the trainees will attempt to extend their routes by moving through the DAG's nodes based on the conditions of the scenario. The verification concludes when:

- The group collectively reaches all mandatory nodes (target states).

[6] https://www.eclipse.org/Xtext/
[7] https://www.eclipse.org/xtend/

- At least one trainee completes their route by reaching a terminal state (a node with no further outgoing edges).

If both conditions are satisfied, the training group can solve the scenario. Two algorithms determine this: Check Scenario Feasibility (CSF) and *ExtendSolutionGraph* (ESG), which are shown in Algorithm 1 and Algorithm 2 respectively.

4.2.1 The CSF Algorithm

The CSF algorithm is responsible for determining whether the scenario is solvable by any of the training groups.

(1) Initialize Infrastructure:
- Line 4: The initial status of the infrastructure is stored in the `infDesc` variable, which represents the state of all the resources available to the trainees. The state will be updated as they progress through the graph.

(2) Identify Terminal States:
- Line 5: The terminal states (endpoints in the DAG) are fetched and stored in `terminalStates`. These are nodes where a trainee's route can end successfully.

(3) Group Processing:
- Line 6: The algorithm iterates through each `trainingGroup` in `TrainingGroups`.
- Line 7: For each group, `buds` (a list of nodes to be explored) is initialized as an empty set.

(4) Trainee Initialization:
- Lines 8-10: For each trainee in the training group, their starting node (`initial_state`) is added to the `buds` set, which represents the initial positions of all trainees.

(5) Solution Graph Setup:
- Line 11: An empty `solutionGraph` is initialized. This graph will store the possible states each trainee can reach.
- Lines 12-18: For each trainee, the algorithm constructs an initial node in the solution graph based on their `initial_state`. The infrastructure description (`infDesc`) is updated to reflect the trainee's initial position and the set of `buds` is updated to include the trainee's possible subsequent nodes (children of the current node).

(6) Graph Extension Loop:
- Lines 19-20: A while loop is initiated to expand the solution graph using the ESG algorithm iteratively. The loop continues until no further expansion is possible, at which point the scenario is determined to be either feasible or infeasible.
- Line 21: Calls the ESG function to extend the solution graph.

Algorithm 1 Check Scenario Feasibility (CSF)

```
1:  function CSF(ITEntities, SSG, TrainingGroups)
2:      infDesc ← initiateInfrastructureDescriptioin(
3:      ITEntities)
4:      terminalStates ← getTreminlaStates(SSG)
5:      for trainingGroup ∈ TrainingGroups do
6:          buds ← {}
7:          for trainee ∈ trainingGroup.members do
8:              buds ← buds ∪ {trainee.initial_state}
9:          end for
10:         solutionGraph ← {}
11:         for trainee ∈ trainingGroup.members do
12:             solutionGraph ← solutionGraph ∪ {Node(
13:                 state ← trainee.initial_state
14:                 infDesc ← updateInfrastructureDesc-
15:                 ription(infDesc, trainee.initial_state)
16:                 hasTerminal ← False
17:                 buds ← buds ∪ {trainee.initial_state.
18:                 children}/{trainee.initial_state}
19:                 targets ← trainingGroup.target_states
20:                 /{trainee.initial_state}
21:                 parent ← {}
22:             )}
23:         end for
24:         while True do
25:             solutionGraph ← ESG(solutionGraph,
26:             terminalStates)
27:         end while
28:     end for
29: end function
```

4.2.2 The ESG Algorithm

The ESG algorithm extends the solution graph by checking if trainees can move to new nodes based on the current infrastructure and node preconditions.

(1) Initialization:
- Line 2: A boolean `grown` is set to `False`. This flag indicates if the solution graph has been successfully extended.
- Line 3: `newNodes` is initialized as an empty set, holding any new nodes added to the solution graph.

(2) Processing Each Node:
- Line 4: The algorithm iterates through each node in the current `solutionGraph`.
- Line 5: For each node, an empty set `removed_buds` is initialized to track which buds (candidate nodes) will be removed after processing.

(3) Precondition Check:
- Lines 6: For each bud (candidate node to explore), the algorithm checks if the preconditions of the bud are met, given

the current infrastructure description (`infDesc`).

- Lines 7-9: If the preconditions are satisfied, the solution graph is considered to have grown (`grown = True`), meaning the graph can be extended.
- Lines 10-22: A new node is created with the following details:
  - Line 13: State: The bud becomes a new node in the graph.
  - Lines 14-15: Infrastructure Update: The infrastructure description is updated to reflect the trainee's move to the new node.
  - Lines 16-17: Terminal Check: If the new node is a terminal state, the `hasTerminal` flag is set to `True`.
  - Lines 18-19: Buds Update: The set of buds is updated to include the children of the new node.
  - Line 20: Target States Update: The `targets` set is updated to remove the newly visited node.
  - Line 21: Parent Update: The `parent` is set to node.

(4) Completion Check:

- Lines 24-31: If all target states are satisfied (i.e., `targets` is empty) and the trainee has reached a terminal state, the scenario is considered feasible (`EXIT::FEASIBLE`). The algorithm exits at this point, indicating success.

(5) Graph Growth Check:

- Lines 35-37: If the graph did not grow during this iteration (`grown = False`), the scenario is deemed infeasible, and the algorithm exits with a negative result (`EXIT::INFEASIBLE`).
- Line 38: If the graph grows, the new nodes are added to the solution graph, and the loop continues.

4.2.3 Complexity Analysis

In this section, we delve into the time and space complexity of the CSE (Algorithm 1) and ESG (Algorithm 2). These algorithms are pivotal in constructing and expanding solution graphs based on specific criteria. Understanding their computational complexity is not just a technical exercise but a crucial step in assessing the performance and scalability of the overall system.

ESG Algorithm: The Extend Solution Graph (ESG) algorithm (Algorithm 2) operates by iterating over all nodes and their corresponding buds to expand the graph when certain preconditions are met. This iterative process is a crucial aspect of the algorithm's

Algorithm 2 Extend Solution Graph (ESG)

```
1:  function ESG(solutionGraph, terminalStates)
2:      grown ← False
3:      newNodes ← {}
4:      for node ∈ solutionGraph do
5:          removed_buds ← {}
6:          for bud ∈ node.buds do
7:              cond = checkPreconditions(bud, node.
8:              infDesc)
9:              if cond then
10:                 grown ← True
11:                 removed_buds ← removed_buds ∪ bud
12:                 newNodes ← newNodes ∪ {Node(
13:                   state ← bud
14:                   infDesc ← updateInfrastruct-
15:                   ureDescription(node.infDesc, bud)
16:                   hasTerminal ← node.hasTerminal OR
17:                   terminalStates.contain(bud)
18:                   buds ← node.buds ∪ {bud.children}
19:                   /{bud}
20:                   targets ← node.targets/{bud}
21:                   parent ← node
22:                 )}
23:             end if
24:             if node.targets/{bud} is empty then
25:                 cond2 = node.hasTerminal
26:                 cond3 = terminalStates.contain(bud)
27:                 if cond2 OR cond3 then
28:                     solutionGraph ← solutionGraph∪
29:                     newNodes
30:                     return EXIT :: FEASIBLE
31:                 end if
32:             end if
33:         end for
34:     end for
35:     if grown is False then
36:         return EXIT :: INFEASIBLE
37:     end if
38:     solutionGraph ← solutionGraph ∪ newNodes
39:     return solutionGraph
40: end function
```

operation. Let N represent the number of nodes in the solution graph and B represent the maximum number of buds per node.

Time Complexity: The algorithm consists of two nested loops. The outer loop runs over the nodes in the solution graph, while the inner loop processes each bud. Therefore, the time complexity of the ESG algorithm is:

$$O(N \cdot B)$$

This analysis indicates that the algorithm's performance scales linearly with the number of nodes and buds associated with each node, providing a clear understanding of its scalability.

Space Complexity: The space complexity is determined by the storage of the buds and new nodes generated during the expansion process. Since the algorithm processes each bud and stores the intermediate results, the space complexity is:

$$O(B)$$

CSF Algorithm: The Compute Solution Framework (CSF) algorithm (Algorithm page 65) constructs a solution graph for each training group and then calls the ESG algorithm iteratively to expand it. Let G be the number of training groups, M be the maximum number of members in a training group, and C be the maximum number of buds associated with a trainee's initial state.

Time Complexity: The outer loop iterates over the training groups (G), and for each group, it processes the members to collect initial states and build a solution graph. The construction of the solution graph for each member involves processing the buds of their initial state, which takes $O(M \cdot C)$. Once the solution graph is built, the algorithm enters a while loop, calling the ESG algorithm until a specific condition is met. The overall time complexity can be expressed as:

$$O(G \cdot M \cdot C + G \cdot M \cdot B)$$

This complexity accounts for processing the training groups and their members, constructing the solution graph, and expanding it through iterative calls to the ESG algorithm.

Space Complexity: The space complexity of the CSF algorithm is driven by the storage of the solution graph and the buds for each trainee. The memory required for storing the infrastructure description and terminal states is also considered, leading to the following space complexity:

$$O(M + N)$$

5 Evaluation

This section presents a basic training scenario consisting of two trainees with different initial resources collaborating to steal information from a target web service. For this scenario, we use the following story: "Alice (trainee1) is working for WebObjects, a fictional startup that provides an object storage web service. WebObjects currently allows users to store and retrieve binary files using their web application. There are two types of users on WebObjects: normal users who can only see their files' URLs and inspector users who can see all of the files' URLs. Since WebObjects was developed without appropriate security considerations, anybody can download any file with the correct URL. Still, if a user tries to access a non-existing URL, their IP will be banned for ten minutes. Alice asks Bob (trainee2), and they make a plan to deceive the cyber security team of WebObjects and steal the URLs. Alice will activate the Test account, an old account used in the early development stage of WebObjects for internal testing, which has inspector access and an unknown but weak password. Alice then pretends she is testing new features of WebObjects for a while and unintentionally (very intentionally) forgets to deactivate the Test account. Bob will brute force the Test account's password and download the URL file from the web application. They will sell the data to a mysterious customer looking for information."

The scenario can be designed in four phases: designing the infrastructure, specifying the questions for guiding users through the scenario, specifying the progress route per trainee, and grouping trainees into training groups. With CST-SDL, we can perform all four phases. Figure 3 shows the required infrastructure for this scenario. We use two separate networks for internal and public traffic from WebObjects' perspective. There are three servers: WWW is the web server hosting the WebObjects web application and database, PC1 is an internal computer for employees (Alice), and PC2 is used by the client (Bob) to access the web application. PC1 only needs a web browser to access the WebObjects web application. PC2 requires a web browser and Burp Suit [8] to perform the attack. The WWW server needs to run multiple containers for storage (MinIO [9] and PostgreSQL [10]) and a Django [11] web application.

Listing 1 (Appendix B) shows the infrastructure definition using CST-SDL. In lines 1-9, we defined the PC1 as a server entity and specified the operating system image, hardware flavor (defined in OpenStack [12]), management user, and existing accounts (which is Alice's account). We also set the initial status as active, which indicates that the server instance should be running as soon as the scenario begins. The definition of PC2 and WWW are similar to PC1, except that they have Bob and WWW accounts, respectively, and Bob uses Kali Linux [13].

In lines 17-24, we specified Alice's account (Alice_ACC) by specifying the username, password, and groups. The initial state is set to PRESENT, which will be used when generating the deployment code for Ansible to create the account. Bob's account is similar to Alice, so its definition is omitted in Listing 1 (Appendix B). The www account is defined in lines 29-43, and besides the basic account information, it shows the definition account's files and software. In lines 45-52, we defined the fixtures.json file, which contains the initial database data and should be copied to the www account. Installation of MinIO,

[8] https://portswigger.net/burp
[9] https://min.io/
[10] https://www.postgresql.org/
[11] https://www.djangoproject.com/
[12] https://www.openstack.org/
[13] https://www.kali.org/

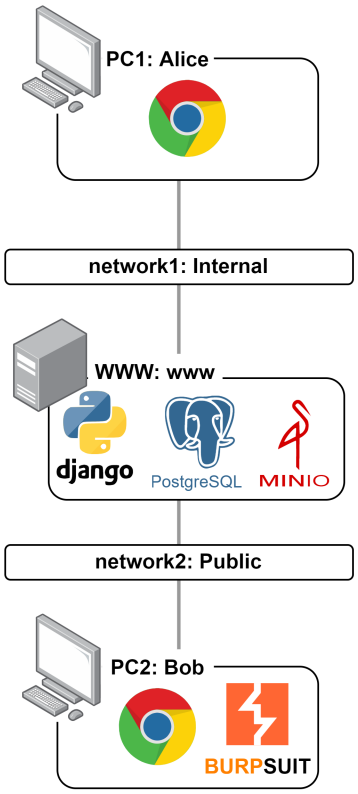

Figure 3. Motivation example's infrastructure

PostgreSQL, and the web application are specified in lines 45-66 (the latter two omitted as they are similar to the first one) by using Docker [14] images (indicated by "container: yes" in line 56) and the corresponding deployment command (line 59). Network topology is described in lines 68-73 and 76-82 for the internal and public networks, respectively.

The second phase of defining the scenario is to specify the questions for guiding the trainees through the scenario. CST-SDL uses a similar model to KYPO cyber range's [19] gamification model, with a few modifications (discussed in more detail in Section 4). In CST-SDL, the level class can be used for providing information and questions to the trainees during the scenario. Listing 2 (Appendix B) shows the usage of level class to define an information level (lines 1-4) and a training level (lines 6-18). the information level provides the trainees with the game's story, which trainees can ignore. In contrast, the training level asks the trainees to answer a given question correctly and has a score, hints, and penalty.

The third phase of designing the scenario is to specify the progress route of each trainee. In this basic scenario, trainees can play as either Alice or Bob. Listing 3 (Appendix B) shows the definition of two routes for Alice and Bob using CST-SDL. In Alice's Route, the trainee should log in to PC1, log in to her account on the WebObjects' website, and activate the Test account. Bob's Route consists of logging in to the PC2, brute-forcing the password with Burp Suit, logging in to the Test account on the WebObjects website, and fetching the data.

Listing 3 (Appendix B) defines these steps using the CST-SDL's state class. Each state can have a set of children (which indicate the following steps in the DAG), preconditions, and postconditions. In step_1 (lines 1-6), we specified a description for more readability, a child step representing the following step when the trainee finishes the current step, and a postcondition. Postcondition in state_1 indicated that after this step, Alice (trainee) would have access to Alice's account (Alice_ACC). step_2 has the same structure as step_1 with some preconditions added. Bob's route is defined using the state_4 through state_7 and uses the preconditions and postconditions to show the progressive changes in Bob's resources and privileges during his path.

Scenario designers can use pre- and postconditions to address the state of infrastructure instances (like line 14), user permissions over resources (like line 6), or define custom tags when there is no appropriate infrastructure instance (like line 16). They can decide to define a trainee who can play multiple users (if needed) in the scenario by adding appropriate levels and states. Another important note is that the routes can share nodes, indicating that multiple approaches exist for reaching a specific state during the scenario.

In the fourth and final phase of scenario definition, we define the trainees and group them into training groups to show collaboration during training. Listing 4 shows the definition of Alice (t_1) and Bob (t_2) using the CST-SDL. we can specify a trainee's resources at the beginning of the scenario (lines 2 and 11) and which levels a trainee should go through (lines 5-8 and 14-18). CST-SDL allows the designer to specify the corresponding state in DAG when a trainee answers a level using the mapping property of the Trainee class (lines 4 and 13). In lines 20-24, we defined a training group to show the collaboration between Alice and Bob.

CST-SDL tries to be a general-purpose SDL for designing arbitrary scenarios by addressing some of the shortcomings of existing SDL solutions. We could not find any standard metrics or method for comparing the SDLs except a high-level comparison between the new features of SDL with the existing solutions (as provided in Table 1). CST-SDL offers a more capable modeling solution for addressing training collaboration, multi-solution scenarios, and gamification. We

[14] https://www.docker.com/

modeled other scenarios based on SEED project web security labs [15], such as SQL-Injection and XSS attacks. We deployed the generated artifacts on the KYPO CR to ensure our SDL's competence.

6 Conclusion and Future Work

This paper presented a scenario description language (SDL), CST-SDL, for addressing the requirements of creating training environments in cyber ranges. CST-SDL enables the designers to describe the training infrastructure and explicitly specify trainees' collaboration, address gamification per trainee, and model complex solution routes in their model. By doing so, we could perform more complex validation algorithms to determine the feasibility of the scenario based on the collaboration of different trainees and reduce the ambiguity about the gamification and actual progress paths per trainee. We addressed some of the limitations of CST-SDL, which we will try to handle in future work. Our SDL can describe the infrastructure entities, specify steps for solving the challenges, handle trainees' collaboration, and embed gamification requirements. It also provides the scenario designer with verification functionalities to check the properties and feasibility of their design. We plan to extend the capabilities of our SDL to detect patterns in the trainees' interactions with the infrastructure by adding image recognition services and including new algorithms for detecting deadlock situations where none of the competitive training groups can progress due to resources other groups have taken.

Acknowledgment

This research was done at the University of Isfahan CERT Center (UI-CERT). The authors acknowledge financial support from the Iranian National CERT Center (Maher) and the Information Technology Organization of Iran (ITO). We would also like to express our gratitude to Dr. Behrouz Shahgholi and other UI-CERT colleagues for sharing their pearls of wisdom with us during this research.

References

[1] Cuong Pham, Dat Tang, Ken-ichi Chinen, and Razvan Beuran. Cyris: A cyber range instantiation system for facilitating security training. In *Proceedings of the 7th Symposium on Information and Communication Technology*, pages 251–258, 2016.

[2] Mika Karjalainen and Tero Kokkonen. Comprehensive cyber arena; the next generation cyber range. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 11–16. IEEE, 2020.

[3] Bernard Chng, Bennet Ng, Muhammad M Roomi, Daisuke Mashima, and Xin Lou. Craas: Cloud-based smart grid cyber range for scalable cybersecurity experiments and training. 2024.

[4] Elochukwu Ukwandu, Mohamed Amine Ben Farah, Hanan Hindy, David Brosset, Dimitris Kavallieros, Robert Atkinson, Christos Tachtatzis, Miroslav Bures, Ivan Andonovic, and Xavier Bellekens. A review of cyber-ranges and test-beds: Current and future trends. *Sensors*, 20(24):7148, 2020.

[5] Michail Smyrlis, Konstantinos Fysarakis, George Spanoudakis, and George Hatzivasilis. Cyber range training programme specification through cyber threat and training preparation models. In *Model-driven Simulation and Training Environments for Cybersecurity: Second International Workshop, MSTEC 2020, Guildford, UK, September 14–18, 2020, Revised Selected Papers*, pages 22–37. Springer, 2020.

[6] Magdalena Glas, Manfred Vielberth, and Guenther Pernul. Train as you fight: evaluating authentic cybersecurity training in cyber ranges. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, pages 1–19, 2023.

[7] Magdalena Glas, Gerhard Messmann, and Günther Pernul. Complex yet attainable? an interdisciplinary approach to designing better cyber range exercises. *Computers & Security*, 144:103965, 2024.

[8] Muhammad Mudassar Yamin and Basel Katt. Modeling and executing cyber security exercise scenarios in cyber ranges. *Computers & Security*, 116:102635, 2022.

[9] Enrico Russo, Gabriele Costa, and Alessandro Armando. Scenario design and validation for next generation cyber ranges. In *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, pages 1–4. IEEE, 2018.

[10] Enrico Russo, Gabriele Costa, and Alessandro Armando. Building next generation cyber ranges with crack. *Computers & Security*, 95:101837, 2020.

[11] Gabriele Costa, Enrico Russo, and Alessandro Armando. Automating the generation of cyber range virtual scenarios with vsdl. *arXiv preprint arXiv:2001.06681*, 2020.

[12] Hussain Aldawood and Geoffrey Skinner. Challenges of implementing training and awareness programs targeting cyber security social engineering. In *2019 cybersecurity and cyberforensics conference (ccc)*, pages 111–117. IEEE, 2019.

[13] Chiara Braghin, Stelvio Cimato, Ernesto Dami-

[15] https://seedsecuritylabs.org/Labs_20.04/Web/

ani, Fulvio Frati, Lara Mauri, and Elvinia Riccobene. A model driven approach for cyber security scenarios deployment. In *Computer Security: ESORICS 2019 International Workshops, IOSec, MSTEC, and FINSEC, Luxembourg City, Luxembourg, September 26–27, 2019, Revised Selected Papers 2*, pages 107–122. Springer, 2020.

[14] Iason Somarakis, Michail Smyrlis, Konstantinos Fysarakis, and George Spanoudakis. Model-driven cyber range training: a cyber security assurance perspective. In *Computer Security: ESORICS 2019 International Workshops, IOSec, MSTEC, and FINSEC, Luxembourg City, Luxembourg, September 26–27, 2019, Revised Selected Papers 2*, pages 172–184. Springer, 2020.

[15] Stuart Kent. Model driven engineering. In *Integrated Formal Methods: Third International Conference, IFM 2002 Turku, Finland, May 15–18, 2002 Proceedings*, pages 286–298. Springer, 2002.

[16] Douglas C Schmidt et al. Model-driven engineering. *Computer-IEEE Computer Society-*, 39(2):25, 2006.

[17] Muhammad Mudassar Yamin, Basel Katt, and Mariusz Nowostawski. Serious games as a tool to model attack and defense scenarios for cyber-security exercises. *Computers & Security*, 110:102450, 2021.

[18] Steven Cheung, Ulf Lindqvist, and Martin W Fong. Modeling multistep cyber attacks for scenario recognition. In *Proceedings DARPA Information Survivability Conference And Exposition*, volume 1, pages 284–292. IEEE, 2003.

[19] Jan Vykopal, Radek Ošlejšek, Pavel Čeleda, Martin Vizvary, and Daniel Tovarňák. Kypo cyber range: Design and use cases. 2017.

Navid Shirmohammadi received his M.Sc. in Secure Computing in 2023 in Information Security at the University of Isfahan. His work is focused on Automated Cyber Range Solutions, emphasizing investigating techniques for simplifying the Generation of Training Environments and Simulating Cyber Attacks.

Behrouz Tork Ladani Behrouz Tork Ladani received his B.Sc. in Software Engineering from the University of Isfahan, Iran, in 1996 and his M.Sc. in Software Engineering from the Amir-Kabir University of Technology, Tehran, Iran, in 1998. He received his Ph.D. in Computer Engineering from Tarbiat-Modarres University, Tehran, Iran, in 2005. He is currently a full professor at the Department of Software Engineering at the University of Isfahan. Dr. Ladani is a member of the Iranian Society of Cryptology (ISC). He is also the Managing Editor of the Journal of Computing and Security (JCS) and a member of the editorial board of the International Journal of Information Security Science (IJISS). Dr. Ladani's research interests include Security Modeling and Analysis, Software Security, and Soft Security.

Appendices

A Details of CST-SDL Classes

In this appendix, we provide the detailed properties and attributes of the classes used in the CTS-SDL

Table A.1. CST-SDL classes used in infrastructure definition

Class	Property	Description
ITEntity	name	unique ID
	entity_type	presents the type of ITEntity, e.g., server
	description	optional description
	initial_permissions	list of ITEntityPermission
	initial_status	list of ITEntityStatus
	options	type-specific options
RouterOptions	image	router operating system
	flavour	hardware configuration of router
	mgmt_user	router administrator
	mgmt_protocol	protocol used to access router
	routers	list of routers using the configuration
	router_mappings	list of router mappings
RouterMapping	ip	router's IP address
	network	the network connected to router interface
RouterMapping	ip	router's IP address
	network	the network connected to router interface
Router	gateway	router's gateway
	mask	router's network mask used by administrator
	net	router's network address used by administrator
NetworkOptions	cidr	network address and mask
	accessible_by_user	a boolean value that determines if trainees can access the network
	net_mappings	list of NetMapping
NetMapping	ip	IP address for a server
	server	name of the server getting the IP address
CrontabOptions	nm	name identifier of crontab entry
	mins	job execution interval in minutes
	job	the job that crontab should run
FileOptions	src	source path of the file
	dst	destination path of the file
	owner	owner of the file
	mode	access mode of the file
	owner	owner of the file
	unarchive	indicates if the file should be uncompressed after copy

Table A.1. Continue

ServiceOptions	service	name of the service
	status	status of the service
ServerOptions	image	operating system image name
	flavor	hardware configuration file name
	mgmt_user	management username for deployment
	mgmt_protocol	management protocol for deployment
	accounts	list of account on accounts on the server
AccountOptions	username	account's username
	password	account's password
	home	account's home directory
	groups	account's groups
	shell	account's shell program
	services	list of services on an account
	files	list of files on an account
	softwares	list of software on an account
SoftwareOptions	software_name	software's name
	version	software's version
	container	a boolean value that determines if a docker container should be deployed
	repository	software's repository
	command	commands that should run to execute the software
	files	files that should be copied for executing the software
ITEntityPermission	negative	a boolean value that negates the permission value
	value	permission value, e.g., Read and Write
ITEntityStatus	negative	a boolean value that negates the state value
	value	status value, e.g., Active and Exist

Table A.2. CST-SDL classes used in DAG definition

Class	Property	Description
State	name	unique ID
	description	optional description
	callback	an API endpoint to be called using GET request when the state reached
	children	list of child states in DAG
	preconditions	list of Conditions
	postconditions	list of Conditions
Tag	key	Tag's key
	value	Tag's value
StatusCondition	condition_value	value of a ITEntityStatus
PermissionCondition	condition_value	value of a ITEntityPermission

‘

B Motivation Example Codes

This appendix contains the implementation of Section 5's example using the CST-SDL [16] .

[16] The file referenced as fixtures˙json in line 48 is a standard JSON file containing the necessary data to initialize the database using Django's fixture mechanism.

Table A.3. CST-SDL classes used for defining collaboration

Class	Property	Description
Trainee	name	unique ID
	description	optional description
	initial_state	starting state
	mapping	list of Path
TrainingGroup	name	unique ID
	description	optional description
	target_states	list of States that training group should enter
	members	list of Trainee
Mapping	level	Mapping's level
	state	Mapping's state

Table A.4. CST-SDL classes used for gamification

Class	Property	Description
Level	name	unique ID
	title	level's title
	content	level's content
	type	level's type, e.g., info, training or exam
	estimated_duration	estimated duration to answer level
	max_score	level's maximum score
	answer	level's answer (optional)
	solution	level's solution
	incorrect_answere_limit	maximum number of incorrect answers before revealing solution
	hints	list of Hint
	questions	list of Question
Question	type	question's type, e.g., multi-choice and text
	content	question's content
	points	question's points (used for levels with multiple questions)
	penalty	wrong answer penalty
	required	a boolean value that indicated if answering the question is required
	choices	list of Choice
Choice	content	choice's content
	correct	a boolean value that indicates if the choice is the correct answer
Hint	title	hint's title
	content	hint's content
	penalty	hint's penalty

Listing 1: *Defining server in CST-SDL*

```
1  entity server PC1:
2   description: Internal PC
3   options:
4     image: ubuntu-focal-x86_64
5     flavor: standard.small
6     mgmt_user: ubuntu
7     accounts:
8       - Alice_ACC
9     initial_state ACTIVE
10
11 entity server PC2:
12  <omitted: similar to 1-9>
13
14 entity server WWW:
15  <ommited: similar to 1-9>
16
```

```
entity account Alice_ACC:
 options:
   username: Alice
   password: 4lic3
   groups:
     - Alice
     - sudo
 initial_state PRESENT

entity account Bob_ACC:
 <omitted: similar to 17-24>

entity account WWW_ACC:
 options:
   username: www
   password: secure9821
   groups:
     - docker
     - www
     - sudo
   files:
     - fixtures_json
   softwares:
     - minio
     - postgresql
     - webapp
 initial_state PRESENT

entity file fixture_file:
 description: database data
 options:
   source: fixtures.json
   destination: /home/www/fixtures.json
   owner: www
   mode: 660
 initial_state PRESENT

entity software minio:
 options:
   container: yes
   software_name: https://nexus.local
       :8090/images/minio
   version: latest
   commands: docker run -p 9000:9000 -p
       9090:9090 --name minio -v ~/minio/
       data:/data -e "MINIO_ROOT_USER=
       minio_root" -e "MINIO_ROOT_PASSWORD
       =fhd08as" minio server /data --
       console-address ":9090" -d
 initial_state PRESENT

entity software postgresql:
 <omitted: similar to 54-60>

entity software webapp:
 <omitted: similar to 54-60>

entity network internal_network:
 options:
   cidr: 10.10.10.0/24
   net_mappings:
     - 10.10.10.10: PC1
     - 10.10.10.20: WWW
 initial_state ACTIVE

entity network public_network:
 options:
   cidr: 10.10.20.0/24
   net_mappings:
     - 10.10.20.10: PC2
     - 10.10.20.20: WWW
 initial_state ACTIVE
```

Listing 2: *Defining Questions in CST-SDL*

```
level level_0:
 title: Story
 content: Alice is working for ...<omitted
     >
 type: info

level level_1:
 title: Alice mission: Step 1
 content: It is time to begin ...<omitted>
 type: training
 estimated_duration: 2
 max_score: 10
 answer: flag{R3ady4Bob}
 solution: First Logint to ...<omitted>
 solution_penalty: 10
 hints:
   - title: Activatation
     content: Activate the Test account
     penalty: 5

...<omited>
```

Listing 3: *Defining DAG in CST-SDL*

```
state state_1:
 description: PC1-Alice Login
 children:
   - state_2
 postconditions:
   - Alice_ACC: ACCESS

state state_2:
 description: WebObjects-Alice Login
 children:
  - state_3
 preconditions:
   - Alice_ACC: ACCESS
   - webapp: PRESENT
 postconditions:
   - tag: WebObjects-Alice-Login

state state_3:
 description: WebObjects=Test Activate
 preconditions:
   - tag: WebObjects-Alice-Login
 postconditions:
   - tag: WebObjects-Test-Active

state state_4:
 description: PC2-Bob Login
 children:
   - state_5
 postconditions:
   - Bob_ACC: ACCESS

state state_5:
 description: Use PC2-Bob's BurpSuit
     repeater and brute force
 children:
```

```
    - state_6
  preconditions:
    - tag: WebObjects-Test-Active
    - webapp: PRESENT
  postconditions:
    - tag: WebObjects-Test-Access

state state_6:
  description: WebObjects-Test Login
  children:
   - state_7
  preconditions:
    - tag: WebObjects-Test-Access
    - webapp: PRESENT
  postconditions:
    - tag: WebObjects-Test-fetch

state state_7:
  description: WebObjects-Test fetches data
  preconditions:
    - tag: WebObjects-Test-Access
    - tag: WebObjects-Test-fetch
    - webapp: PRESENT
  postconditions:
    - tag: fetch-done
```

Listing 4: *Defining training groups in CST-SDL*

```
trainee t_1:
 description: Trainee 1-Alice
 initial_state: state_1
 mapping:
   - level_0:
   - level_1: state_1
   - level_2: state_2
   - level_3: state_3

trainee t_2:
 description trainee 2-Bob
 initial_state state_4
 mapping:
   - level_0:
   - level_4: state_4
   - level_5: state_5
   - level_6: state_6
   - level_7: state_7

group training_group_1:
 description: Team 1
 members:
   - t_1
   - t_2
```