Template Patterns for Improving Configurability and Scalability of Enterprise Distributed Real-time and Embedded System Testing and Experimentation

James H. Hill, Aniruddha Gokhale, and Douglas C. Schmidt

Vanderbilt University, Nashville, TN, USA
{hillj,gokhale,schmidt}@dre.vanderbilt.edu
http://www.dre.vanderbilt.edu

Abstract. Testing and experimentation (T&E) is the process of executing many tests of a system using different configurations and scenarios. T&E is particularly important for enterprise distributed real-time and embedded (DRE) systems since its enables evaluation of quality-of-service (QoS) attributes, such as performance and reliability, throughout the software lifecycle. The heterogeneity of software and hardware, however, makes T&E of enterprise DRE systems hard, *e.g.,* developers often use handcrafted T&E solutions that evaluate just a few of many different host, threading model, and workload configurations. Moreover, existing T&E techniques (such as handcrafted shell scripts) are tedious, time consuming, and labor intensive when trying to scale to address T&E concerns of enterprise DRE systems.

This paper presents four related patterns for improving T&E scalability and flexibility by deferring realization of T&E configurations until as late as possible, *e.g*, based on the operating environment. We provide a synopsis, evaluation, and application for each pattern in the context of a representative enterprise DRE system. Our experience applying these patterns shows they can significantly improve the scope of T&E and evaluation of enterprise DRE system QoS.

1 Introduction

Enterprise distributed real-time and embedded (DRE) systems, *e.g.*, shipboard computing environments, mission avionic systems, and air traffic control, possess characteristics that complicate verification and validation [12], such as complexity, heterogeneity, and scale. Testing and experimentation (T&E), *i.e.*, running many tests of the system under different configurations to exercise validation of enterprise DRE systems, is the conventional method for evaluating system behavior and quality-of-service (QoS). T&E of enterprise DRE systems in realistic environments and operating conditions also helps increase confidence that the system being developed meets its functional and QoS requirements [5, 11]. For example, enterprise DRE system developers can leverage dynamically configurable testbeds, such as Emulab [18], to produce realistic environments for understanding and evaluating system QoS throughout the software lifecycle.

The T&E process, however, can be expensive and time consuming [11], even when it is automated. Moreover, the effectiveness of T&E relates directly to the ability to test many different system configurations (such as number of hosts, threading model,

and number of clients) in realistic and hypothetical operating conditions. For example, understanding and evaluating the worst-case response time of critical execution paths in a DRE system requires testing a DRE system under different workloads.

Existing techniques that address T&E configuration concerns include domain-specific modeling languages [15], which can help alleviate the complexity of handcrafting configuration files via models and model interpreters. These techniques, however, rely largely on *single instance* configurations, where one configuration is used to evaluate a system under a single operating condition. To evaluate the system under different operating conditions, enterprise DRE system developers must produce multiple variants of the same configuration file (or model), which is tedious, time-consuming, and error-prone. Developers therefore need improved techniques and patterns to improve T&E configurability and scalabilty to broaden their evaluation scope.

Solution approach → Template patterns. A *template* is an abstraction that captures the fixed and variable portions of a context, such as an algorithm [3, 7], model transformation [14, 20], and configuration file [9, 21]. For example, the Template Method [8] design pattern enables developers to capture the fixed portion of an algorithm while deferring certain variable steps to subclasses. *Template patterns* describe techniques for transforming the variable portion of a given template, such as setting a variable in a configuration file's template based on it hostname. In general, templates help increase the configurability and scalability of their application context, including offline situations where timing constraints are not an issue.

This paper describes the following related template patterns that help increase T&E configurability and scalability: *Variable Configuration*, *Batch Variable Configuration*, *Dynamic Variable Configuration*, and *Batch Dynamic Variable Configuration*. We have implemented variants of each pattern in the *Component Workload Emulator (CoWorkEr) Utilization Test Suite (CUTS) Template Engine* (CUTE). CUTE uses a domain-specific language to capture fixed and variable portions of T&E configurations, such as resolving the correct network interface based on the operating enviroment of a test. Enterprise DRE system developers can use CUTE to decrease the amount of *single instance* configuration files for T&E, while increasing T&E configurablity and scope. This paper shows how the template patterns supported by CUTE help improve enterprise DRE systems evaluation capabilities, such as testing hypothetical workloads or migrating to between different operating environments.

Paper Organization. The remainder of this paper is organized as follows: Section 2 introduces a case study of a representative enterprise DRE system project to motivate the need for CUTE; Section 3 describes the four template patterns that CUTE implements to improve T&E configurability and scalabilty; Section 4 provides quantitatively analyzes these template patterns in the context of the case study; Section 5 explores the relationship between CUTE and related work on improving T&E configurability and scalability; and Section 6 presents concluding remarks.

2 Case Study: The QED Project

The QoS-Enabled Dissemination (QED) [1] project is a large-scale, multi-team collaborative project involving Vanderbilt, Boeing, BBN Technologies, and IHMC aimed at

addressing QoS concerns within the Global Information Grid (GIG) [2]. In particular, this project provides reliable and real-time communication middleware to application-level components and end-user scenarios that operate within dynamically changing conditions and environments. Figure 1 shows the relationship of components in QED and the GIG. As shown in this figure, QED is a layer between the application-level compo-

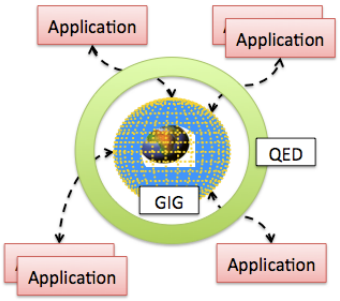

Fig. 1. Conceptual Model of QED in the Context of the GIG

nents and communication protocols for the GIG. At the heart of the QED middleware is a Java-based information broker that tailors and prioritizes information based on end-user scenario needs and importance.

The goal of the QED project is to improve QoS concerns of the GIG, which involves evaluating many points-of-variability, such as scalablity, operating environments, and workload. To evaluate that GIG QoS concerns are addressed adequately, QED developers plan to test many different candidate technologies and implementations, including Mockets [22] and differentiated service queues [4, 13], under various deployment scenarios. Conventional T&E techniques, such as handcrafted and hardcoded scripts or conventional domain-specific modeling language, require QED testers to expend much time and effort conducting T&E on the QED middleware. For example, if QED testers used domain-specific modeling languages to auto-generate configuration files, they would have to create a model for each single instance configuration. More specifically, QED testers are faced with the following challenges:

– **Varying T&E scenario configurations with minimal effort.** Each T&E scenario in the QED project is expensive in both time and effort to create. Forcing QED testers to (re)implement each scenario for each different configuration is also an expensive—yet tedious and error prone—task to undertake. QED testers, therefore, need techniques that will enable them to define a T&E scenario once and vary its configuration without incurring the effort required to realize the original T&E scenario.
– **Supporting multiple testers and operational environments for single T&E scenario.** Each stackholder in QED, such as BBN Technologies, Boeing, IHMC, and Vanderbilt, have testers responsible for evaluating their respective development features in QED against the GIG middleware under different scenarios. Moreover, all

tests are conducted in a dynamic testing environment named ISISlab [1]. Forcing QED testers to manage duplicate versions of a single T&E scenario to account for the dynamics and heterogeneity of each individual tester and the target testing environment is an expensive process. QED testers therefore need better techniques to reduce such complexity when defining T&E scenarios.

The remainder of this paper discusses four T&E template patterns we identified, implemented, and analyzed to address the challenges of QED testers and simplify T&E efforts, while increasing evaluation capabilities of QED.

3 Template Patterns for Testing and Experimentation

This section discusses four template patterns for T&E that increase configurability and scalability. As shown in Figure 2, each pattern builds upon the other, and are the building blocks of CUTE. The remainder of this paper provides a detailed synopsis (*i.e.*, problem statement and solution), evaluation, and application of each pattern in the context of CUTE and the QED project case study described in Section 2.

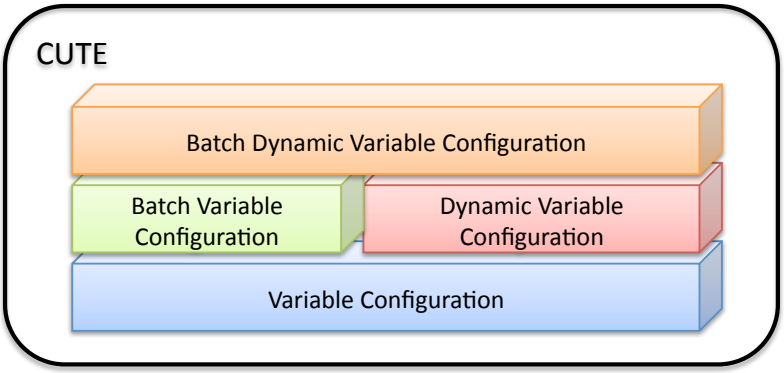

Fig. 2. Template patterns that are building blocks for CUTE.

3.1 Variable Configuration Pattern

Problem statement. Traditional techniques for T&E rely on configuration files to determine the test scenario for the enterprise DRE system under developement. The benefit of configuration files is that they decouple the implementation from configuration and behavior, *i.e.*, enable late-binding. For example, component-based middleware [16, 17]

[1] ISISlab (`www.isislab.vanderbilt.edu`) is a software integration testbed powered by Emulab software that allows developers to configure network topologies and operating systems on-the-fly to produce a realistic operating environment for distributed system testing.

uses verbose XML files that determine how components intercommunicate, what properties to set for the underlying middleware, and what values to set for attributes of a component. System developers therefore implement the static portion of the enterprise DRE system in terms of the variable portion, and let the variable portion (which realizes the decoupling) be controlled by external configuration files.

In T&E, a single instance configuration file represents a single instance of a test run. To evaluate an enterprise DRE system, such as QED, under many different scenarios (*i.e.*, configurations), system developers must manually produce many different configurations. In many cases, however, the size of the static portion of the configuration has greater cost (*e.g.*, time to generate, or number of characters) than the variable portion of the configuration. Even in cases when the cost of the static portion is less than the cost of the variable portion, manually producing multiple variants is both time-consuming and error prone.

Solution. The *Variable Configuration* pattern allows enterprise DRE system developers to define the static portion of a configuration file, such as required header information in an XML document, while using variables (or placeholders) to capture the variable portion of the configuration. We formally define the Variable Configuration pattern $C = (S, V)$ as:

- A set S of characters that capture the static portion of the configuration C.
- A set V of variables that define the variable portion of the configuration C.

A *configuration instance* is a single configuration used to execute a test scenario. It is derived from a template configuration C by replacing the variable portions V of the configuration with concrete values. Equation 1 defines the equation used to evaluate a template configuration C.

$$C' = eval(C, D) \tag{1}$$

where C' is a single instance configuration of C and D is a set of tuples (K, v) determined by system testers such that $K \in V$ and $v = value(K)$ for configuration C'—which is analogous to a dictionary.

Manifestation in CUTE and QED. We have realized the Variable Configuration pattern in CUTE and applied to the QED project. QED testers utilize the Variable Configuration pattern using the following steps:

1. Define a text-based file that captures a single configuration.
2. Replace static portions of the single configuration with a template variable. This represents a point-of-variablity in the template configuration.[2]
3. Define a dictionary file that consists of all key-value pairs for each variable in the Variable Configuration. Developers also have the option of overriding keys in the dictionary file at the command-line when invoking CUTE.

Using the user-defined template and dictionary, CUTE applies Equation 1 to produce a single instance configuration. QED testers then use the derived configuration to evaluate the system under development.

[2] It is possible to combine this step and the previous step by auto-generating a template with its variables already defined.

```
1  ...
2    <configProperty>
3      <name>cpuTime</name>
4      <value>
5        <type><kind>tk_long</kind></type>
6        <value><long>${cpuTime}</long></value>
7      </value>
8    </configProperty>
9    <configProperty>
10     <name>testOwner</name>
11     <value>
12       <type><kind>tk_string</kind></type>
13       <value><string>${testOwner}</string></value>
14     </value>
15   </configProperty>
16 ...
```

Listing 1. Example of the deployment and configuration that uses the Variable Configuration pattern

Listing 1 shows an excerpt from the deployment and configuration (D&C) file of a test scenario from the QED project. Listing 1 contains two variables named `cpuTime` (line 6) and `testOwner` (line 13), which are points-of-variability. Likewise, Listing 2 is an example dictionary for the template configuration in Listing 1, and Listing 3 shows the concrete instantiation of the template configuration using the dictionary in Listing 2.

```
1  cpuTime=33.4
2  testOwner=hillj
```

Listing 2. Dictionary for the D&C template.

```
1  ...
2    <configProperty>
3      <name>cpuTime</name>
4      <value>
5        <type><kind>tk_long</kind></type>
6        <value><long>33.4</long></value>
7      </value>
8    </configProperty>
9    <configProperty>
10     <name>testOwner</name>
11     <value>
12       <type><kind>tk_string</kind></type>
13       <value><string>hillj</string></value>
14     </value>
15   </configProperty>
16 ...
```

Listing 3. Evaluation of the Variable Configuration pattern for the D&C.

Without the Variable Configuration pattern of the configuration, QED testers would have to define each single instance configuration manually. Moreover, CUTE helped

increase the configurability and flexibility of T&E configuration by deferring realization of a single instance configuration until as late as possible, *i.e.*, not solely at test design time.

Pattern evaluation. The following is a list of benefits for using the Variable Configuration pattern:

- It reduces the number of single instance configurations that must be handcrafted since system testers have to produce a template only once and use a dictionary to define each configuration instance.
- It reduces the amount of error that can be incurred from manually replicating each configuration and modifying a small portion of it. In the case of tools that can auto-generate configuration files, *e.g.*, domain-specific modeling languages, it reduces the amount of effort required to produce each configuration, such as manually modeling each configuration via a cumbersome and tedious process. This also in turn helps increase the scalability of T&E.
- It highlights the points-of-variability in a configuration. This makes is easier for enterprise DRE system developers to know what are the control parameters in a test scenario (or configuration)—similar to control parameters in an experiment. This also helps increase the scope of T&E.

Although the Variable Configuration pattern has many benefits, it also has several consequences. The following is a list of consequences for using the Variable Configuration pattern:

- Developers must define the value of each variable in V, or the configuration is considered to be *invalid* since all variables do not expand to a concrete value. This can become problematic in situations where portions of the Variable Configuration can be ignored, such as doing multiple passes n a single template using different configuration dictionaries and each configuration dictionary contains a disjoint subset of the variables in the template configuration. Developers must take this scenario into account to ensure each variable is properly expanded before using the single instance configuration.
- The *usability* of a configuration is unknown until it is applied to the system under development in a test scenario. This can be overcome, however, if the mechanism for generating the dictionary understands the contraints of the target context, similar to constraints in domain-specific modeling languages.

3.2 Batch Variable Configuration Pattern

Problem statement. In Section 3.1, we discussed the Variable Configuration pattern and how it enables developers to generate single instance configuration files using a template and dictionary. This technique is acceptable when testing only a few different configurations since the number of dictionaries required to realize each single instance configuration is minimal. For example, if a developer wanted to validate a single configuration by executing a test run of the configuration, then the Variable Configuration pattern is a satisfactory technique.

When trying to test many different configurations derived from a single template, however, it becomes problematic trying to manage many different dictionary files—where each instantiates a single instance configuration. Moreover, it becomes hard to logically understand how each dictionary relates without using *ad hoc* techniques, such as creating test suites where each configuration is representative of each test in the test suite. For example, a QED tester who is testing performance and security logically maintains each dictionary for evaluating the respective concern in a directory that has the appropriate name, such as `./performacnce` or `./security`. Although this is acceptable it negatively impacts T&E maintainability and scalability.

Solution. The *Batch Variable Configuration* pattern builds upon the Variable Configuration pattern and enables enterprise DRE system developers to logically group common configurations (*i.e.*, dictionaries) that are evaluated at once. Developers leverage the Batch Variable Configuration pattern by defining logically related dictionaries, such that all the dictionaries used to generate configurations for evaluating performance, in a single monolithic configuration.

The Batch Variable Configuration has the same formal definition as the Variable Configuration (see Section 3.1) since it is using a single template. It, however, has a different evaluation function as illustrated by Equation 2:

$$C'' = batcheval(C, D') = \{\forall d \in D' : eval(C, D)\} \tag{2}$$

where D' is the set of dictionaries such that if $d \in D'$ then d is an instance of D (see Section 3.1) and C'' is the set of configurations such that $C' = eval(C, d)$ and $C' \in C''$.

Manifestation in CUTE and QED. We have realized the Batch Variable Configuration pattern in CUTE and applied it to the QED project. QED testers leverage the Batch Variable Configuration pattern using the following steps:

1. Define a text-based template configuration file that contains variables, similar to the process in the Variable Configuration pattern (see Section 3.1).
2. Define a set of dictionaries in a single file where each individual dictionary defines the key-value pair for each variable in the template configuration file.

Using the user-defined template file and batch configuration file, CUTE applies Equation 2 to produce a set of configuration files. QED testers then use the derived configuration files to evaluate system QoS.

```
1  config (lowCPU.cdp) {
2    cpuTime=33.4
3    testOwner=hillj
4  }
5
6  config (highCPU.cdp) {
7    cpuTime=87.8
8    testOwner=hillj
9  }
```

Listing 4. Batch dictionary for the D&C template.

Listing 4 illustrates a set of dictionaries for the template configuration in Listing 1. As highlighted in Listing 4, there are two different—yet related—configurations for testing CPU workload named: `lowCPU.cdp` (line 1) and `highCPU.cdp` (line 6). Likewise, Listing 5 shows the concrete instantiation of the D&C excerpt for the batch configurations in Listing 4.

```
1  // lowCPU.cdp
2  ...
3    <configProperty>
4      <name>cpuTime</name>
5      <value>
6        <type><kind>tk_long</kind></type>
7        <value><long>33.4</long></value>
8      </value>
9    </configProperty>
10   <configProperty>
11     <name>testOwner</name>
12     <value>
13       <type><kind>tk_string</kind></type>
14       <value><string>hillj</string></value>
15     </value>
16   </configProperty>
17 ...
18
19 // highCPU.cdp
20 ...
21   <configProperty>
22     <name>cpuTime</name>
23     <value>
24       <type><kind>tk_long</kind></type>
25       <value><long>87.8</long></value>
26     </value>
27   </configProperty>
28   <configProperty>
29     <name>testOwner</name>
30     <value>
31       <type><kind>tk_string</kind></type>
32       <value><string>hillj</string></value>
33     </value>
34   </configProperty>
35 ...
```

Listing 5. Evaluation of the Batch Variable Configuration pattern for the D&C.

Without the Batch Variable Configuraton pattern, QED testers would have to a hard time logically maintaining and producing multiple configuration files used to evaluate system QoS.

Pattern evaluation. In addition to the benefits for using the Variable Configuration pattern, the following is a list of benefits of using the Batch Variable Configuration pattern:

– It reduces the amount of single configurations that must be managed by enterprise DRE system developers. Developers can just define all *valid* configurations in a single file and batch process it.
– It helps logically group different configurations that can be processed at once. For example, using the Batch Variable Configuration pattern, developers can group configurations by ownership (*i.e.*, who created the configuration) or by QoS concern each individual configuration tests, such as performance, reliability, and security.

Although the Batch Variable Configuration patterns has many benefits, it also has several consequences. In addition to the consequences of the Variable Configuration pattern, the following is a list of consequences for using the Batch Variable Configuration pattern:

– The Batch Variable Configuration is *valid* if and only if each individual configuration is valid. It is, however, possible to suppress this consequence through *partial validity* where it is acceptable to have invalid configurations in the batch configuration, or ensuring each individual configuration is valid before including it in the batch configuration.
– The Batch Variable Configuration can be a *point-of-failure* because multiple configurations are defined in a single file. If the batch file is lost, then testers must redefine the configuration file. This can be overcome, however, by chaining a batch configuration using external configuration—similar to the C++ include preprocessor definition.

3.3 Dynamic Variable Configuration Pattern

Problem statement. In Section 3.1 and Section 3.2, we discussed two template patterns for improving both configurability and scalability of T&E. Both patterns, however, do not fully take into account T&E concerns, such as operating environment. For example, `testOwner` in Listing 2 and Listing 4 is hardcoded to `hillj`. There can be situations where the value of `testOwner` needs to be determined by the username of the person evaluating the configuration for accountability purposes.

Requiring enterprise DRE system developers to maintain many different configuration files based on some side-effect of the operating environment—even in the case of the Batch Variable Configuration pattern—can negatively affect adaptability and configurability. For example, if developers were using dynamic testing environments, such as ISISlab, and want to evaluate performance of the same enterprise DRE system under different experiments, then it would require developers to maintain different (yet similar) configurations for each experiment since the operating environment is *logically* different, *e.g.*, different hostnames and IP addresses.

Solution. The *Dynamic Variable Configuration* pattern builds upon the Variable Configuration pattern, and enables developers to capture the variable portion of a configuration that depends on a context-based side-effect, such as the hostname of a node in an experiment or the output of an equation evaluator. We formally define the Dynamic Variable Configuration pattern $DC = (C, \Delta)$ as:

– A template configuration C that contains the static and variable portions of the configuration file (see Section 3.1).

– A set Δ of context-based side-effects (or dynamic variables) that capture dynamic portions of configuration DC. We assume that $\delta \in \Delta$ produces a single line of text that can be used in place of its respective dynamic variable in C.

We derive a single instance configuration of DC by first evaluating C using Equation 1 in Section 3.1. Next, we evaluate the result C' using Equation 3:

$$DC' = eval(C', \Delta) \tag{3}$$

where DC' is a single instance configuration for the Dynamic Variable Configuration pattern that enterprise DRE system developers use for T&E.

Manifestation in CUTE and QED. We have realized the Dynamic Variable Configuration pattern in CUTE and applied it to the QED project. QED testers leverage the Dynamic Variable Configuration pattern using the following steps:

1. Define a text-based template configuration file that contains variables similar to the process in the Variable Configuration pattern (see Section 3.1).
2. Replace portions of the configuration with dynamic variables that will perform a side-effect to determine the value for that particular portion of the document.
3. Define a dictionary file that consists of all key-value pairs for each variable in the configuration. Developers also have the option of overriding keys in the dictionary file at the command-line when invoking CUTE.

Using the user-defined template, dictionary, and side-effects, CUTE uses Equation 1 then Equation 3 to evaluate the configuration. QED testers then use the derived single instance configuration to evaluate system QoS.

```
1  ...
2    <configProperty>
3      <name>cpuTime</name>
4      <value>
5        <type><kind>tk_long</kind></type>
6        <value><long>${cpuTime}</long></value>
7      </value>
8    </configProperty>
9    <configProperty>
10     <name>testOwner</name>
11     <value>
12       <type><kind>tk_string</kind></type>
13       <value><string>${userName}</string></value>
14     </value>
15   </configProperty>
16 ...
```

Listing 6. Example of the D&C that uses the Dynamic Variable Configuration pattern

Listing 6 illustrates an example configuration that uses the Dynamic Variable Configuration pattern, which is a variant of the configuration from Listing 1. As highlighted on line 13, we have replaced `hillj` with a variable named `userName`. Likewise, Listing 7 lists an example dictionary for Listing 6, which produces the same single instance

configuration in Listing 3[3]. In this case, `userName` is defined as a dynamic variable where `userName` is determined by the output of the command `/usr/bin/whoami`.

```
1 cpuTime=33.4
2 userName=`/usr/bin/whoami`
```

Listing 7. Example dictionary for the Dynamic Variable Configuration pattern.

Without the Dynamic Variable Configuration pattern, QED testers would have to either use *ad hoc* techniques of maintaining many different single instance configuration files for different operating contexts, such as different developers executing tests or running experiments on different hosts. CUTE, therefore, improves the adaptability, configurability, and scalability of T&E.

Pattern evaluation. In addition to the benefits of using the Variable Configuration pattern, the following is a list of benefits for using the Dynamic Variable Configuration pattern:

- It enables configurations to adapt to their operating environment. Developers do not have to have multiple configurations for each context, such as different hosts or users.
- It greatly increases the agility and configurability of a configuration for T&E.
- Developers can implement user-defined side-effects that understand the execution environment, such as auto-generating a UUID or binding to the appropriate network interface, and can be applied to the configuration. This helps increase the configuration flexibility for T&E.

Although the Dynamic Variable Configuration pattern has many benefits, it has several consequences. In addition to the consequences from the Variable Configuration pattern, the Dynamic Variable Configuration pattern has the following consequences:

- The configuration is *valid* if and only if C is valid and all side-effects in Δ execute successfully. This means developers are not able to learn about the validity of the configuration until it is evaluated in its target operating environment.
- Even if each side-effect executes successfully, the result of each side-effect may produce an invalid value for its respective variable in the configuration. This can make the Dynamic Variable Configuration invalid.

3.4 Batch Dynamic Variable Configuration Pattern

Problem statement. In Section 3.3 we discussed the Dynamic Variable Configuration pattern, which enables developers to determine portions of a configuation using context-based side-effects. Similar to the Variable Configuration pattern, the Dynamic Variable Configuration only produces a single instance configuration. When trying to *logically* group common configurations, it requires developers to rely on *ad hoc* techniques to realize the logical associations, such as grouping all configurations generated by each user based on tests that evaluate performance. Although this is acceptable in some cases, *e.g.*, testing the validity of a single instance configuration, it can negatively impact the scalability and maintainability of configurations for T&E.

[3] The configuration in Listing 6 is invalid on standard Windows-based machines

Solution. The *Batch Dynamic Variable Configuration* pattern builds upon the Dynamic Variable Configuration pattern and enables enterprise DRE system developers to logically group common configurations that are derived at once. Developers leverage the Batch Dynamic Variable Configuration pattern by defining logically related dictionaries, such as all the dictionaries used to generate configurations for evaluating security, in a single monolithic dictionary.

The Batch Dynamic Variable Configuration has the same formal definition of the Dynamic Variable Configuration (see Section 3.3) since it is using a single template. It, however, has a different evaluation function as illustrated by Equation 4:

$$DC'' = batcheval(C, D', \Delta) = \{\forall d \in D' : eval(eval(C, d), \Delta)\} \quad (4)$$

where D' is the set of dictionaries such that if $d \in D'$ then d is an instance of D (see Section 3.1) and DC'' is the set of configurations such that $C' = eval(C, d)$, $DC' = eval(C', \Delta)$, and $DC' \in DC''$.

Manifestation in CUTE and QED. We have realized the Batch Dynamic Variable Configuration pattern in CUTE and applied it to the QED project. QED testers leverage the Batch Dynamic Variable Configuration pattern using the following steps:

1. Define a template configuration file that contains variables and side-effects similar to the process in the Dynamic Variable Configuration pattern (see Section 3.3).
2. Define a set of dictionaries in a single file where each individual dictionary defines the key-value pair for each variable in the template configuration file.

Using the user-defined template file and batch configuration file, CUTE applies Equation 4 to produce a set of configuration files. QED testers then use the derived configuration files to evalate the system under development.

```
1  config (veryLowCPU.cdp) {
2    cpuTime=12.5
3    userName=`/usr/bin/whoami`
4  }
5
6  config (veryHighCPU.cdp) {
7    cpuTime=207.1
8    userName=`/usr/bin/whoami`
9  }
```

Listing 8. Batch dictionary for D&C that uses the Dynamic Variable Configuration pattern.

Listing 8 illustrates a set of dictionaries for template in Listing 6. As highlighted in Listing 8, there are two different configurations named: `veryLowCPU.cdp` (line 1) and `veryHighCPU.cdp` (line 6). Likewise, Listing 9 shows the evaluation of the D&C excerpt for the batch configurations in Listing 8.

```
1  // veryLowCPU.cdp
2  ...
3    <configProperty>
4      <name>cpuTime</name>
5      <value>
```

```
6           <type><kind>tk_long </kind></type>
7           <value><long>12.5</long></value>
8         </value>
9       </configProperty>
10      <configProperty>
11        <name>testOwner</name>
12        <value>
13          <type><kind>tk_string </kind></type>
14          <value><string>hillj </string></value>
15        </value>
16      </configProperty>
17  ...
18
19  // veryHighCPU.cdp
20  ...
21    <configProperty>
22      <name>cpuTime</name>
23      <value>
24        <type><kind>tk_long </kind></type>
25        <value><long>207.1</long></value>
26      </value>
27    </configProperty>
28    <configProperty>
29      <name>testOwner</name>
30      <value>
31        <type><kind>tk_string </kind></type>
32        <value><string>hillj </string></value>
33      </value>
34    </configProperty>
35  ...
```

Listing 9. Evaluation of the Batch Variable Configuration pattern for the D&C.

Without Batch Dynamic Variable Configuration pattern, QED testers would have a hard time logically maintaining and producing multiple configuration files used to evaluate system QoS. CUTE, therefore, helps increase the adaptability and configurability T&E configurations.

Pattern evaluation. In addition to the benefits of the Dynamic Variable Configuration and Batch Variable Configuration, the following is a benefit for using the Batch Dynamic Variable Configuration pattern:

– Reduces the amount of single instance configurations needed to logically associate common configurations.

Although the Batch Dynamic Variable Configuration pattern has many benefits, it has several consequences. In addition to the consequences from the Dynamic Variable Configuration and Batch Variable Configuration Pattern, the Batch Dynamic Variable Configuration pattern has the following consequence:

– The batch configuration is *valid* if and only if all the individual configurations in the batch file are valid.

4 Quantitative Analysis of Template Patterns

In this section, we quantify the improvement testers gain for T&E from using each template pattern in the context of an example test scenario from the QED project.

4.1 Multistage Workflow Application Scenario

The *multistage workflow application* scenario is an example T&E scenario designed to test QED's ability to ensure application-level QoS properties, such as reliability and end-to-end response time, when handling applications with different priorities and privileges. The multistage workflow application illustrated in Figure 3 is composed of six different components (each represented by a rectangle object). The lines between each component represents a communication channel that must pass through both QED and the GIG middlware as illustrated in Figure 1.

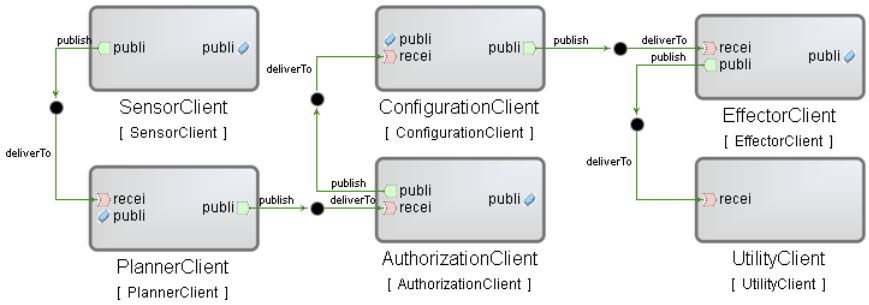

Fig. 3. Model of the multistage workflow application

Each component in the multistage workflow application contains behavior and workload, which can be configured by QED testers at D&C time using configuration files similar to the listings in Section 3. For example, developers can configure the periodicity of the `SensorClient` depending on what aspects of the QED they are testing (*e.g.*, using high periodicity to produce high utilizations) or alter the throughput ratio for an input event triggering an output event to produce different network workloads.

In addition, all experiments involving the multistage worklow application are conducted in ISISlab. Each development group on the QED project (*i.e.,* BBN Technologies, Boeing, and IMHC) uses the multistage workflow application to test their features in QED. This means T&E for the multistage workflow application must be able to handle a wide variety of configurations, operational scenarios, heterogeneity, such as different users executing the same test scenario under different ISISlab projects, or varying the CPU workload or event publish rate of each component to produce different effects on QED and the GIG middleware. Overall, the multistage workflow application provides QED testers with 11 points-of-variability for T&E.

If the QED testers relied on traditional techniques, such as handcrafted configuration files, then they would have a hard time managing T&E for the multistage workflow

application. QED testers, therefore, leverage CUTE and its template patterns to increase the scalability and configurablity of the multistage workflow application. More importantly, they elect to use CUTE because CUTE will help them increase their scope of T&E and evaluate each point-of-variability. The remainder of this section quantifies the improvement the template patterns manifested in CUTE provide QED testers.

4.2 Quantitative Analysis Results

To evaluate the improvement gained from CUTE, we calculate the number of single instance configuration files derived from each template pattern in Section 3, which is analogous to the number of *ad hoc* configuration files QED testers would have to manually create for T&E using traditional techniques. Although we discussed four template patterns in Section 3, the Batch Variable Configuration, Dynamic Variable Configuration, and Batch Dynamic Variable Configuration pattern are defined in terms of the Variable Configuration pattern. The subtle difference is when testers realize the valid *range* of each variable, *i.e.*, the set of valid values for the variable.

We, therefore, can determine the number of single instance configuration files by analyzing the Variable Configuration pattern. Equation 5 highlights the equation for determining the number of single instance configuration files $|S|$—where S is the set of single instance configuration files—based on the points-of-variability in a single configuration (see Section 3.1).

$$|S| = \prod_{v \in V} |range(v)| \tag{5}$$

As illustrated in Equation 5, $range(v)$ is the set of valid values for variable $v \in V$. The number of the number of single instance configurations S that is realizable from a single template configuration, therefore, is determined by the product of the size each variable's range, *i.e.*, $range(v)$. For example, using only a subset of the 11 points-of-variability in the multistage workflow application, if each component has a `cpuTime` integer variable (see Listing 1) that determines its CPU workload for the current test, and valid values for `cpuTime` are [10,100] msec, then QED testers can derive $(100 - 10) \times 6 = 540$ different single instance configurations from a single template configuration. Likewise, if there are 5 different QED testers executing the multistage workflow application in different ISISlab experiments and there is a dynamic variable named `userName=`\`/usr/bin/whoami\`` to determine who is executing the test for accountability reasons, then the number of possible single instance configurations increases to $540 \times 5 = 2700$.

To QED testers, this a great improvement over manually handcrafting 540 or 2700 different single instance configurations, especially when a large portion of the single instance configuration is constant (or static) between different configurations. Instead, QED testers focus on defining the different dictionaries used to instantiate the template configuration. This also helps concentrate QED testers efforts on locating points-of-variability in the T&E efforts, which can be used to conduct more controlled experiments against both QED and the GIG.

5 Related Work

This section compares our work on CUTE with related work on existing approachs for addressing T&E configurability and scalability.

Model-driven engineering techniques. Model-driven engineering (MDE) [19] is a common solution for improving T&E configurability and scalability. Existing MDE tools, such as GME [15], GEMS [23], and Microsoft DSL Tools [6], enable testers to construct domain-specific modeling languages (DSMLs) that capture the context and constaint of an application domain, such as T&E. Moreover, models constructed using a DSML can be tranformed by model interpreters into concrete archifacts, such as T&E configuration files. Such tools, however, facilate generation of single instance configuration files only.

CUTE extends existing MDE techniques by replacing single instance configuration files with template files, and delay realization single instance configuration file using its template patterns. Furthermore, CUTE synergizes with existing MDE techniques because it is possible to generate the configuration templates from constructed models. This, therefore, reduces the number of models tester must create to evaluate a enterprise DRE system under different configurations and operating scenarios (or environments).

Programmatic techniques. Template libraries, such as Google Templates [9] and CodeSmith [21], enable developers to programmtically construct template engines for generating files, such as T&E configuration files. End-users, such as testers, then define dictionaries to derive concrete files via the Template Configuration pattern. In a similar fashion, CUTE enables derivation of single instance configuration files using the Variable Configuration Pattern. CUTE extends existing template engines by enabling batch processing of configuration template. We are aware that such support can be added to existing template engines by handling multiple configurations sequentially.

CodeSmith also has the notion of what we would consider dynamic variables in its template engine. Unlike CUTE, CodeSmith evaluates its dynamic variables outside of its target environment, such as the testbed for T&E. CUTE extends CodeSmith's effort by allowing evaluating dynamic variables based on its target operating environment. Consequently, this improves both the configurability and flexibility of CUTE's template engine above and beyond what CodeSmith's capabilities.

6 Concluding Remarks

Increasing T&E scope and scale can improve understanding and evaluation of enterprise DRE system QoS concerns, such as performance, reliability, and security. this paper presented four template patterns for improving T&E configurability and scalability named. Each template pattern has been realized in the CUTE template engine. Testers use CUTE by defining configuration templates, and delay realization of single instance configuration until late in the system lifecycle, *e.g.*, when enough detail is known about the operating environment. Our analysis of CUTE on a representative enterprise DRE project showed that it can significantly increase T&E scope without increasing the number of single instance configurations required to reach that level of

T&E. Moreover, our quantitative analysis showed that CUTE can improve enterprise DRE system QoS evaluation.

We learned the following lessons while developing and applying CUTE and its template patterns to the QED project:

- Handcrafting template configurations for T&E can be labor intensive, especially if the template configurations are dense XML files. MDE techniques, such as domain-specific modeling languages, help alleviate the complexity of handcrafting such files via model interpreters that transform constructed models into concrete files. Our future work will integrate the template patterns in CUTE with MDE tools, such as GME, to improve the shortcomings of MDE tools and CUTE.
- Although CUTE can generate many different single instance configuration using a single template configuration, testers must manually run each configuration to evaluate enterprise DRE system QoS. Continuous integration environments, such as CruiseControl (`cruisecontrol.sourceforge.net`) alleviate the complexity of manually executing tests via an autonomous build engine. Our future work will combine CUTE with continuous integration environments to improve the efficiency and effectiveness of running many tests continuously throughout the software lifecycle—especially when integrated with system execution modeling tools [10].

CUTS and CUTE are available in open-source format at `www.dre.vanderbilt.edu/CUTS`.

References

1. BBN Technologies Awarded $2.8 Million in AFRL Funding to Develop System to Link Every Warfighter to Global Information Grid. BBN Technologies—Press Releases, www.bbn.com/news_and_events/ press_releases/2008_press_releases/pr_21208_qed.
2. Global Information Grid. The National Security Agency, www.nsa.gov/ia/industry/ gig.cfm?MenuID=10.3.2.2.
3. D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.
4. W. A. Aiello, Y. Mansour, S. Rajagopolan, and A. Rosén. Competitive Queue Policies for Differentiated Services. *Journal of Algorithms*, 55(2):113–141, 2005.
5. G. Concas, M. D. Francesco, M. Marchesi, R. Quaresima, and S. Pinna. An Agile Development Process and Its Assessment Using Quantitative Object-Oriented Metrics. *Agile Processes in Software Engineering and Extreme Programming*, 9:83–93, 2008.
6. S. Cook, G. Jones, S. Kent, and A. C. Wills. *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley, 2007.
7. K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Reading, Massachusetts, 2000.
8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.
9. google-ctemplate. google-ctemplate. `code.google.com/p/google-ctemplate`, 2007.

10. J. H. Hill, J. Slaby, S. Baker, and D. C. Schmidt. Applying System Execution Modeling Tools to Evaluate Enterprise Distributed Real-time and Embedded System QoS. In *Proceedings of the 12th International Conference on Embedded and Real-Time Computing Systems and Applications*, Sydney, Australia, August 2006.
11. C.-Y. Huang and M. R. Lyu. Optimal Release Time for Software Systems Considering Cost, Testing-Effort, and Test Efficiency. *IEEE Transactions on Reliability*, 54(4):583–591, 2005.
12. S. E. Institute. Ultra-Large-Scale Systems: Software Challenge of the Future. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, Jun 2006.
13. Internet Engineering Task Force. Differentiated Services Working Group (diffserv) Charter. www.ietf.org/html.charters/diffserv-charter.html, 2000.
14. G. Karsai, A. Agrawal, F. Shi, and J. Sprinkle. On the Use of Graph Transformations in the Formal Specification of Computer-Based Systems. In *Proceedings of IEEE TC-ECBS and IFIP10.1 Joint Workshop on Formal Specifications of Computer-Based Systems*, pages 19–27, Huntsville, AL, USA, Apr. 2003. IEEE.
15. Á. Lédeczi, Á. Bakay, M. Maróti, P. Völgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing Domain-Specific Design Environments. *Computer*, 34(11):44–51, 2001.
16. Object Management Group. *Deployment and Configuration Adopted Submission*, OMG Document mars/03-05-08 edition, July 2003.
17. Object Management Group. *CORBA Components v4.0*, OMG Document formal/2006-04-01 edition, Apr. 2006.
18. R. Ricci, C. Alfred, and J. Lepreau. A Solver for the Network Testbed Mapping Problem. *SIGCOMM Computer Communications Review*, 33(2):30–44, Apr. 2003.
19. D. C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.
20. Software Composition and Modeling (Softcom) Laboratory. Constraint-Specification Aspect Weaver (C-SAW). www.cis.uab.edu/ gray/Research/C-SAW, University of Alabama, Birmingham, AL.
21. C. Tools. CodeSmith Tools. `www.codesmithtools.com`, 2009.
22. M. Tortonesi, C. Stefanelli, N. Suri, M. Arguedas, and M. Breedy. Mockets: A Novel Message-Oriented Communications Middleware for the Wireless Internet. In *International Conference on Wireless Information Networks and Systems (WINSYS 2006)*, August 2006.
23. J. White, D. C. Schmidt, and A. Gokhale. Simplifying autonomic enterprise java bean applications via model-driven engineering and simulation. *Journal of Software and System Modeling*, 7(1):3–23, 2008.