2017

A Runtime Verification and Validation Framework for Self-Adaptive Software

David B. Sayre
Nova Southeastern University, ds1258@nova.edu

A Runtime Verification and Validation Framework for Self-Adaptive Software

by

David B. Sayre

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in
Computer Science

Graduate School of Computer and Information Sciences
Nova Southeastern University

2017

We hereby certify that this dissertation, submitted by David Sayre, conforms to acceptable standards and is fully adequate in scope and quality to fulfill the dissertation requirements for the degree of Doctor of Philosophy.

Gregory E. Simco, Ph.D. Chairperson of Dissertation Committee	Date
Francisco J. Mitropoulos, Ph.D. Dissertation Committee Member	Date
Sumitra Mukherjee, Ph.D. Dissertation Committee Member	Date

Approved:

Yong X. Tao, Ph.D., P.E., FASME Dean, College of Engineering and Computing	Date

College of Engineering and Computing
Nova Southeastern University

2017

An Abstract of a Dissertation Submitted to Nova Southeastern University in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

A Runtime Verification and Validation Framework for Self-Adaptive Software

by
David B. Sayre
February 2017

The concepts that make self-adaptive software attractive also make it more difficult for users to gain confidence that these systems will consistently meet their goals under uncertain context. To improve user confidence in self-adaptive behavior, machine-readable conceptual models have been developed to instrument the adaption behavior of the target software system and primary feedback loop. By comparing these machine-readable models to the self-adaptive system, runtime verification and validation may be introduced as another method to increase confidence in self-adaptive systems; however, the existing conceptual models do not provide the semantics needed to institute this runtime verification or validation.

This research confirms that the introduction of runtime verification and validation for self-adaptive systems requires the expansion of existing conceptual models with quality of service metrics, a hierarchy of goals, and states with temporal transitions. Based on this expanded semantics, runtime verification and validation was introduced as a second-level feedback loop to improve the performance of the primary feedback loop and quantitatively measure the quality of service achieved in a state-based, self-adaptive system.

A web-based purchasing application running in a cloud-based environment was the focus of experimentation. In order to meet changing customer purchasing demand, the self-adaptive system monitored external context changes and increased or decreased available application servers. The runtime verification and validation system operated as a second-level feedback loop to monitor quality of service goals based on internal context, and corrected self-adaptive behavior when goals are violated. Two competing quality of service goals were introduced to maintain customer satisfaction while minimizing cost. The research demonstrated that the addition of a second-level runtime verification and validation feedback loop did quantitatively improve self-adaptive system performance even with simple, static monitoring rules.

Acknowledgments

The time and effort required to accomplish this dissertation would not have been possible without the support of my wife Demita Sayre. She made it possible for me to focus on this effort and covered for the many family events I missed. My children – Isaac, Hannah, Caleb and Rebekah – bore the brunt of my absence, and I truly appreciate the understanding that they have extended to me. Thank you family for the grace you have given me over the years to accomplish this goal. I must also thank my mother June Sayre, father Hursel Sayre, and sister Sharon Highfield who have been constant advocates for my continued education.

No dissertation effort is possible without the continuing guidance and direction of a dissertation advisor. Thank you, Dr. Simco for the steady hand in guiding my work, and encouraging me at times when roadblocks seemed too big. Thank you also to Dr. Mukherjee and Dr. Mitropoulos for also supporting me as members of this dissertation committee.

I must also thank my peers Dr. John Hudzina, Dr. Ron Krawitz, and Dr. Ray Halper, who have been trailblazers, encouragers, and sounding boards throughout the coursework and dissertation process. Lastly, I must acknowledge my lifelong friend Dr. Charles Bryant who daily encouraged and held me accountable to achieve this significant life goal.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Background

Self-adaptive software (SAS) employs a feedback loop where sensors monitor the target application along with the surrounding environment. Decision components make choices that maintain system goals, and effectors issue commands to modify the system structure or application behavior. Sensors focus on detection of the target application's system state, called internal context, as well as the state of the overall computing environment, called external context. The commands that effectors issue can be imposed at any phase in the system lifecycle to maintain requirements, not only during normal system operation (Cheng, Lemos, & Giese, 2009).

This research showed that the introduction of runtime verification and validation for self-adaptive systems required the expansion of existing conceptual models with quality of service metrics, a hierarchy of goals, and states with temporal transitions. Based on these expanded semantics, runtime verification and validation was introduced as a second-level feedback loop to improve the performance of the primary feedback loop to quantitatively measure the quality of service achieved in a state-based, self-adaptive system.

One of the original representations of a generic adaption loop was the MAPE-K (Monitoring, Analyzing, Planning, and Executing with Knowledge) structure shown in Figure 1 (Salehie & Tahvildari, 2009). This general concept provided a structure on which many of the self-adaptive feedback models were built. This work also implemented a feedback loop that provided both self-adaptive and runtime verification and validation functionality using a hierarchical structure of quality of service goals. The Salehie and Tahvildari (2009) feedback concept was foundational to this research.

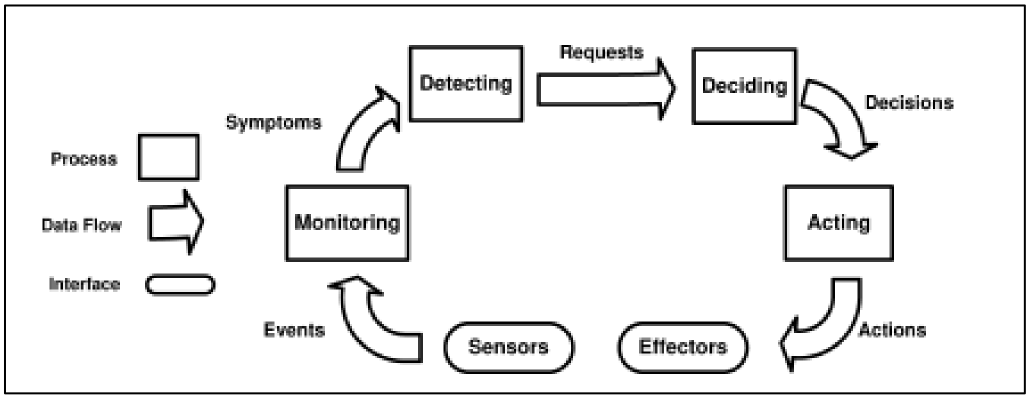


Figure 1. Four adaption processes in self-adaptive software (Salehie & Tahvildari, 2009)

Software increasingly depends on layers of infrastructure that are labor intensive to set up and maintain, regardless of whether the software implements a web site or controls an airplane. While the SAS approach does add structural complexity to applications, it can simplify problems throughout the software lifecycle. During installation, SAS can achieve a successful software configuration with differing hardware capabilities (Salehie & Tahvildari, 2009). The MAPE-K structure may also be applied to minimize an application's energy consumption, while still maintaining normal operation (Calinescu & Kwiatkowska, 2009). When components fail, SAS can re-host applications on the remaining data center infrastructure (Arshad, Heimbigner, & Wolf, 2004). Software components can be added to a running system and then later invoked upon context changes, thus reducing maintenance downtime (Calinescu, Ghezzi, Kwiatkowska, & Mirandola, 2012). Industry has focused on reducing the cost of operating today's distributed, power hungry applications, and SAS approaches have demonstrated research solutions to these problems.

The concepts that made SAS attractive have also made it more difficult for users to gain confidence that these systems consistently met their goals (Tamura et al., 2012). Users and developers gain trust in software systems by first verifying that the software has achieved its functional requirements, and then gather sufficient evidence to characterize and validate non-functional behavior (Banks, Carson, Nelson, & Nicol, 2001). Despite the stated benefits of SAS, its concepts have not been widely adopted in industry (Tamura et al., 2012). It remains difficult to attain human trust in applications that are specifically designed to modify themselves (Dahm, 2010; Tamura et al., 2012). The cost of mechanically testing complex, conventional applications has become a growing component of the overall systems engineering process (Feldt, Torkar, Ahmad, & Raza, 2010; Laurent, 2010). The ability to properly test and verify SAS behavior is also costly, and has emerged as a substantial research challenge (Calinescu et al., 2012); therefore, for SAS to gain wider acceptance, a method of verifying and validating behavior was needed to provide trustworthy results for users and developers.

The more established discipline of modeling and simulation (M&S) faces the similar challenge of attaining human trust in the computer programs that simulate real systems (Banks et al., 2001). Banks et al. (2001) described verification as comparing the computer program against a conceptual model. This model may be a document, a process chart, a state diagram, or a modeling language. The conceptual modeling approach represents all the branching decisions that a simulation can employ. The level of validation needed to gain trust in SAS requires V&V methods similar to those used in M&S, except that they must be active participants not just data collectors (Tamura et al., 2012). SAS can invoke a nearinfinite set of states, and these states can't be fully represented in a model or quantitatively verified (Calinescu et al., 2012; Lemos, Giese, Muller, & Shaw, 2011; Tamura et al., 2012). Cheng et al. (2009) proposed that this explosion in possible state transitions is an integral part of SAS and use the term uncertainty to define this characteristic. This concept of uncertainty acknowledges that all the possible SAS branching decisions can't be fully documented in a conceptual model or traditional software requirements; however, the system must still maintain its goals. SAS lacked a descriptive modeling capability that will allow users and developers to verify system operation and validate non-functional behavior.

While the concept of uncertainty is embraced by SAS, the concept does not map well to traditional V&V methods where each branching decision must be assessed (Cheng et al., 2009). Thus, for SAS to be more widely adopted, a representation of the system was required that was a V&V enabler. It has been possible to integrate V&V at runtime so that self-adaptive systems can capture validation data when new branching decisions are observed (Dahm, 2010; Tamura et al., 2012). Like M&S conceptual models, SAS validation required that substantial evidence be gathered throughout the system and correlated with input data to help users and developers gain confidence in system behavior. SAS V&V required a conceptual representation that integrates system goals with overarching V&V direction (Villegas, Muller, & Tamura, 2011). By integrating adaptive goals and validation logic into a single model, overall system behavior could be maintained while necessary and sufficient V&V evidence (Banks et al., 2001) is collected from a running system.

This work established that current SAS models (Calinescu et al., 2012; Villegas et al., 2011) are not expressive enough to describe self-adaption with states and a hierarchy of

V&V goals necessary to measurably improve the quality of SAS applications. This dissertation extended the Villegas et al. (2011) self-adaptive model with a V&V layer and quality of service (QoS) measures (Fu, Zou, Jiang, & Shang, 2007). This new model was demonstrated by developing a simulation that is the combination of the Tamura et al. (2012) benchmark example and concepts from the Arshad et al. (2004) example. This example verified system goals by measuring the value of RV&V when integrated with the primary adaptation loop.

The addition of V&V, as a necessary component of SAS, is the focus of work by Tamura et al. (2012), and they proposed that Runtime V&V would benefit the feedback loop. By expanding upon the MAPE-K architecture with Runtime V&V (RV&V) components shown in Figure 2, a complementary framework was able to evaluate both the target system and the feedback loop. A Runtime Validator and Verifier evaluated context changes proposed by the Planner and determined if they would violate overall system consistency. This evaluation of consistency was not based on the running system, but was based on the structure of the underlying model at a future point in time (Tamura et al., 2012). Consistency was viewed as a self-adaptive property where the adaptable system and environment was maintained to the system's conceptual model. It was not a measure of whether the system is similar to itself over time (Tamura et al, 2012). SAS may change its structure to meet overall

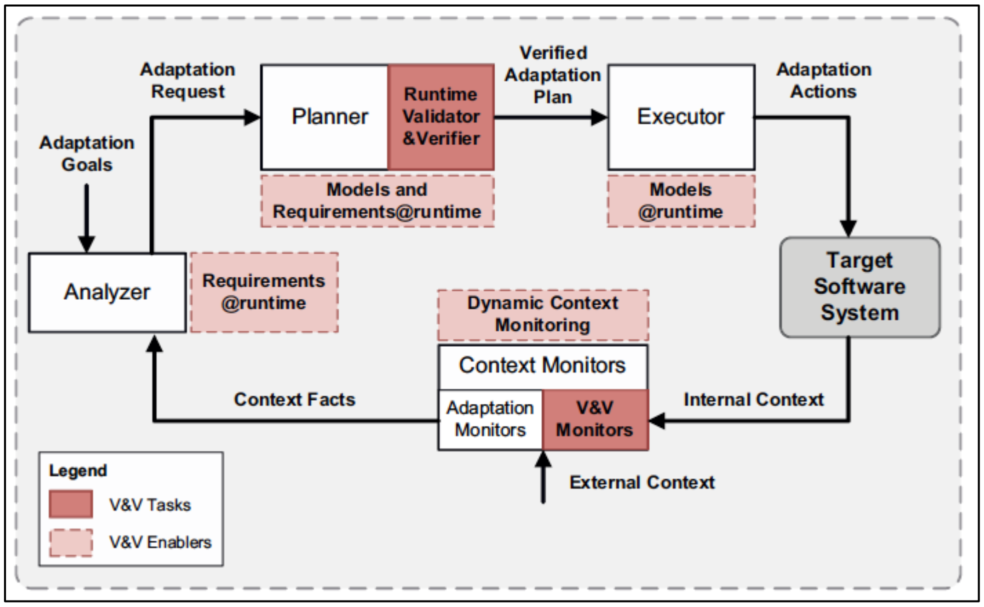


Figure 2. Verification and validation tasks added to the SAS loop (Tamura et al., 2012)

goals, but maintains consistency to its model. The Planner was a more specific form of the MAPE-K Deciding component that elicits commands to migrate the system to its next stable state. Planning is an Artificial Intelligence (AI) method that was implemented by Arshad et al. (2004) as the Deciding component of the Salehie & Tahvildari (2009) adaption loop (Russell & Norvig, 2010). The goals that the RV&V system maintains at the output of the Planner were typically not the goals of the current system state, but those of the next most likely state that the Planner had to achieve.

The V&V Monitor in Figure 2 was concerned with evaluation and enforcement of current state behavior, and collection of validation data. Thus, the RV&V architecture proposed by Tamura et al. (2012) was active and instituted a second layer of adaption when RV&V goals were violated. The Adaption Monitor identifies and serializes changes in context for use by the Analyzer; however, the V&V Monitor performs the higher-level functions of capturing RV&V goal violations for the Requirements@Runtime Analyzer

(Sawyer, Bencomo, & Whittle, 2010). The RV&V Analyzer determined if a state transition must occur based on context updates. When a new state has been achieved, the RV&V Monitor updates its own data structures.

Tamura et al. (2012) provided an example problem that can be used to benchmark RV&V experiments. This example problem was a web purchase ordering service that is managed by an adaption loop. This adaption loop adds and removes servers in a cloud environment where the addition of servers implies additional cost per unit time ("Amazon web services," 2013). These decisions to add or subtract services are based on aspects such as customer load, sale days, and customer affinity. This example maintained multiple goals, such as the minimization of customer wait time. Customers are stratified into normal and premium groups. Each group had a wait time goal, and the system modified itself to achieve those goals despite different purchasing loads throughout the test cases.

Another goal of the benchmark example was to maximize throughput while minimizing the cost of the cloud-based system. This goal was distinct from any single adaption decision, but involves measurement and optimization of the purchasing service and primary adaption loop. This overarching RV&V goal was very similar to previous work on QoS measures described by the Fu et al. (2007) measurement hierarchy. Fu et al. (2007) defined an ontology for QoS criteria that could be integrated into an RV&V model to establish standard terms, and was shown in Figure 3 below. The categories of performance, availability, economic value, and reliability all directly correlate to items that an RV&V platform should validate. Fu et al. (2007) also proposed that the calculation of the lowest level terms in the ontology provide a means for identifying an overall metric called consistency. This consistency measure is then used to establish a reputation for individual web services, but the Fu et al. (2007) concept of consistency differs from the self-adaptive property. The concept of reputation might also be integrated into an RV&V platform to aid in the tuning provided to the feedback loop.

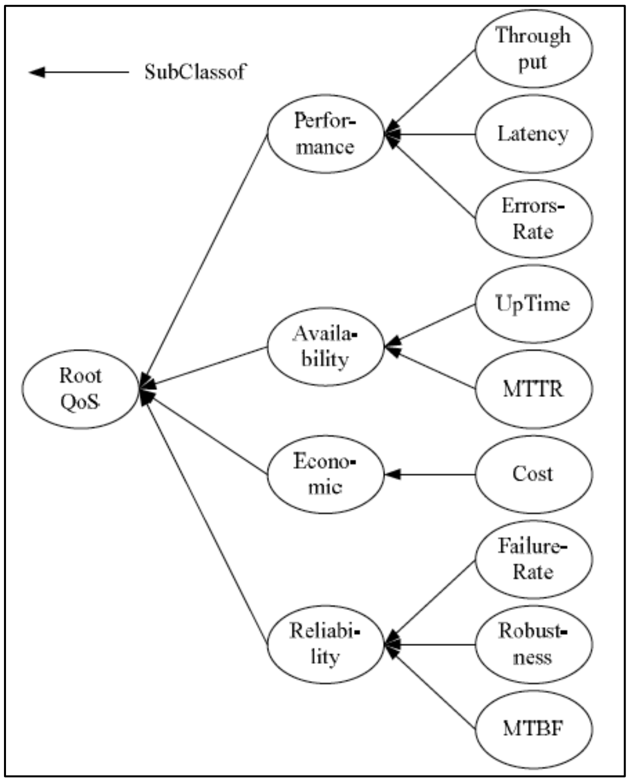


Figure 3. QoS Ontology (Fu et al., 2007)

Independent of the adaption system, the RV&V system was concerned with the combined behavior of the target system and the adaptation loop. The time that it took to transition from one stable state to another was referred to as *settling time.* The concept of minimizing settling time was also another key RV&V property identified by Tamura et al. (2012). The RV&V system was concerned with minimizing settling time of the overall application after state transitions (Tamura et al., 2012). The RV&V system must act if adaption behavior exceeds acceptable settling time, and will then force the system into a stable state. While settling time was discussed as a global parameter in Tamura et al. (2012), individual states may have different individual settling time targets – or local settling times. In the context of the Tamura et al. (2012) benchmark example, a longer settling time incurs greater costs by the cloud service without providing additional benefit to customers. The interaction between self-adaptation and RV&V correction was briefly explored in Tamura et al. (2012), but in Calinescu et al. (2012) adaption and RV&V correction were combined. Villegas et al. (2011) documented the parameters that the system should use for adaptation decisions, but not the RV&V measures needed to verify quality measures after a state transition.   No recent literature provided a method to document the RV&V layer that surrounds the feedback loop (Calinescu et al., 2012; Tamura et al., 2012; Villegas et al., 2011).

Villegas et al. (2011) used a service-oriented architecture (SOA) example to demonstrate a Resource Description Framework (RDF) taxonomy, called SmartContext, that represents contextual entities, service-level agreements, and service-level objectives in a machine-readable taxonomy. The Villegas et al. (2011) implementation was one of the first concrete examples of a working measurement taxonomy in a self-adaptive application. The Villegas et al. (2011) approach parallels the RV&V framework outlined by Tamura et al. (2012), and utilizes the semantic structure displayed in Figure 4. This figure addresses a single Service Level Agreement (SLA), but not the overall measurement processes necessary to manage system state transitions, like those in the Tamura et al. (2012) example problem. The SLA management approach assumed that an SLA is managed at the level of an

Executable Code Unit versus at the system-level. An example of this weakness can be described in terms of settling time. The time behavior metric (TB) in Figure 4 applies to a specific Executable Code Unit, not the measurement of the system from one acceptable zone of operation to another. While the Villegas et al. (2011) approach was novel and provided a basis for expansion, it did not acknowledge that RV&V operates at a separate, higher level than the verification of normal application component behavior.

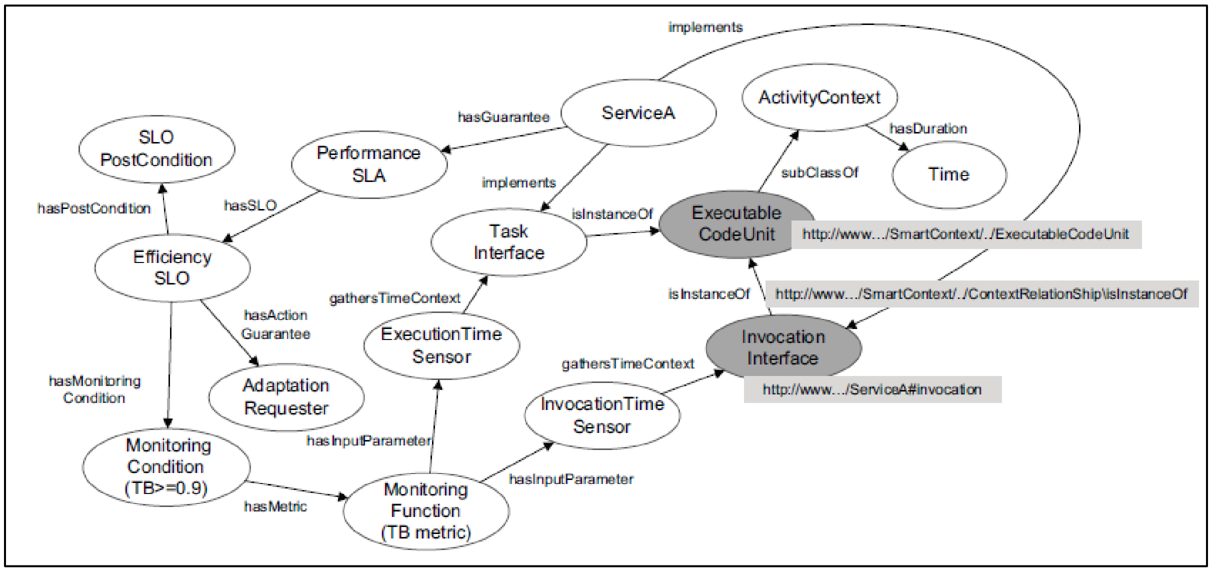


Figure 4. Service level agreement monitoring RDF structure (Villegas et al., 2011)

Movement between acceptable states of system operation and the minimization of settling time were described by Tamura et al. (2012) as behavior in support of Viability Zones. When these zones were violated, a primary feature of the RV&V layer was to intervene in the adaption process. Viability zones comprised the set of valid system states, context attributes, and their corresponding values that define uncompromised system performance (Tamura et al., 2012). RV&V actions included not only sampling the system while it remains inside its viability zone, but acting if a viability zone is compromised

(Tamura et al., 2012). Settling time could be measured by the RV&V layer after the adaptation system has decided to make an inter-state transition; however, the viability zone concept inferred that RV&V metrics be defined to trigger changes as a system nears the edge of a viability zone. The QoS metrics identified in the Fu et al. (2007) taxonomy provided a viable basic set of identifiers to meet the needs for an RV&V metric inventory.

Using the previous example of a SAS application that is nearing the edge of a viability zone, the RV&V Monitor may need to take special action and increase its sampling rate as the SAS system nears the edge of a viability zone. When the viability zone is breached, the RV&V system must start collecting settling time data, and evaluate whether a local metric is achieved. If the settling time metric was violated, the RV&V Validator may override adaption and attempt to achieve a neighboring viability zone. No RV&V data representation existed that defined viability zones, the metrics that trigger actions around the zone edges, or the actions themselves. The Villegas et al. (2011) conceptual RDF method did not provide the syntax or semantics to capture these inter-state transition features that are key aspects of RV&V. Calinescu et al. (2012) and Tamura et al. (2012) both concluded that RV&V was a necessary component of SAS, but neither defined the data types or parameters needed to measure a self-adaptive system, nor document the higher-level decisions that an RV&V system required.

While RV&V was new to SAS, it does exist in other control system domains, and is an integral component of the system test process for unmanned space vehicles. Ground tests are conducted with an active RV&V subsystem, and these runs log data for subsequent manual evaluation (Artho, Barringer, & Goldberg, 2005; Goldberg, Havelund, & McGann, 2005).

To achieve trust in autonomous systems, testing is performed in-depth prior to launch, using a combination of traditional unit testing, model checking, and RV&V. Since autonomous systems may also achieve operational states not specified during design time activities, they also must validate operational system performance, referencing an explicit model (Artho et al., 2005). Artho et al. (2005) and Goldberg et al. (2005) each developed languages that allowed for the key parameters of an autonomous system to be described and measured. Languages, such as Eagle and RuleR, allow testers to define how the system should generally respond to new context (Barringer, Havelund, Rydeheard, & Groce, 2009).

The RuleR language provided a construct where the system performs roll back operations to a consistent state if the next state can't be achieved (Barringer et al., 2009). This rollback concept is similar to the idea that that the RV&V system would intervene after the local settling time parameter was violated to achieve a stable state, as previously mentioned. State transitions were typically caused by changes in system performance or failures of some type. Thus, the RV&V system may also have to enforce load shedding behavior in order to maintain a stable state. By integrating the RuleR rollback concepts, a SAS RV&V language may be able to achieve verifiable consistency by comparing the system output to its model representation.

Another concept of the RuleR language was the inclusion of temporal logic to define state transitions. This autonomous systems concept was necessary because RV&V systems must support the evaluation of multiple, simultaneous state transitions and Propositional Temporal Logic (PTL) was the tool used to isolate state transitions into threads of execution (Artho et al., 2005; Barringer et al., 2009). PTL accounts for time by sequencing the events that must follow one another (Merz, 2001). Concepts such as *next φ, φ until φ, always φ,* and *eventually φ* were combined with traditional Boolean operators allowing a time-sequenced specification to be created without needing to provide specific time values, especially when specific time values are unknown during design. A slight variation on PTL is linear temporal logic (LTL) which states that all event states happen in a single timeline, but uses the same concepts previously noted in PTL (Baier & Katoen, 2008). LTL transition systems (TS) can be formally proven by first transforming the logic into an inverted non-deterministic Büchi automation. If a path can be found through the negation of the TS, the LTL is disproved. Model checkers show that control systems are well defined by using formal methods such as these. These powerful methods are, however, difficult to apply to data intensive applications, where states are not fully defined.

Temporal concepts were also implemented in Calinescu et al. (2012) to achieve a second-level of self-adaptive behavior that was referred to as RV&V; however, the temporal concepts were not presented as a language that separated adaption from RV&V. Calinescu et al. (2012) integrated requirements achievement, behavioral modification, and verification measurements into a single self-adaptive application. The limitation of the Calinescu et al. (2012) work was that it did not provide a language structure to guide future self-adaptive applications. The Villegas et al. (2011) SLA example was an elementary example of a structure separate from the self-adaptive feedback loop and implementation. By separating representation from implementation, a variety of solutions can evolve using a standard representation. The corollary to this structured language approach is a database schema. By having a schema, new problems can utilize existing meta-knowledge versus starting with a new data structure each time.

Like Calinescu et al. (2012), the Requirements@Runtime community required that a self-adaptive system be able to read, modify, and react to its requirements at runtime (Qureshi, Jureta, & Perini, 2011). Qureshi et al. (2011) acknowledged the need for a self-adaptive language structure that was readable by all components in the feedback loop. The Qureshi et al. (2011) approach began with an abstract self-adaptive structure, but more recently they expanded upon the abstract language with a concrete ontology language, called Adaptive RML, based on the Ontology Web Language (OWL) (Herman, 2014; Qureshi, Jureta, & Perini, 2012). In both works, Qureshi et al. (2011, 2012) argued that a semantic representation of domain context, goals, tasks, and relations required an ontology approach; however, neither work from Qureshi et al. (2011, 2012) provided a semantic structure to represent RV&V concepts. Without the expansion of a semantic structure to include RV&V concepts, SAS components in the Tamura et al. (2012) structure of Figure 2 lacked the ability to institute RV&V along with adaptation.

The basic RV&V concept of settling time can't be represented by the Qureshi et al. (2012) Adaptive RML language. Settling time was the simplest concept to apply directly to the RV&V problem, and required that consistency be re-established between measured performance and the model (Tamura et al., 2012). To provide a general-purpose language solution for RV&V, the semantics must be expressive enough to capture the RV&V concepts of state transition time and be capable of documenting new measurement criteria without language expansion (Tamura et al., 2012). Early self-adaptive examples from Arshad et al.

(2004), and more recent examples from Villegas et al. (2011) and Calinescu et al. (2012) all utilized a language semantics to achieve adaptation, but not a common semantics for RV&V that integrated with context and requirements development for SAS (Qureshi et al., 2011, 2012). The problems of self-adaptive goal representation and temporal state representation were being addressed separately from RV&V measurement and actions. To achieve the Tamura et al. (2012) RV&V structure, these two research areas must be combined into a common language structure.

The remainder of this chapter states the problem and need for a self-adaptive RV&V language. It then outlined the goal of this research, and the relevance of this work in expanding SAS RV&V understanding. A barriers and issues section highlighted the gaps in the current literature to which this work contributed, and the last section summarized this chapter. A definition of terms as used in the context of SAS RV&V was also provided.

Problem statement

The semantic language proposed by Villegas et al. (2011) and referenced by Tamura et al. (2012) as an introductory RV&V specification did not define the RV&V entities or properties needed to collect QoS measures within a viability zone, institute state management, or improve adaption decisions based on QoS goals. Each of these concepts was needed to determine if the Tamura et al. (2012) RV&V model can reduce cloud server costs when implemented in the benchmark example. The works of Qureshi et al. demonstrated the need for a language approach for self-adaptive goal behavior that began with a requirements model (Qureshi et al., 2011, 2012); however, RV&V concepts were absent from the Qureshi et al. (2012) language. The temporal approach of Calinescu et al. (2012) provided the basis for state management, but it did not introduce these concepts into a language for general use. The lack of public SAS standards and results means that SAS research has not been well integrated (Weyns, Iftikhar, de la Iglesia, & Ahmad, 2012). Neither Villegas et al. (2011), Calinescu et al. (2012), nor Qureshi et al. (2012) demonstrated the integration of self-adaptive goals and RV&V concepts. Facets of the RV&V problem were further elaborated in the context of achieving the Tamura et al. (2012) RV&V model and benchmark example.

In Figure 4, Villegas et al. (2011) provided a structure by which a single ExecutableCodeUnit can be monitored for adherence to a MonitoringCondition. This is a concrete example of a semantic structure for monitoring self-adaptive goals; however, this structure did not provide the entities or linked properties by which a system state can be monitored, integrating multiple MonitoringConditions. In Figure 5, below, the taxonomy used to populate the application example of Figure 4 also showed that there is no place to monitor application state, only the performance of a specific ExecutableCodeUnit. To evaluate the benefits of RV&V SAS, the ability to monitor and maintain state were implemented such that the evaluation of multiple MonitoringConditions contribute to a single application state, and the structure must inform the RV&V subsystem of the steps to take when a MonitoringCondition is violated. The taxonomic structure of Villegas et al. (2011) in Figure 5 did not address state transitions by which the system moves from one viability zone to another when a system state has been violated. Lastly, there was no structural component by which the RV&V system knows how to rollback if a desired state can’t be achieved within a desired settling time. State entities were required as top-level components in the taxonomy of Figure 5 to determine if the Villegas et al. (2011) semantic approach was able to demonstrate that RV&V improves SAS quality.

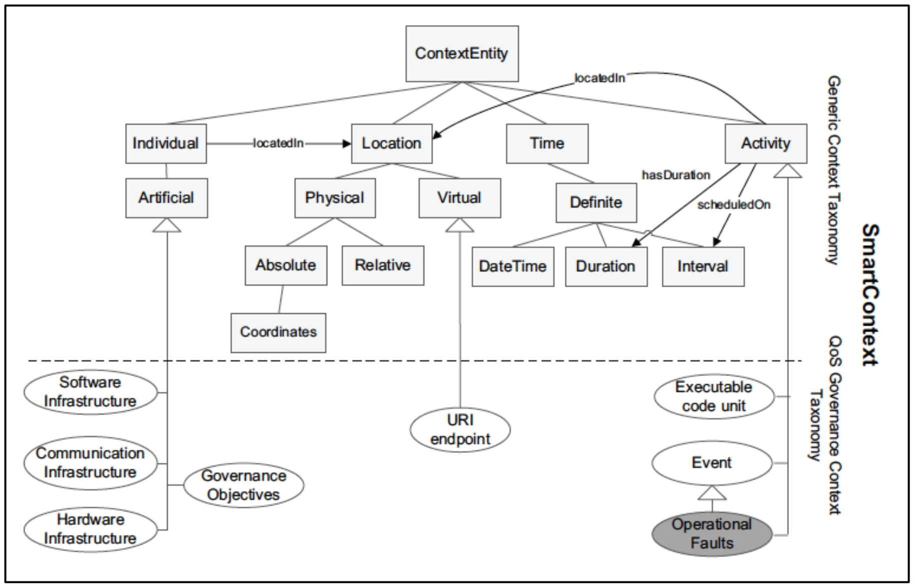
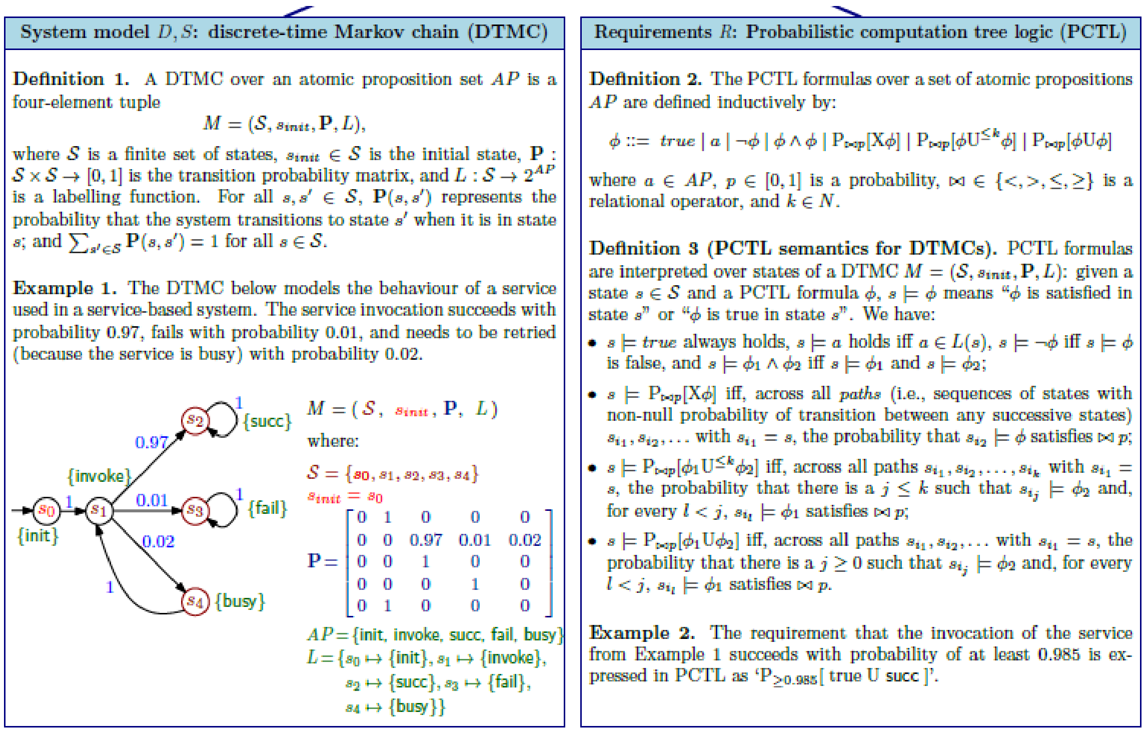


Figure 5. SmartContext (Villegas et al., 2011)

The ability to represent states and state transitions was a fundamental component of the Calinescu et al. (2012) quantitative verification approach, and was depicted in the leftmost graphic of Figure 6. The state transition diagram was populated with probabilities that are used in a Markov chain to determine the best, next-state transition. However, this approach was codified in an equation-like language called Probabilistic Computation Tree Logic (PCTL) that can't be generically applied to other problems. A published, general purpose solution for the RV&V problem would enhance the SAS literature by allowing future research to be more integrated (Weyns et al., 2012).The state transition approach is really the method by which adaptation takes place, not a higher-level RV&V. On the right side of the figure is an example of the Calinescu et al. (2012) implementation of PCTL. The combination of state transitions and temporal logic were the tools that were missing in the Villegas et al. (2011) semantic structure to evaluate RV&V metrics, such as settling time.



System model D, S: discrete-time Markov chain (DTMC)

Definition 1. A DTMC over an atomic proposition set AP is a four-element tuple

$$M = (S, s_{init}, \mathbf{P}, L),$$

where S is a finite set of states, $s_{init} \in S$ is the initial state, $\mathbf{P} : S \times S \rightarrow [0,1]$ is the transition probability matrix, and $L : S \rightarrow 2^{AP}$ is a labelling function. For all $s, s' \in S$, $\mathbf{P}(s, s')$ represents the probability that the system transitions to state s' when it is in state s; and $\sum_{s' \in S} \mathbf{P}(s, s') = 1$ for all $s \in S$.

Example 1. The DTMC below models the behaviour of a service used in a service-based system. The service invocation succeeds with probability 0.97, fails with probability 0.01, and needs to be retried (because the service is busy) with probability 0.02.

$$\begin{tikzcd}
 & & s_2 \arrow[loop right, "1"] \; \{\text{succ}\} \\
\arrow[r] & s_0 \; \{\text{init}\} \arrow[r, "1"] & s_1 \; \{\text{invoke}\} \arrow[u, "0.97"] \arrow[r, "0.01"] \arrow[d, "0.02"] & s_3 \arrow[loop right, "1"] \; \{\text{fail}\} \\
 & & s_4 \arrow[u, bend left, "1"] \; \{\text{busy}\}
\end{tikzcd}$$

$M = (S, \; s_{init}, \mathbf{P}, \; L)$

where:

$S = \{s_0, s_1, s_2, s_3, s_4\}$

$s_{init} = s_0$

$$\mathbf{P} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0.97 & 0.01 & 0.02 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

$AP = \{\text{init, invoke, succ, fail, busy}\}$

$L = \{s_0 \mapsto \{\text{init}\}, s_1 \mapsto \{\text{invoke}\},$
$s_2 \mapsto \{\text{succ}\}, s_3 \mapsto \{\text{fail}\},$
$s_4 \mapsto \{\text{busy}\}\}$

Requirements R: Probabilistic computation tree logic (PCTL)

Definition 2. The PCTL formulas over a set of atomic propositions AP are defined inductively by:

$$\phi ::= \ true \mid a \mid \neg\phi \mid \phi \wedge \phi \mid \mathrm{P}_{\bowtie p}[\mathrm{X}\phi] \mid \mathrm{P}_{\bowtie p}[\phi \mathrm{U}^{\leq k} \phi] \mid \mathrm{P}_{\bowtie p}[\phi \mathrm{U} \phi]$$

where $a \in AP$, $p \in [0,1]$ is a probability, $\bowtie \in \{<, >, \leq, \geq\}$ is a relational operator, and $k \in N$.

Definition 3 (PCTL semantics for DTMCs). PCTL formulas are interpreted over states of a DTMC $M = (S, s_{init}, \mathbf{P}, L)$: given a state $s \in S$ and a PCTL formula ϕ, $s \models \phi$ means "ϕ is satisfied in state s" or "ϕ is true in state s". We have:

- $s \models true$ always holds, $s \models a$ holds iff $a \in L(s)$, $s \models \neg\phi$ iff $s \models \phi$ is false, and $s \models \phi_1 \wedge \phi_2$ iff $s \models \phi_1$ and $s \models \phi_2$;
- $s \models \mathrm{P}_{\bowtie p}[\mathrm{X}\phi]$ iff, across all *paths* (i.e., sequences of states with non-null probability of transition between any successive states) $s_{i_1}, s_{i_2}, \ldots$ with $s_{i_1} = s$, the probability that $s_{i_2} \models \phi$ satisfies $\bowtie p$;
- $s \models \mathrm{P}_{\bowtie p}[\phi_1 \mathrm{U}^{\leq k} \phi_2]$ iff, across all paths $s_{i_1}, s_{i_2}, \ldots, s_{i_k}$ with $s_{i_1} = s$, the probability that there is a $j \leq k$ such that $s_{i_j} \models \phi_2$ and, for every $l < j$, $s_{i_l} \models \phi_1$ satisfies $\bowtie p$;
- $s \models \mathrm{P}_{\bowtie p}[\phi_1 \mathrm{U} \phi_2]$ iff, across all paths $s_{i_1}, s_{i_2}, \ldots$ with $s_{i_1} = s$, the probability that there is a $j \geq 0$ such that $s_{i_j} \models \phi_2$ and, for every $l < j$, $s_{i_l} \models \phi_1$ satisfies $\bowtie p$.

Example 2. The requirement that the invocation of the service from Example 1 succeeds with probability of at least 0.985 is expressed in PCTL as '$\mathrm{P}_{\geq 0.985}$[true U succ]'.

Figure 6. Propositional temporal logic semantics and state diagram (Calinescu et al., 2102)

Qureshi et al. (2012) built their self-adaptive ontology language by expanding upon a previous requirements language – Techne - and the revised language, called Adaptive RML, includes only four minor additions (Jureta, Borgida, Ernst, & Mylopoulos, 2010). One of these additions was the concept of a quality constraint, and an example is shown in the top right diamond of Figure 7, titled "Message sent in < 1 hour after the Payment". The completion of two subordinate goals was verified by the measurement of this overarching quality constraint. The addition of quality constraints was similar to the measurement of viability zones desired by Tamura et al. (2012), the implementation of MonitoringConditions by Villegas at al. (2011), and the Fu et al. QoS ontology (2007); however, the self-adaptive system described by Adaptive RML was not given any direction regarding how to react when the quality constraint is violated. Without defining the steps to take (1) when gathering quality measurements, (2) when a quality constraint was violated, or (3) when the system was back in a stable state, RV&V does not occur. As well, the concept of validation implied that

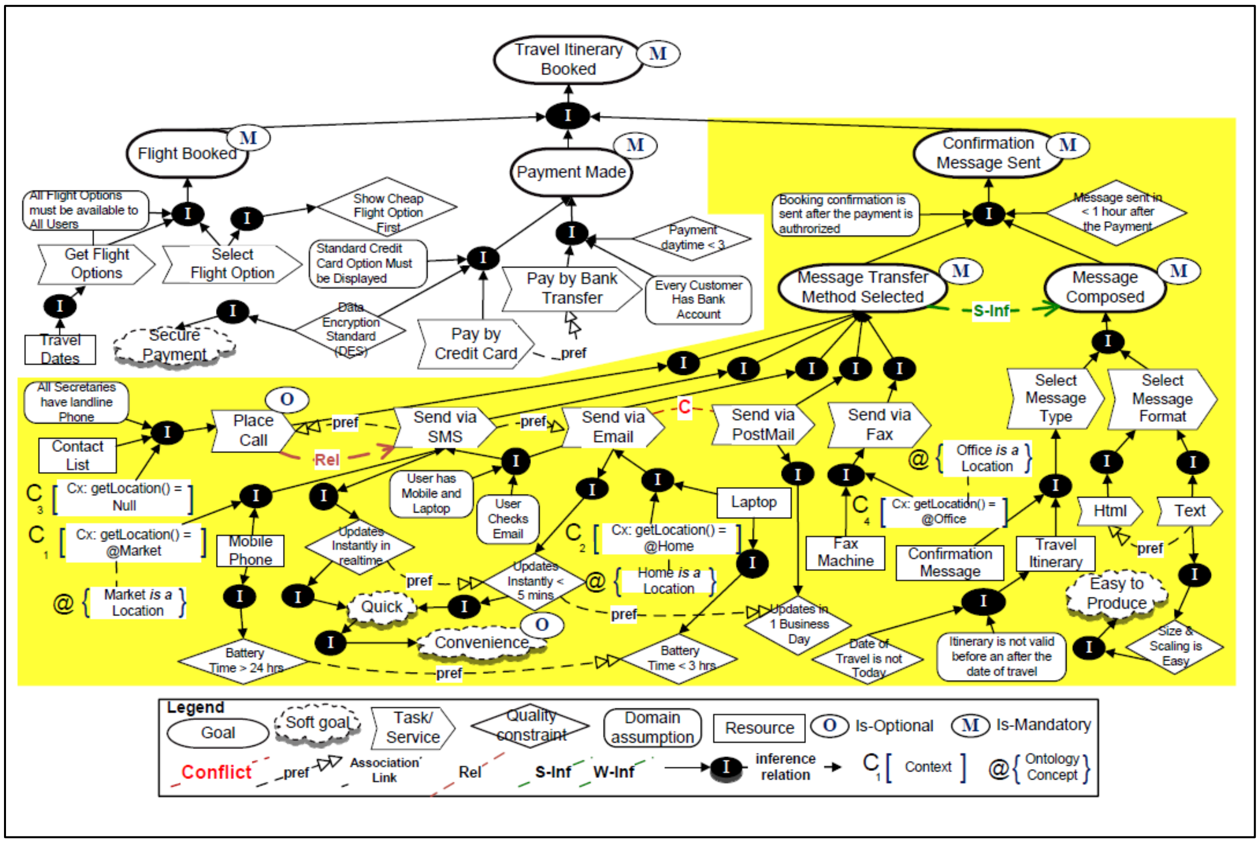


Figure 7. Adaptive RML diagram (Qureshi et al, 2012.)

the system was being calibrated to best meet its functional and non-functional goals (Banks et al., 2001). This calibration functionality was also missing from the Qureshi et al. (2012) modeling language, and additional entities and properties were required for RV&V to operate at a higher, independent level from self-adaption.

Each of the works referenced in this section provided building blocks by which an RV&V capability may be described, but none actually implement the extensions needed to achieve the RV&V goals set out by Tamura et al. (Calinescu et al., 2012; Qureshi et al., 2012; Tamura et al., 2012; Villegas et al., 2011).To appropriately model an RV&V system, monitoring conditions must exist between goals that identify non-functional quality constraints. A set of remediation actions must then describe what actions to take when quality is violated, and the modeling language must describe how the system achieves a stable viability zone where unhindered self-adaptive behavior can proceed. The entities and properties necessary to evaluate RV&V behavior in self-adaptive systems were not defined in any of these referenced works (Calinescu et al., 2012; Fu et al., 2007; Qureshi et al., 2012; Tamura et al., 2012; Villegas et al., 2011).

Dissertation Goal

The goal of this dissertation was to show that the addition of states, temporal logic and goals to the Villegas et al. (2011) modeling language and the Tamura et al. (2012) feedback loop would maintain customer QoS goals and reduce cloud server costs for the Tamura et al. (2012) benchmark example. There were no SAS runtime modeling languages that informed the feedback loop by the using QoS metrics, calibrated self-adaptive behavior, reacted when a monitoring condition is violated, or evaluated settling time. Because an example of an RV&V implementation using the Tamura et al. (2012) model was unavailable, RV&V had not been demonstrated as a method for gaining trust in SAS. This research provided an example implementation. By implementing the self-adaptive reference problem from Tamura et al. (2012) with states decorated with temporal logic, a baseline set of data was gathered.

Then, RV&V measurement and intervention was added to the same problem, and QoS achievement was measured. By implementing additional entities and properties to the Villegas et al. (2011) language, along with temporal logic concepts, self-adaptive RV&V behavior was conveyed to a second-level feedback loop, independently of the basic feedback loop. The resulting self-adaptive language and demonstration simulator implemented a second-level of adaptive behavior where the RV&V loop intervenes in the actions of the self-adaptive loop. Based on the addition of this RV&V capability, the Tamura et al. (2012) RV&V simulation demonstrated lower operational costs when compared against a SAS-only baseline.

Relevance and Significance

The Internet – along with mobile devices – have enabled corporations to become an integrated part of their customer’s lives, but this desire to present a ubiquitous corporate presence came at the price of increasing complexity (Kephart & Chess, 2003). The problem of software complexity extends to every facet of the computing domain, from design to sustainment. Kephart and Chess (2003) challenged the computing community to address the problem of complexity by taking a holistic look at design, development, testing, deployment and maintenance of software-based systems. Even the best designers and architects didn’t anticipate the changes that an even short-lived, Internet application may experience (Cheng et al., 2009; Kephart & Chess, 2003; Lemos, Giese, Muller, & Shaw, 2011b). Today’s demanding distributed software applications require frequent human intervention to remain in continuous operations, and there is a desire to reduce this costly, management overhead (Salehie & Tahvildari, 2009). Managing software systems that must not fail only compounds the complexity problem, requiring redundant layers of application and middleware infrastructure (Salehie & Tahvildari, 2009). The need to address software complexity was the primary problem that spawned self-adaptive and autonomic research (Cheng et al., 2009; Kephart & Chess, 2003).

Self-adaptive software offered the opportunity to lessen the impact of complexity by allowing the feedback loop to accomplish labor-intensive tasks such as application configuration, tuning, repair and upgrade (Salehie & Tahvildari, 2009). In the Internet applications domain, commercial-grade, self-adaptive applications remain unavailable, and the lack of trust in such technologies has been cited as a predominant reason (Dahm, 2010; Lemos et al., 2011b; Tamura et al., 2012); RV&V offers a method by which self-adaptive Internet applications can verify and validate their behavior; however, examples of the type of RV&V solutions defined by the Tamura et al. (2012) framework were also lacking. For SAS to be considered as a candidate solution for the problem of system complexity, a less human-intensive approach to V&V must be found. A demonstration of RV&V concepts for SAS provided an opportunity to achieve higher levels of trust and speed their introduction into the Internet applications marketplace. The RV&V concept of consistency was just one facet by which SAS was measured if it is to be part of the solution space for to reduce software complexity.

A complementary technology to SAS is Cloud Computing ("Amazon web services," 2013; Creeger, 2009). Many of the self-configuring and self-healing concepts in SAS are possible with adoption of any of the Cloud services. The focus of Cloud Computing is the infrastructure of an application, and the ability for the infrastructure to meet the changing needs of Internet applications. SAS defined this surrounding infrastructure as the external context, but SAS also delved into the internal context of the application. This focus on internal context was not addressed by Cloud Computing, except for specific application services, such as authentication, authorization, or storage. With the demonstration that the self-adaptive feedback loop is trustworthy through RV&V, SAS and Cloud Computing are clearly supporting technologies. Cloud Computing does not, however, address application complexity nor provide a method to build trust in SAS.

The work of Tamura et al. (2012) established a framework by which RV&V can be added to the self-adaptive feedback loop, and the Villegas et al. (2011) SOA example provided the basic building blocks for monitoring a web application with input from a semantic definition language. Villegas et al. (2011) did not, however, provide for monitoring of application states or state transitions. Calinescu et al. (2012) did maintain states as part of their solution, and adopts a form of PTL to transition from one execution unit to another. The language approach proposed by Tamura et al. (2012) and implemented by Villegas et al. (2011) was missing in Calinescu et al. (2012). Qureshi et al. (2012) also proposed a semantic language for SAS that was goal-based and provided monitoring extensions for quality constraints. The Qureshi et al. (2012) approach did not provide syntax or semantics to define what to do when a quality constraint is violated. All of these implementations provided building blocks by which as SAS RV&V language can be constructed, but each fell short of a complete implementation.

This research provided a syntax and semantics for RV&V that was an expansion on previous SAS modeling languages that only define runtime goals and context. By starting with the semantic context structure in Villegas et al. (2011) and adding measurement components from Qureshi et al. (2012) and Fu et al. (2007), temporal logic between states from Calinescu et al. (2012), and RV&V actions, the self-adaptive feedback loop was calibrated by the RV&V loop. Implementation of a functioning RV&V loop, along with a runtime definition of RV&V properties, provided the basis by which trust can be established in SAS through improved consistency. The temporal logic concepts of next φ, φ until φ, always φ, and eventually φ provided the basis for implementing state transitions with a rollback capability. This implementation was novel in SAS literature and demonstrates a multilayered RV&V approach. By implementing the RV&V as a second feedback loop, the adaption settings of the primary loop were calibrated at runtime. These two components of the proposed RV&V approach extend the SAS literature.

The concept of periodic measurement of viability zones was discussed by Tamura et al. (2012) as a desired component of future RV&V for SAS. The RV&V loop, defined by Tamura et al. (2012), was to take an active role in returning a self-adaptive system to a stable state when a viability zone was violated. In this mode of operation, the self-adaptive loop was suspended or overridden by the second-layer RV&V loop, whose goal was to reestablish consistency through actions and the measurement of settling time. If successfully implemented, each of these concepts extends the literature and answer specific questions posed by Tamura et al. (2012). This work was relevant because it integrated previous work into a semantic RV&V language and provided a demonstration of improved goal-oriented performance of the self-adaptive feedback loop. This work was significant because the implementation of self-adaptive RV&V has been proposed a method by which SAS methods can become trustworthy.

Barriers and Issues

SAS has received ample research attention, with over 75 papers submitted on the topic from 2000 to 2011 in 13 journals or conference proceedings (Weyns et al., 2012); however, solutions for SAS RV&V were not demonstrated, and multiple authors have called for SAS RV&V examples (Tamura et al., 2012; Weyns et al., 2012). Few, if any, examples existed that allowed for the distinction between adaption and runtime verification and validation. The call for SAS RV&V was really a request for a second-level of adaption based on higher-level quality goals, but the majority of the SAS community continues to focus on improvements to the primary feedback loop (Weyns et al., 2012). The ability to demonstrate a SAS RV&V system that operates independently from the primary feedback loop was an existing limitation in the literature.

Multiple formal methods have been applied to the primary feedback loop, including automata, Markov models, and Petri nets (Weyns et al., 2012). None of this exploration has been accomplished for the second-level, RV&V feedback loop. A second-level feedback loop required the implementation of a different decision algorithm from the primary adaption loop to make goal determination independently from state transition adaptation decisions. The reason for this separation was that the RV&V loop sought to avoid intervention in the primary loop but to tune the primary loop over time. An open issue in SAS research was the determination of which class of algorithms should be used for the deciding component of the RV&V loop.

A demonstration of SAS RV&V required the development of a representation method that was descriptive enough to define the requirements for the basic application, the adaption subsystem, and RV&V quality goals. Current representation methods either focus on monitoring a single executable for performance (Villegas et al., 2011), or integrating goals into the primary logic of the application (Qureshi et al., 2012). Representation methods were available that enable adaption, but none that allow for higher-level, quality goal enforcement. SAS representation methods have emerged from two distinct research domains: runtime requirements (Qureshi et al., 2012; Sawyer et al., 2010) and web application monitoring (Tamura et al., 2012; Villegas et al., 2011). These two domains provided basic ideas on which RV&V can be represented, but no demonstration of integrated RV&V has been accomplished. Quality goal enforcement, like that described in Fu et al. (2007), was a distinct concept from goal-based requirements management, and the two concepts have not been integrated.

The recurring call for formal methods to demonstrate provable SAS (Salehie & Tahvildari, 2009; Weyns et al., 2012) first required the use of a formal syntax for state transitions, such as LTL. The use of a temporal logic as this formal syntax provided concrete transition rules for the adaption system (Calinescu et al., 2012) and informed the RV&V loop when to act; however, only Calinescu et al. (2012) attempted to integrate the temporal logic into their SAS research. The Calinescu et al. (2012) example did not demonstrate the triggers that a formal logic must provide to the RV&V subsystem, and this should be addressed in any demonstration of a SAS RV&V capability.

The lack of generally available reference or baseline examples for SAS and SAS RV&V also limited expansion of the domain because research was not additive or supportive (Weyns et al., 2012). A problem must be developed that was accessible by the research community, and clearly presented the opportunity for RV&V management independent of self-adaption. The lack of source code for baseline examples was also described by Weyns et al. (2012) as a known problem in this domain. A baseline RV&V example presents the opportunity to validate some general quality constraints, such as minimizing operational cost or improving the mean time between failure (MTBF) and not only verify functional behavior of monitored system (Fu et al., 2007; Tamura et al., 2012). The Tamura et al. (2012) description of a self-adaptive system provided an example, but no implementation was provided in their research or referenced publications.

For RV&V to provide a method by which self-adaptive systems can gain the trust of their users, the previous barriers and issues must be addressed. A second level of intervention must be demonstrated that provides oversight of both the monitored system and the feedback loop. The representation of the system must integrate system requirements, self-adaptive states, and actions with overarching RV&V goals in the form of standard quality measures. A formal temporal logic should apply both to the self-adaptive system as well as the RV&V system, and the research should demonstrate this mutual utilization of the syntax. Lastly, the lack of baseline reference examples for SAS RV&V prevents research in this domain from being well integrated (Weyns et al., 2012).

The proposed research was novel, not because it introduces entirely new thought, but because it integrates a number of existing research ideas into a working SAS RV&V

platform. This platform was used to measure a SAS baseline example to show that a second-level RV&V feedback loop can allow SAS designers to better define and achieve overarching quality goals. These ideas included the introduction of temporal logic for state transition, integration of RV&V goals and temporal states into an SAS representation language, and the inclusion of a QoS taxonomy into the semantics of the SAS RV&V language. The integration of these ideas was novel and has not been demonstrated by previous literature.

Assumptions, Limitations, and Delimitations

It was assumed that a simulation could be developed that represents the Tamura et al. (2012) benchmark example in an Amazon Web Services (2013) server instance using timing data derived from actual measurements. It was also assumed that the measurement of time values and cost savings on small cloud-server instances could be extrapolated to make substantive conclusions on larger scale applications. Lastly, it was assumed that the demonstration of some temporal logic properties was sufficient to show that the entire set were valuable for SAS RV&V. This research was delimited to specific test cases that demonstrated transition points in the state machine of Tamura et al. (2012) example. Thus, the experimentation did not attempt to run an entire year as was described in the Tamura et al. (2012) paper, but only specific time periods where state transitions were expected.

Definition of terms

Consistency – a self-adaptive property where the adaptable system and environment are maintained to the system's conceptual model

External context – the state of the overall computing environment external to the target application.

Internal context – the target applications system state.

MAPE-K – An early self-adaptive framework that has the following components: Monitoring, Analyzing, Planning, and Executing with a Knowledge base.

Settling time – the time that it takes to transition from one stable state to another.

SLA – a Service Level Agreement is a quantified expectation of performance between a service and its customer that may encompass multiple individual measurements.

SLO – a Service Level Objective is an atomic of an SLA that can be measured by a single monitoring condition.

Viability zone - the set of valid system states, context attributes, and their corresponding values that define uncompromised system performance

Summary

Self-adaptive systems have garnering substantial interest from the research community, but were not been widely adopted because there remains a lack of trust in adaption decisions (Tamura et al., 2012). SAS RV&V and supporting standards have also been requested in the literature (de la Iglesia & Weyns, 2013; Tamura et al., 2012; Weyns et al., 2012), but examples of these standards for any baseline problem were not yet available. This research proposed to expand on the taxonomy provided by Villegas et al. (2011) by incorporating application states with temporal logic transitions, prioritized QoS goals, and additional entities necessary to document and then measure whether the introduction of a second-level feedback loop can better achieve system goals. The Tamura et al. (2012) benchmark example was employed with the primary self-adaptive feedback loop, but without RV&V interaction. Then, the second RV&V feedback loop was enabled, and results compared to determine if the addition of an RV&V capability measurably improves overall system QoS.

Chapter 2

Review of the Literature

Introduction

This section provided a synopsis of the literature for SAS RV&V expansion, and described in greater detail the reasons why an integrated model approach was needed for SAS RV&V (Weyns et al., 2012). The previous chapter provided ample evidence to justify the need for SAS RV&V, and highlighted much of the literature for this work. This chapter expanded upon specific concepts and reinforced them with appropriate literature support. First, the Tamura et al. (2012) reference application example was explained and expanded as the basis for this work. The use of state machines, or automata, was then explored as a method to manage the complexity of a self-adaptive system that maintained multiple goals (de la Iglesia & Weyns, 2013). State machines alone did not provide a sufficient RV&V solution, but a method for linking system states using temporal logic was derived from the literature to address this need (Baier & Katoen, 2008; Merz, 2001). This chapter also explained the emergence of SAS requirements languages and their integration into the feedback loop. Lastly, the Villegas et al. (2011) ontology structure was explored to discover where a more expressive language is needed to represent a general-purpose RV&V solution.

Despite the attention given to SAS research since the Kephart and Chess (2003) autonomic software challenge, the call for an integrated, runtime approach to V&V of self-adaptive systems was a recent concept (Cheng et al., 2009; Lemos et al., 2011a; Salehie & Tahvildari, 2009; Tamura et al., 2012). Few SAS RV&V examples existed, and their demonstration was difficult to discern from the basic SAS feedback structure (Calinescu et al., 2012). The vast majority of current SAS research was disconnected, and few SAS benchmarks or standards were available for expansion (Weyns et al., 2012). Because these standards were lacking, each research solution stood alone and did not provide for stepwise improvement. Weyns et al. (2012) specifically called for original research, versus reusing previous research in the new context of SAS (Calinescu et al., 2012). The need for a standards-based RV&V approach has been established, but a cumulative lineage of SAS RV&V research was not established. The Tamura et al. (2012) case study and its supporting application example provided the first documented benchmark on which this, and future, research built. Even the more recent work of de la Iglesia and Weyns (2013) did not provide RV&V of the runtime system. Rather, it provided formal verification of the processes within the feedback loop.

The following topics were selected as the basis for the proposed work as their concepts support SAS additions for a generic RV&V capability. The Tamura et al. (2012) SAS RV&V case study provided expansive coverage of how general RV&V concepts should be applied to SAS. This work also lays out many research challenges for the community in the area of SAS RV&V. One of the research areas discussed in Tamura et al. (2012) is that of state machines that defined where the system was operating in a stable condition or codified unstable areas where RV&V interaction was required. Thus, the use of state machines as an RV&V zone identification mechanism required further expansion. Recent work by de la Iglesia and Weyns (2013) also focused on the use of state machines in a formally verified feedback loop.

Other areas of computer science, such as autonomous control systems, also utilized state machines for RV&V; however, these systems almost always utilized some form of temporal logic to connect the states (Baier & Katoen, 2008; Barringer et al., 2009). Thus, temporal logic additions to state machines were also explored. The MAPE-K feedback loop formed the basis for many of the SAS efforts to date (Calinescu et al., 2012; Cheng et al., 2009; de la Iglesia & Weyns, 2013; Lemos et al., 2011a; Salehie & Tahvildari, 2009; Tamura et al., 2012); however, no standards have emerged to define how knowledge is to be introduced into the MAPE-K loop, or maintained during feedback loop operation. The desire to introduce knowledge into a machine-readable feedback loop was a topic that was closely linked to SAS requirements languages and goal-based ontologies. Each of these areas were explored in this section.

Problem Overview

The Tamura et al. (2012) RV&V case study provided the general description for a self-adaptive, cloud-based e-commerce purchasing platform. Their example provided a real-world problem for which SAS RV&V solutions may be demonstrated. The Tamura et al. (2012) example was generic in nature, and did not provide implementation detail; however, it was the first cited example of an application where RV&V requirements were delineated for a SAS application. The example did not provide any detail about the cloud-based component of the system, nor an architecture on which the requirements were to be implemented. These details were left for experimentation.

In the proposed WPO system, users browsed a list of products and placed orders. The problem defined multiple customer types, and thus the system required customer account functionality as well. A user required an account to purchase a product, but was able to browse the product list without an account. The four major use cases for this system were: create a user account, browse a product list, login, and purchase a product. Adaption occurred to maintain non-functional performance requirements in the form of the settings for a metric entitled Processing Purchase Orders (PPO). Tamura et al. (2012) defined this overarching metric as the number of transactions per unit time; however, it was most easily measured in the example application as the inverse, or unit time per transaction. An average over a period, such as 10 minutes, achieved the same PPO concept. Two classes of customers existed in the WPO example: Regular customers and Preferred customers. Preferred customers were to be serviced 10% faster than Regular customers, and thus an additional PPO metric was created for Preferred customers (PPPO).

The system requirements also defined three configuration zones based on the two PPO metrics. The first zone was an infrastructure configuration representing a Regular response time for all customers. The second zone required that the system operate at Medium capacity when special offers were placed on social networks or sale days were scheduled using a system calendar. The last configuration established a cloud-based infrastructure capable of dealing with the highest peak load defined by atypical shopping days, like Black Friday. The RV&V system tuned the SAS metrics to optimize system performance over time.

The non-functional requirements of the system were defined by the categories of QoS Throughput adherence and Cost reduction (Fu et al., 2007). As server load increased, the system had to determine when to activate new server instances in order to transition from one configuration to another. The time between the decision to expand or subtract the number of server instances - and their availability - constituted the settling time of the system. A longer settling time provided a direct correlation to increased cost in a cloud-server environment. As the response time metric falls below the current zone set point, the system must also decide when to idle servers until all customers have exited, shut the server down, and assume a lesser zone configuration. The concept of settling time applied to the time that a change in configuration decision was made until the new configuration was attained.

As Weyns et al. (2012) note, few benchmark examples of SAS RV&V systems existed, and thus this Tamura et al. (2012) application example provided an important reference for this and future SAS RV&V work. This example formed the basic structure by which experimentation was performed. The remainder of this chapter described research areas that contribute to or were used to expand this work. Each area was incorporated in some way into the Tamura et al. (2012) example to provide an expansion of the SAS RV&V topic.

State machines

State machines were highlighted in the literature as a common tool used to document self-adaptive behavior, but few state machine examples supported RV&V aspects of SAS (Calinescu et al., 2012; de la Iglesia & Weyns, 2013; Tamura et al., 2012). In the context of SAS, states were named, logical abstractions that documented a point in the system where input criteria have been satisfied, but output transition criteria remain unsatisfied. Sipser (2006) provided a reference for classic state machine behavior, and his definition was adapted to the context of SAS. The formal definition of a state machine, or finite automata from computational theory, was a 5-tuple (Q, Σ, δ, q_0, F), where Q was a finite set of states, Σ was a finite set of input values called the alphabet, δ: $Q \times \Sigma \rightarrow Q$ was the transition function, $q_0 \in Q$ was the start state, and $F \subseteq Q$ was the set of accept or final states (Sipser, 2006). In the context SAS RV&V, there were no final or accept states, as the RV&V platform was designed for non-stop operation. Lower level state machines, such as those defined in de la Iglesia and Weyns (2013), may terminate and the state documentation mechanism should support both types.

Figure 8 showed a state machine example of recent SAS work by de la Iglesia and Weyns (2013) where they implemented each component of the MAPE-K primary feedback

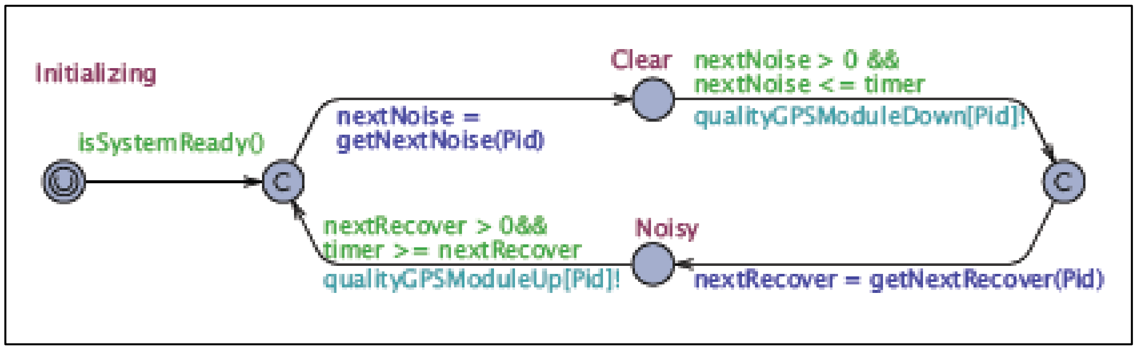


Figure 8. MAPE-K automata example (de la Iglesia & Weyns, 2013)

loop with formally verified automata. These automata utilized the Uppaal state transition syntax, but did not allow for the modification of the state monitoring conditions based on real-world feedback as SAS RV&V requires (Behrmann, David, & Larsen, 2006). As can be seen in Figure 8, the de la Iglesia and Weyns (2013) feedback loops were implemented with a logic language surrounding software methods. This approach was similar to that of Villegas et al. (2011) who evaluated the results of a single MonitoringCondition from an ExecutableCodeUnit, shown in Figure 4. The Uppaal approach did recognize the need for evaluating temporal constraints within the state machine, but forced these constraints to be pre-defined before the execution of the self-adaptive system.

This research also acknowledged that monitoring conditions were formed using logic that surrounds the output of target application methods. As well, time was a critical factor needed to evaluate self-adaptive decisions. The Uppaal approach required that scalar values be assigned at design time (de la Iglesia & Weyns, 2013). In contrast, this research used the more general LTL syntax, which allows the RV&V loop to assign concrete values at runtime (Baier & Katoen, 2008). While Uppaal was a documented syntax, it was not widely used outside of research projects, and Uppaal was really a research tool for model checking. This research demonstrated that RV&V can affirm the actions of the transition function in the context of QoS goals, and not only provide further evidence that state machines or automata are valuable in the context of self-adaption. The RV&V system not only affirmed the actions of the transition function, but also acted to modify the surrounding monitoring condition when self-adaptive behavior had to be improved.

The de la Iglesia and Weyns (2013) work focused on the ability to formally verify SAS state transition decisions within each component of the MAPE-K loop using not just the Uppaal monitoring syntax, but the entire suite of Uppaal tools (Baier & Katoen, 2008; Behrmann et al., 2006). Uppaal provided a timed automata language that implemented hard temporal constraints on state transitions and a separate constraint language for model checking. Once a state machine was defined and constraints are applied, the system was checked in the Uppaal application environment. The desire for formally verified SAS transition systems was identified in Weyns et al. (2012) and satisfied in de la Iglesia and Weyns (2013). These works showed both the need for V&V of SAS transitions as well as the need for oversight of the self-adaptive loop actions. While the earlier work by Weyns et al. (2012) focused on the need for formal methods in RV&V, the recent work did not show the true benefit of having a formal model checker as part of the solution. This research focused less on formal verification of the logic in the primary feedback loop and more on the integration of RV&V surrounding the target application and the primary feedback loop to improve overall system goals using QoS measures.

Tamura et al. (2012) proposed a benchmark example as a simple, three state non-terminating automata shown in Figure 9. This simple presentation was consistent with the automata structure required by Baier & Katoen (2008) for model checking except that they applied LTL logic. Figure 9 showed the Tamura et al. (2012) example with the addition of LTL logic in the state transition arrows. These states defined a higher level of abstraction than the de la Iglesia and Weyns (2013) approach from Figure 8, which modeled each MAPE-K feedback loop component as a separate state machine. This higher level of abstraction highlighted the differences between self-adaptation and RV&V. The de la Iglesia and Weyns (2013) work defined each possible step of the system in the state machine syntax. Tamura et al. (2012) defined only general parameters for each state, but they then included a separate hierarchy of system goals, ordered as (1) non-stop operation, (2) customer satisfaction, and (3) cost minimization.

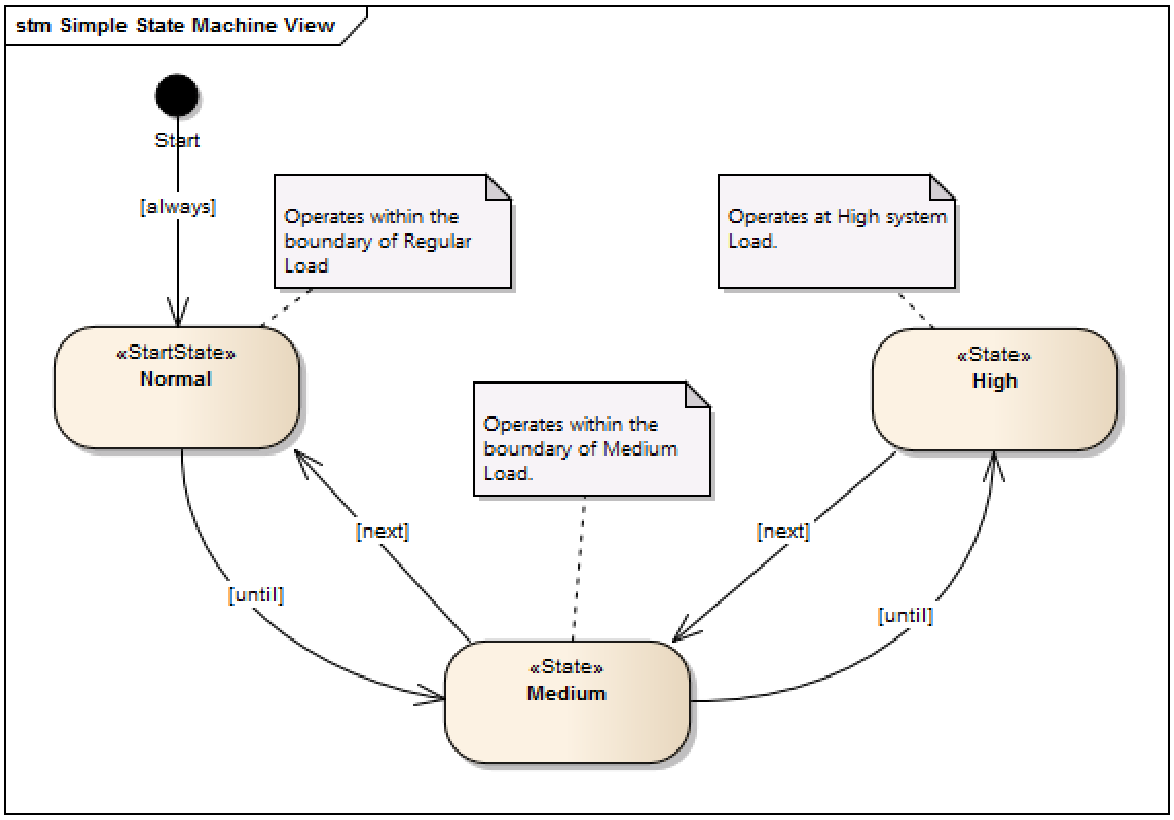

stm Simple State Machine View
Start
[always]
Operates within the boundary of Regular Load
Operates at High system Load.
«StartState»
Normal
«State»
High
Operates within the boundary of Medium Load.
[next]
[next]
[until]
[until]
«State»
Medium

Figure 9. Simple state machine view

The state entitled “Normal”, in the Tamura et al. (2012) example, was described by being able to accept a “Regular Load” of customers on the purchase order web application. The actual value that identified a “Regular Load” was not specified, and may be optimized by the RV&V system over time based on the capacity of available hardware. A SAS system will naturally transition states over time, and this behavior may be increased by RV&V intervention or decreased by RV&V tuning (Tamura et al., 2012). This research demonstrated both RV&V intervention and tuning of the Tamura et al. (2012) SAS benchmark.

The idea of an infinitely running system was also unique from the temporal logic approach of Calinescu et al. (2012) shown in Figure 6. Calinescu et al. (2012) used a state machine to determine the next best state transition among multiple options using Markov's Random Walk algorithm. In contrast, the Tamura et al. (2012) RV&V benchmark example did not use probabilities to determine the next successful state transition, but gathered runtime data to evaluate whether the self-adaption loop is making the right choice to stay within a Viability Zone. While the use of state machines for SAS seemed to be approaching a standard, their employment in the context of RV&V required a more general treatment so that multiple SAS implementations can fit within a single RV&V specification.

Temporal Logic

Temporal logic, and more specifically LTL, abstracted the definition of time-based behavior in a state transition system by removing the exact definition of time; however, LTL did implement a logical, temporal progression (Baier & Katoen, 2008; Barringer et al., 2009; Goldberg et al., 2005; Heimdahl & Leveson, 1996). This differed from the Uppaal approach employed by de la Iglesia and Weyns (2013) in that their transition logic required exacting time measurements (Baier & Katoen, 2008; Behrmann et al., 2006). By using the aforementioned LTL, an infinite transition system could be represented by the following statement, [*always* φ *until* π] (where φ represents the Normal state, and π represents the Medium state), using the Tamura et al. (2012) example. The time that it takes to achieve π was not defined in the state transition. Instead, a hierarchical, goal-based system attempted to minimize this transition time as it equates to lost computing cycles and thus increased cost in a cloud-based environment. Extending this example, the Medium state may transition to the High state, or go back to the Normal state; however, the time to achieve the Normal state would be immediate, using the following logic, [*always* {π (*until* λ) | (*next* φ)}] (where φ

represented the Normal state, π represented Medium state, and λ the High state). This Medium to Normal state transition was proposed to be observed at the very next measurement period. The exploration of temporal logic and a hierarchy of goals was a key component of exploration in this research.

Using the previous LTL example, the statement [*always {π (until* λ) | (*next* φ)}] was easily converted into a graph-based statement shown in Figure 9. The start state of an automata implies the *always* statement, and a transition to the Normal state is accomplished as soon as startup conditions are achieved. The Normal to Medium state transition was the only available transition and was not represented by this LTL statement. From the Medium state π, a branch occurred, and temporal logic allowed for a delineation of which branch is taken. The φ branch back to the Normal state was measured at the next measurement cycle, but the λ branch takes many measurement cycles to be achieved. Until the High state λ was achieved, the RV&V system gathered statistics on this cumulative time so that future decisions were better informed. By using the LTL approach, the flexibility that Tamura et al. (2012) required of the RV&V loop was provided to the SAS loop. These concepts provided the basis for the graph-based state machine that this research evaluated.

Goal-based SAS ontologies

Goal-based approaches that include state machines also emerged as a mechanism to document self-adaptive behavior in a human-readable form (Qureshi et al., 2011, 2012). The latest of these ontologies from Qureshi et al. (2012), shown in Figure 7, was not suitable for an RV&V loop to directly ingest, perform monitoring functions, or institute actions when the self-adaptive feedback loop failed. The Qureshi et al. (2012) approach was, however, among the first to link goals to states, but did not mention RV&V as a consumer of the ontology.

The Villegas et al. (2011) approach documented a monitoring system in human-readable form that was also machine-readable using an RDF-based ontology. The ability to provide a requirements documentation trail that the RV&V system can directly ingest and update at runtime was discussed in Tamura et al. (2012). The Villegas et al. (2011) SmartContext identified code units by a Uniform Resource Locator (URL), and this mechanism was extended to include the monitoring points where the RV&V system measures adaptive system performance. As well, the system exposed action points where the RV&V system can inject commands into the feedback loop when a violation occurs. No current literature documented these action points in human readable or URL form. The URL format was used in this research to provide unique name for both the monitoring points and the action points in the simulation.

Since a documentation system should include the capability to define the self-adaptive state model as well was the RV&V specification, the boundary between adaption and RV&V can be easily blurred (Calinescu et al., 2012). The self-adaption system should be able to operate unimpeded by the RV&V platform, when operating within a viability zone (Tamura et al., 2012), but the role and the decision process for an RV&V system required further exploration. Previous self-adaption research used off-the-shelf AI-planning systems to make adaption decisions (Arshad et al., 2004), or made state transition choices using Markov decision processes (Calinescu et al., 2012). The RV&V system needed a different form of decision approach from the one in the primary feedback loop, and one that imposed its will only when a fundamental goal is violated. With multiple actions available, the RV&V decision process used a simpler decision algorithm than that of the primary decider. Multiple options were available to the RV&V platform, such as Naïve Bayes, a simple Markov-chain, or a learning algorithm based only on system goals and input probabilities from previous measurements (Mitchell, 1997). While the focus of this research was not the selection of an optimal decision algorithm, an simple rule-based, RV&V decision algorithm was selected.

SAS Requirements Languages

Requirements-driven design techniques were introduced prior to the focus on SAS in the domain of autonomic computing, and these were easily applied to SAS RV&V (Lamsweerde, 2000; Lapouchnian, Yu, Liaskos, & Mylopoulos, 2005; Sawyer et al., 2010; Welsh, Sawyer, & Bencomo, 2011). Lapouchnian et al. (2005) defined a semantics for a goal-oriented requirements model where the system takes action to achieve requirements goals. This semantics created a hierarchy of connected goals with actions that *help(+), hurt (-), make(++)* or *break (--)* the connected goal as shown in Figure 10. The work of Lapouchnian et al. (2005) also described simple statistical techniques that might be used to

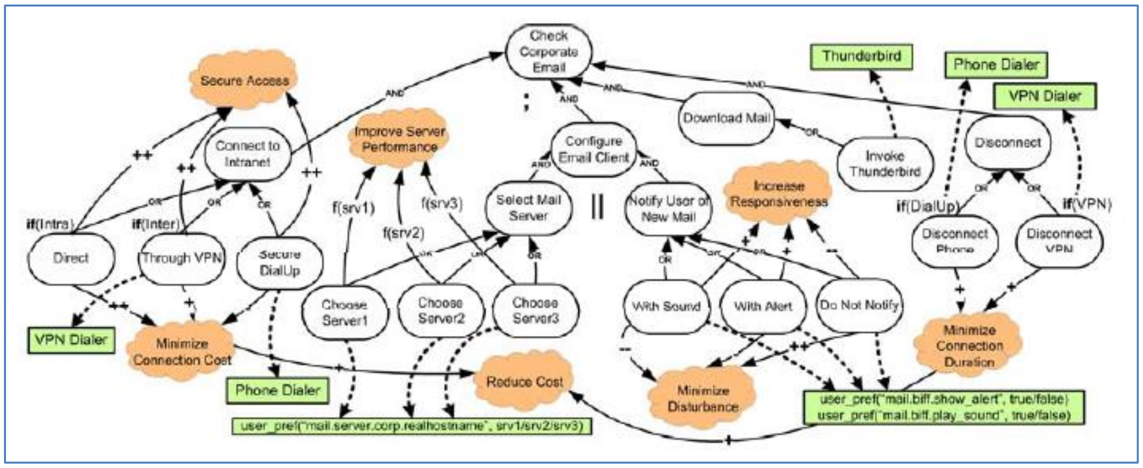


Figure 10. Help, hurt, make, or break requirements diagram (Lapouchnian et al.,2005)

select more important goals from lesser ones when adaptation decisions must make tradeoffs. An RV&V modeling language and system must also have the ability to represent multiple goals, and rank them independently from states. Using the Tamura et al. (2012) example problem, the goal hierarchy was defined as minimizing customer delay to preferred customers as a primary goal, minimizing regular customers' wait times as a secondary goal, and minimizing operational costs as a tertiary goal. The ability to integrate an RV&V goal hierarchy with a self-adaptive state model was a required component of a machine-readable SAS RV&V modeling language (Calinescu et al., 2012; Qureshi et al., 2012; Tamura et al., 2012). The determination of how to traverse the hierarchical ranking of system goals was an area for exploration during this research.

While the Qureshi et al. (2012) model shown in Figure 7 did not utilize temporal logic, it did propose an infinite state transition model. Qureshi et al. (2012) decorated individual states with quality goals, but the entire system was not provided with a hierarchy of goals.

Individual states had QoS goals, but the overall system also required a set of quality goals that should hold independent of the current or future states. The QoS goal structure in Fu et al. (2007) provided a reference point by which named instances of QoS attributes can be injected into the ontology. The combination of the Qureshi et al. (2012) linkage of an ontology with goals, the Villegas et al. (2011) RDF utilization of URLs to link metrics to objectives, and the Fu et al. (2007) QoS hierarchy provided a complete structure to describe RV&V monitoring.

Summary

The literature referenced in this section supported further research in the following areas of SAS RV&V. The call for a standard, lightweight RV&V capability demonstration was necessary to establish a baseline for this topic, and allows future community expansion (Weyns et al., 2012). To achieve RV&V independent from self-adaption, a hierarchy of goals must be complementary to the state transition logic of the self-adaptive system. This goal hierarchy must have a decision sub-system that is also independent of the self-adaptive feedback loop, but was not invasive during normal operation (Tamura et al., 2012). State transition logic required a syntax that allowed the RV&V platform to improve the settings for state management over time, and the temporal logic approach allowed for this tuning of time-based parameters. The RV&V system demonstrated a syntax and semantics that is readable by both humans and the adaption system using a combination of the Qureshi et al. (2012) goal approach and the Villegas et al. (2011) RDF approach. The combination of these research ideas determined if an RV&V system adds measurable value to the SAS feedback loop.

Chapter 3

Methodology

Overview

This research expanded upon the Villegas et al. (2011) SLA governance structure by implementing RV&V goals and states in a self-adaptive application, and then measuring both application throughput and cloud-based costs to determine if improved QoS performance were achieved. The works of Weyns et al. (2012), Calinescu et al. (2012), and Tamura et al. (2012) each called for the introduction of a self-adaptive RV&V capability, but specific methods were not introduced. The Tamura et al. (2012) web-purchase order (WPO) target application was implemented as the primary research platform in a cloud-based environment, and this environment was used throughout the research.

In the first phase of experimentation, a single server in the WPO cloud-based platform was loaded with client transactions to determine the number of web clients required to saturate a single-server. As well, set points for normal and preferred customer response time were determined. In the second phase of the research, a self-adaptive feedback loop was introduced to monitor the target application using the expanded ontology structure of SAS states and RV&V goals. The objective of this second phase was to determine if a second-level, RV&V feedback loop improves QoS over the primary, self-adaptive loop. Time-based test cases were run with self-adaptive-only feedback enabled, and then the same test cases rerun with SAS RV&V enabled to determine if QoS metrics are maintained while overall system costs were reduced.

This work demonstrated multiple facets of SAS RV&V. First, the addition of an RV&V feedback loop validated that the self-adaptive system was operating within its boundaries. Secondly, the RV&V feedback loop used quality goal feedback to improve the performance of the primary, self-adaptive feedback loop by reducing operating costs yet maintaining system throughput. Third, the use of temporal logic in a state machine enabled the feedback system to recognize opportunities to limit *settling time.* Lastly, the establishment of a standard mechanism to communicate QoS goals to self-adaptive applications allowed for the introduction of different RV&V approaches to be evaluated quickly against a consistent problem space.

This chapter provided a detailed overview of the experimental system that must be developed, a description of the ontology expansion, a detailed description of each test case, the complete experimentation approach, the analysis methods employed, and required resources. The first section of this chapter provided a description of the ontology expansion made to the Villegas et al. (2011) structure. Next, a general overview of the entire experimental system was provided, and then each component of the experimental system design was described in further detail. The detailed design discussion of the experimental system included the WPO target application, self-adaptive and RV&V feedback loops, and web load simulator that all run in a cloud-based environment. A discussion of the first phase test cases and approach used to collect baseline statistics was also provided. Finally, the full set of test cases that were used in the second phase of the experimentation was described, along with the approach for analyzing results. Resource requirements were then defined.

Representation Overview

The Villegas et al. (2011) governance taxonomy, shown again as Figure 11, was the basis on which the SAS application and the RV&V measurement approach was documented in the experimental system. SmartContext – the name of the Villegas et al. (2011) taxonomy – defined SLAs as statements of performance that were further defined as one or more SLOs measuring values attained by a service interface. The SLO was defined by a triple *(p, a, s)*, where *p* was an n-ary predicate used to evaluate a quality property, *a* was an action to take if *p* was violated, and *s* was a post-condition that defined service operation after *a* takes place (Villegas et al., 2011). Taxonomy elements were linked together via object properties into an RDF graph shown in Figure 12, and this graph provided a machine-readable method by which a governance system can measure compliance. This governance framework was not specifically developed for RV&V, but was suggested by Tamura et al. (2012) as an example of a structure that could be expanded to provide SAS RV&V. No specific SAS elements were defined in SmartContext, but much of the same contextual information was reused for self-adaption.

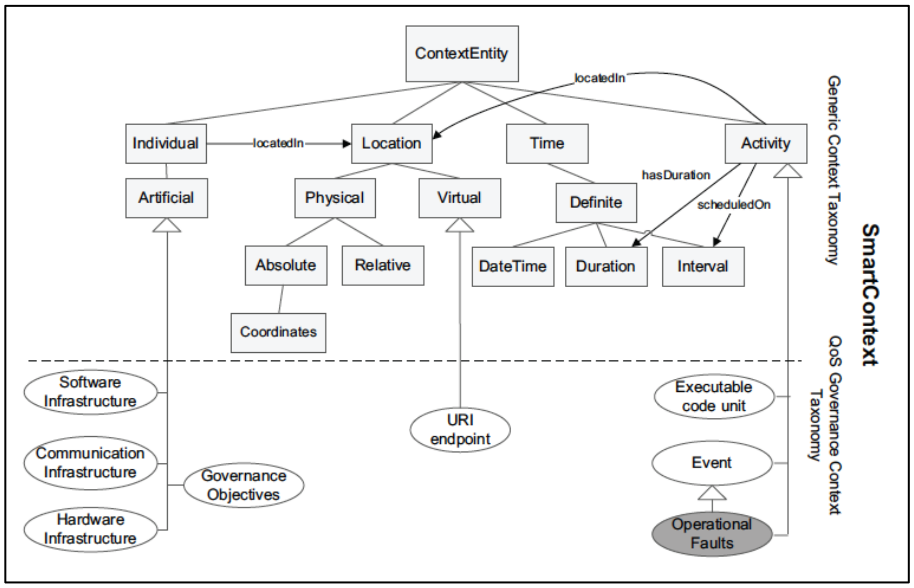


Figure 11. SmartContext (Villegas et al., 2011)

The RDF graph in Figure 12 provided a basic example on which to monitor a governance objective. The target software application was referred to as ServiceA. It had a governance guarantee in the form of a Performance SLA and a single Efficiency SLO. The SLO contained a single MonitoringCondition rule using output from a MonitoringFunction that combined the output of two ServiceA interfaces to form the TimeBehavior (TB) metric. For the experimental system it was proposed that this same structure be utilized, but expanded to provide a hierarchy of RV&V goals.

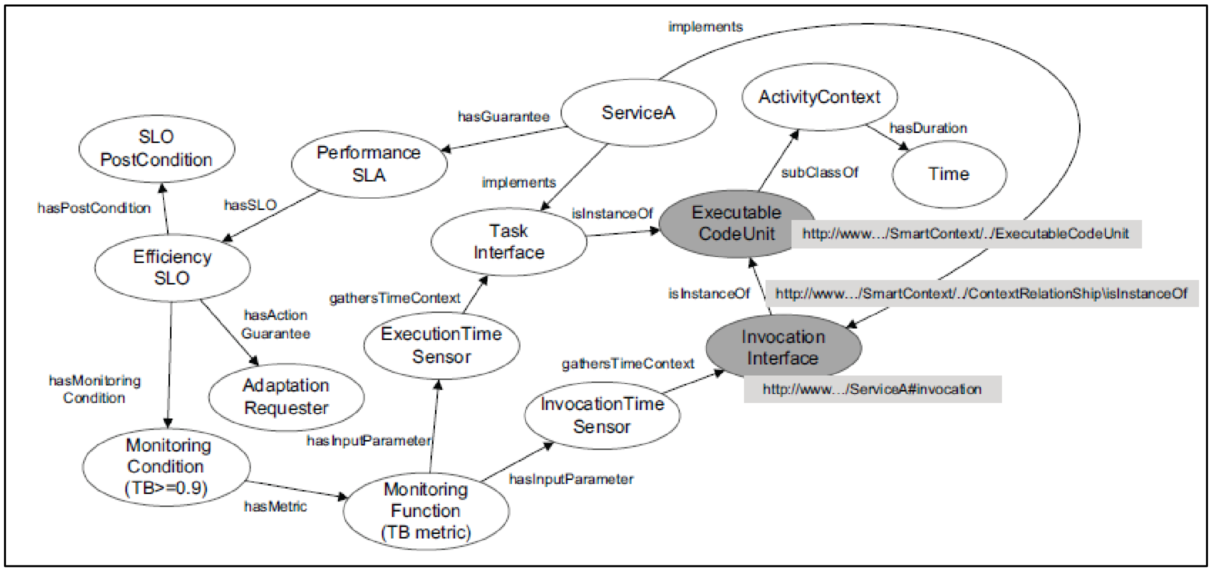


Figure 12. Service level agreement monitoring RDF structure (Villegas et al., 2011)

RDF graphs provided a mechanism to link content into machine readable form, but were not true ontologies. A true ontology approach defined the domain and range of each entity, and defined the schema by which object and data properties were attached to entities. To accomplish this, the Villegas et al. (2011) taxonomy was re-entered using the Protégé (2012) ontology editor, and each SmartContext entity was given domain and range limitations that are enforced by Protégé. The structure of SmartContext with possible SAS and RV&V expansions was stored in an ontology referred to as *R1*, and this ontology was imported into the final experimental ontology *R2*. *R1* can be thought of as the schema that restricts the relationships in which entities and properties can participate. Each ontology was developed in Protégé and saved in the OWL format. Contextual entities were stored in *R2* and included the names of Servers One through Five, their IP addresses, service names, and URI endpoints where their interfaces are exposed. Service names included the WPO, DatastoreService, Feedback, and SimpleQueueService. Specific interfaces of each service will be detailed in the following sections.

The first proposed expansion to SmartContext was the addition of a *State* and temporal logic on which the self-adaptive behavior was documented. The *StartState* had a sub-class relationship with a *State* so that it was easy selected from the ontology graph. SmartContext had a definition for Definite Time, but temporal logic required *Relative* time to be available for its entities. The *Always, Eventually, Next, and Until* entities were implemented in the *R1* schema as *Relative* entities. Each temporal logic element required an object property that linked the relative time entity to a SAS MonitoringCondition and adapted from that in Figure 12. For the experimental system *States* were uniquely identified by the server names associated with each *State*. For example, the *Normal* state had an object property attached called *isDefinedBy* that contained a link to the *HardwareInfrastructure* entity shown in Figure 11.

With the addition of state transitions and temporal context, the feedback system – whether SAS or RV&V – programmatically identified the points in the system lifecycle where *settling time* may be addressed. The *Until* state transition decorator provided a marker that the feedback system used to minimize time associated with settling between states. In a generic sense, *Until* transitions constitute a point in system behavior where the transition trigger logic was modified by the RV&V system to minimize settling time behavior. The *Next* transition also required an RV&V modification.

In the context of this experimentation, the RV&V set points were not modified by a distinct settling time goal triggered by the *Until* decoration. The QoS goals established in this experimental system naturally tended to limit settling time, and thus an additional settling time goal and associated rules did not provide statistically significant benefit. A more detailed description of *Until* behavior as it relates to settling time is addressed in the feedback loop section.

For the general case, the *Next* transition type did not specifically define a settling time opportunity. Specific rules in the SAS and RV&V Planner were required to address *Next* transitions, but these *Next* state decorations did not provide a settling time demonstration opportunity in this experimentation as the spin up and spin down times for the WPO servers were very short.

The second additions to the SmartContext structure were RV&V entities and QoS metric types. Much like SLAs, RV&V *Goals* were defined by a sub-graph of MonitoringConditions, MonitoringFunctions, and violation actions. A *Goal*, however, was system-wide, and not tied to a single service as shown in Figure 12, and a Goal was linked into a hierarchy where subordinate *Goal* actions were negated in order to maintain a higher-level *Goal*. This concept of higher-level *Goal* satisfaction was the most significant component of the changes from the SmartContext approach. A *Goal* was further defined by a type of QoS metric, such as *Performance,* or *Cost* from Fu et al. (2007). These QoS goals were also directional, and informed the system which direction to orient the system in order to best achieve the goal. Example directions included *Maximize* or *Minimize*. These directional titles were optimization suggestions, not the requirement for formal mathematical techniques. An example of the proposed WPO feedback subsystem goals were shown in Figure 13 below where the red entities exist in the ontology and the yellow entities are implied by the ontology structure. The combination of these new context entities within the existing SmartContext allowed for RV&V monitoring.

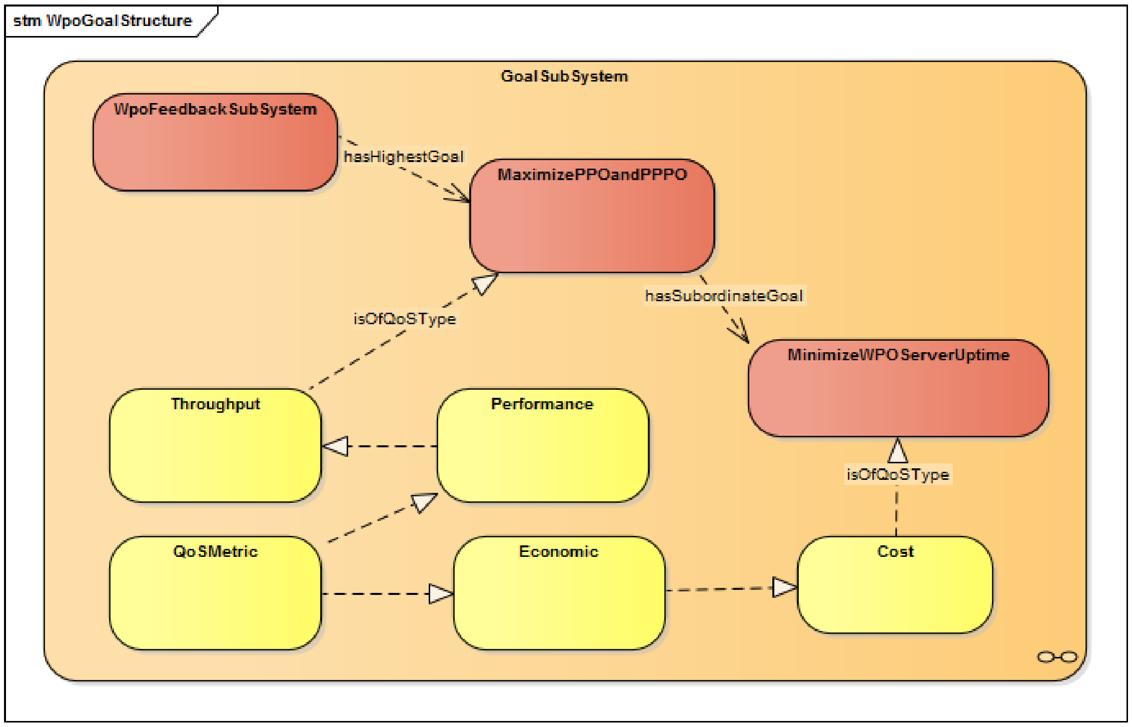
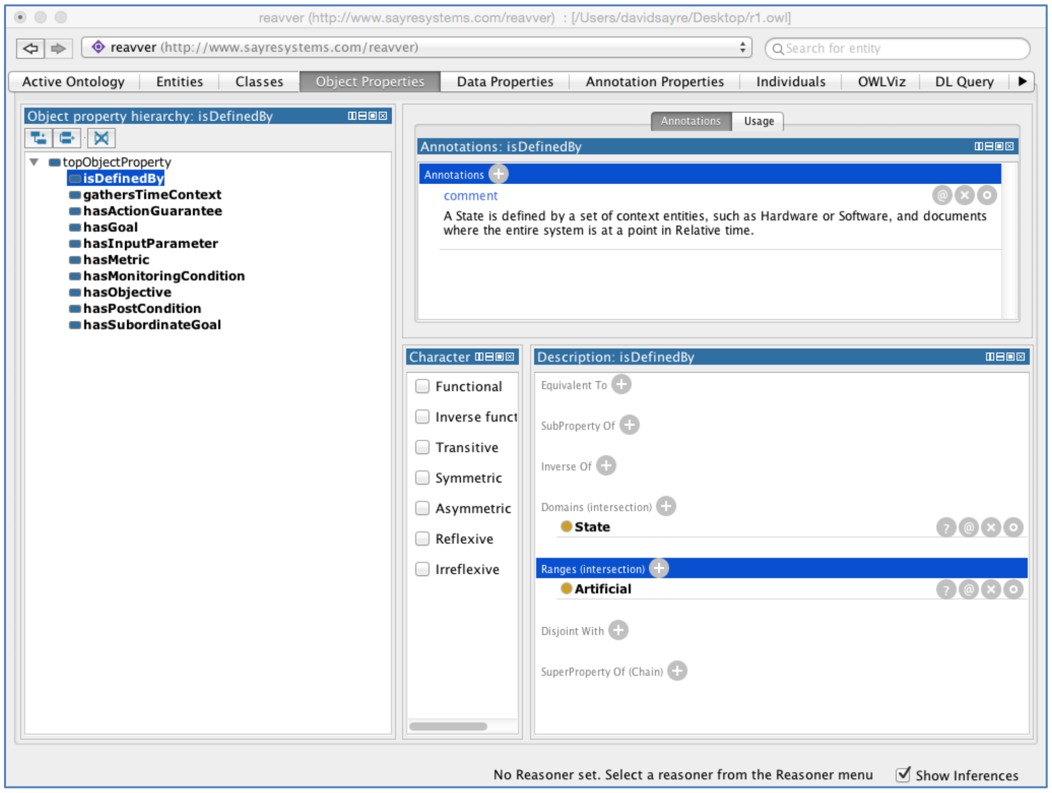


Figure 13. WPO feedback goal structure

Each entity and property in SmartContext was manually entered with domain and range values added using Protégé (2012), and an example domain and range entry screen was shown in Figure 14. Figure 15 showed the proposed entity additions to the SmartContext taxonomy in an ontology format extracted from Protégé (2012). First, *Relative* time concepts were added at the same level as that of *Definite* time. Under the *Relative* time entity each of the proposed temporal logic components were added. Next, the QoS metrics were added under the *Artificial* context entity. Under the *Artificial* context entity, *State* and *StartState* entities were added to allow for the creation of verifiable finite state machines. As well, the *Goal* and *HighestGoal* entities were added to *Artificial* and provided for the hierarchical approach described earlier. Each of these proposals was demonstrated by experimentation, but were modified slightly to meet experimentation goals. The final ontology approach was not the focus of experimentation, but can be derived from the final state machine displayed in the Results section.



comment
A State is defined by a set of context entities, such as Hardware or Software, and documents where the entire system is at a point in Relative time.

Domains (intersection)
State

Ranges (intersection)
Artificial

No Reasoner set. Select a reasoner from the Reasoner menu

Figure 14. Protégé ontology development screenshot

By using the proposed expanded taxonomy shown in Figure 15, an ontology of SAS and RV&V knowledge was developed. This data structure informed components of the SAS and RV&V feedback loops how to monitor a specific target software system – in this case the WPO research example. A sample of the ontology sub-graph that represents the state transition system was shown in Figure 16. This figure demonstrated the integration of state transitions and temporal logic to guide the system by defining the proper state to assume with changing contextual behavior. A synthetic state, called Start, was an instance of the ontology class StartState so that the Feedback application easily located it as the entry point into the state machine. The transition from Start to Normal utilized an Always decorator. Movement from the Normal state to the Medium state passed through a Until entity. This type defined that the system must continue to operate in the Normal state Until the Medium state was achieved. Using the same RDF graph approach shown in Figure 12, the state transition monitoring approach was developed using a MonitoringCondition and AdaptionRequestor to move the system to the next system state. MonitoringCondition rules for SAS and RV&V are shown in the next section. The PostCondition was that the temporal logic constraint has been satisfied, and the system was in a subsequent state per the state machine.

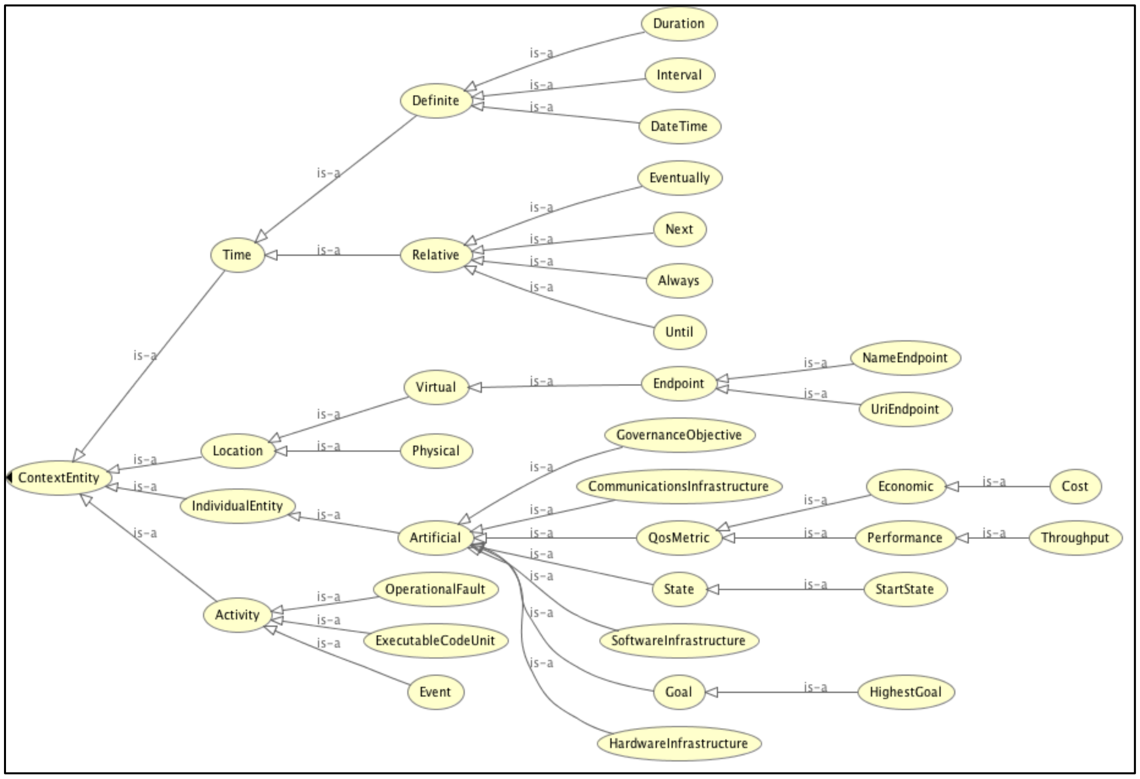


Figure 15. Possible SmartContext class hierarchy extension

The experimental representation of SAS and RV&V contextual items provided a generic mechanism by which SAS and RV&V transitions were documented and automated in the feedback loops. By using an ontology versus just an RDF graph, structure was provided to the documented system so that future inference capability can be applied in the Planner. Structures such as *States* and *Relative* time, shown in Figure 16 below, were added to the SmartContext taxonomy in order to permit the management of SAS transitions at the system

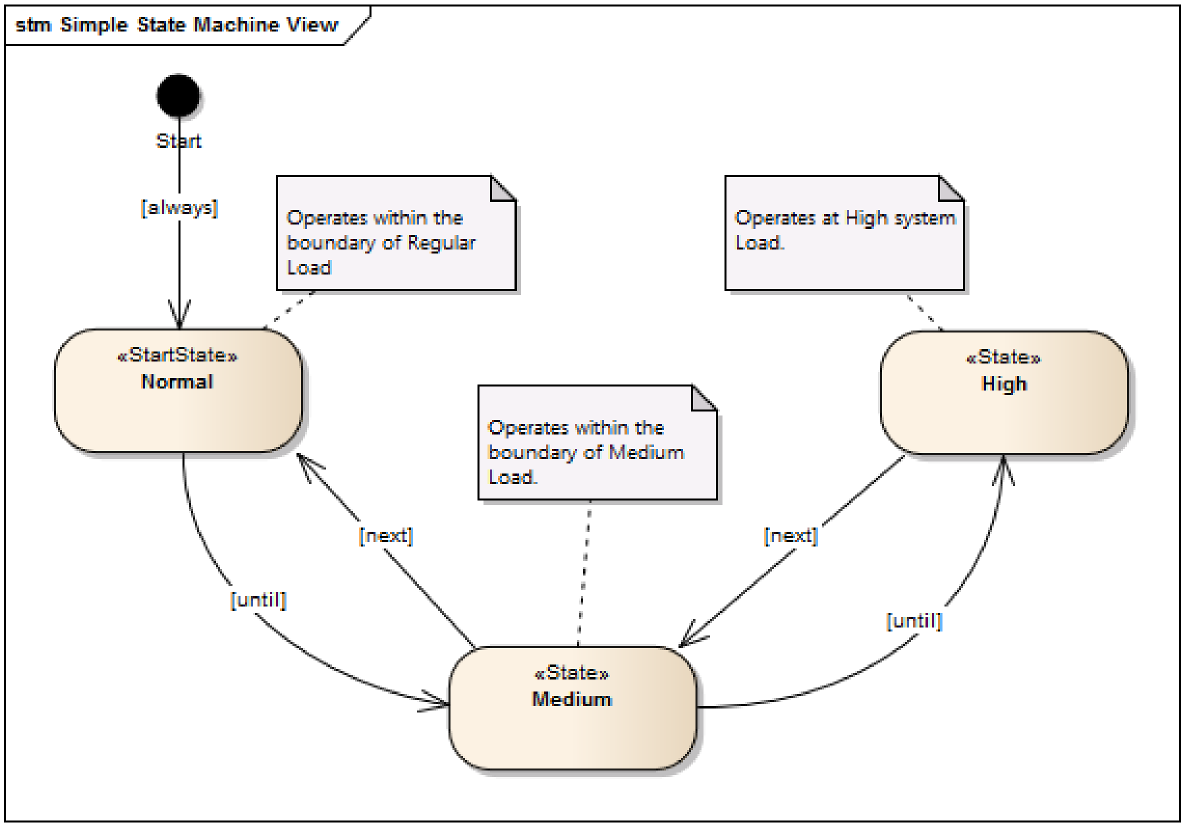

stm Simple State Machine View
Start
[always]
Operates within the boundary of Regular Load
Operates at High system Load.
«StartState» Normal
«State» High
Operates within the boundary of Medium Load.
[next]
[next]
[until]
[until]
«State» Medium

Figure 16. Proposed state transition diagram

level versus at the level of an individual software service or interface. Additional RV&V structures such as the *Goal* structure were inserted into the MonitoringCondition hierarchy to provide a mechanism to measure QoS metrics and react when a *Goal* violated. By making the *Goal* a hierarchical component, subordinate *Goal* actions were suppressed when their implementation violates higher-level goals. The ontology described here was not read into the Feedback subsystem by the Analyzer subcomponent, shown in Figure 17 as the transformation from RDF to objects was not the focus of experimentation. The next section introduced the experimental system that will utilize this representation.

Experimental System

The experimental system utilized the Amazon AWS (“Amazon web services,” 2013) cloud environment and ran on Ubuntu Linux (“Ubuntu,” 2013) images as the server infrastructure of the WPO application, feedback loops, and load simulator. A Unified Modeling Language (UML) component diagram of the Tamura et al. (2012) benchmark example and RV&V framework implementation was shown in Figure 17 (“Unified modeling language,” 2013). Tamura et al. (2012) suggested a cloud-based approach for their WPO reference problem, and AWS was the chosen platform on which the experimental system was implemented.

In this section the major components of the Figure 17 architecture are discussed. Subsequent sections break down the WPO, feedback loops, load simulator, and AWS services, and described the individual applications. Working counterclockwise from the top right of Figure 17, each major server component is now described. The WPO was a web application that may be duplicated on multiple servers in order to scale with user load. Each WPO application instance was hosted on a single server, and additional WPO servers were added to the environment as external context changes trigger server configuration changes. The web and application tiers of the WPO application were housed within a single server environment, but the database tier was separate from the WPO application. Measuring WPO transaction times under increasing load was the focus of the feedback system, and thus the scalability of the WPO application across multiple servers was a key component of the design. The WPO servers also provided a definitive architectural construct with which to demonstrate settling time behavior. When adding an additional server to the environment, the time where the added server incurs cloud server costs, but was not providing benefit to the WPO environment was an example of settling time. As well, the time when a server is being deactivated also constituted settling time behavior. The input interface to the WPO component – iWebPurchase – was a standard HTML interface. The first output interface was a queuing service to serialize the QoS metric of transaction time and number of concurrent connections per server (CC/Svr) on the WPO application. The second was a datastore interface to synchronize user and product configuration information for the website, and to store completed purchases. Figure 17 showed servers three through five as the WPO application servers that was the focus of load-based test cases in both phases of experimentation.

The next component of the platform was a load balancer to spread client connections to the WPO across available servers, and the AWS Elastic Load Balancer ("Amazon elastic load balancing," 2013) shown between server two and three through five allowed multiple WPO server instances to operate in parallel and service simultaneous web requests. The combination of the WPO servers and the load balancer comprised a basic working web application without SAS or RV&V feedback in the component architecture.

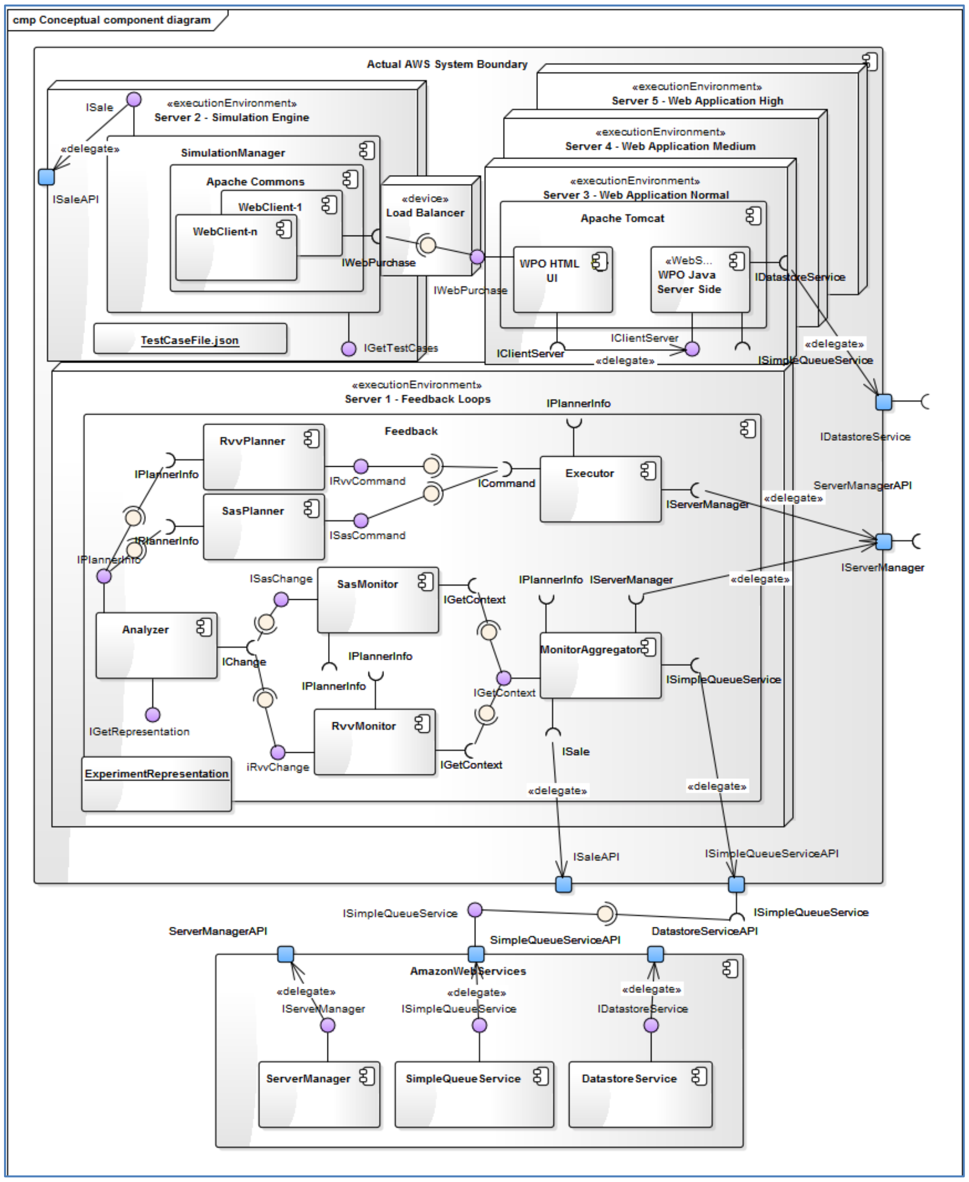


Figure 17. UML component diagram

The web load simulator, shown as server two, was not prescribed by Tamura et al. (2012), but was a required component to drive experimental load. This simulator created multiple web clients to load the WPO application servers to the level defined by the test cases. The SimulationManager read in serialized test cases and generated the appropriate number of web clients per time period. It also managed the simulation events via the iSale interface for the entire environment by generating sale day of year, social network sale events, and black Friday sale events as defined by the test cases. The simulation server was sized so that a single server could generate enough simultaneous client transactions to achieve the test case loads that were described later in this chapter.

The primary and secondary feedback loops – depicted in the bottom half of Figure 17 – constituted the SAS and RV&V components of the application. These components represented seven logical modules of the Feedback application that are linked together via an event listener pattern to form the Tamura et al. (2012) SAS RV&V framework. Each module performed a specific role in the feedback process that will be further described in the feedback loop section. At a high-level, the Feedback component gathered internal context in the form of WPO transaction times and concurrent connections per server from a database queuing service, and external context from the iSale interface regarding the sale event previously described. The feedback component issued commands through the ServerManager interface to add or subtract WPO application servers. Like the simulation server, the feedback loop's performance was not to be considered a measured component of the research, and thus this server was sized so that server load – whether CPU or I/O – was not a limitation during experimentation.

The last component in Figure 17 was a set of wrappers around supporting AWS services that are used by the previously described components. The transaction performance QoS metrics and concurrent connections metrics, previously discussed, were logged to the iSimpleQueueService or iDatastore service interface. These two interfaces were implemented by simply polling a database transaction table using a stored procedure based on a timer in the Feedback application. The metrics from this table were the primary input to the feedback loops and are described in detail in the next section. AWS provides a queue service on which to implement the iSimpleQueueService, but simple MySQL queries using the stored procedure were utilized. AWS also provided multiple Structured Query Language (SQL) data store services from which to choose. The MySQL RDS instance was the selected datastore capability to implement the iDataStore interface. It held the static WPO configuration information

and purchases from the WPO. The last component and interface depicted at the bottom of Figure 17 was the iServerManager. This application is a thin wrapper around the AWS server Application Programmer Interface (API) for stopping, starting and load-balancing AWS images. The iServerManager interface was implemented as scripts wrapped around the AWL Command Line Interface (CLI). Calls to these scripts were made from the Feedback application with appropriate input parameters. The AWS CLI services were also not an evaluated portion of the experimentation, and it was assumed that these services can scale to meet the simulated loads experienced by the WPO servers. The following sections decomposed each component into its individual applications.

WPO Target Application

In this section the WPO target application architecture, requirements, and flow were described. The WPO, shown in the top right of Figure 17, was a simple purchasing website that was used as the target application for all SAS RV&V experimentation. The WPO was implemented using the Apache Wicket ("Apache wicket," 2014) web application framework to ease the coding of the web application, and each WPO application ran in a single Apache Tomcat container ("Apache tomcat," 2013) per server. Wicket applications used only pure

HTML pages as the web front end, and pure Java server-side components (Dashorst & Hillenius, 2009). HTML components were replaced by embedding wicket identifiers in the HTML. By using this pure HTML/Java architecture, automated web clients more easily loaded the WPO application without the complexity of executing JavaScript on the client side. Most automated HTTP web client APIs do not execute embedded JavaScript.

The WPO application followed the page flow shown in Figure 18. Six pages were available to users. These were the HomePage, CustomerLogin, CreateAccount, ItemDetail, Cart, and Purchase webpages, and their interactions were also shown in Figure 18. First, the HomePage presented 15 items for purchase. Each item had a graphic image, name, price, and a link to the ItemDetail page. Graphic images were stored on the web server and the application associated images to the appropriate product using a filename URI stored in the database.

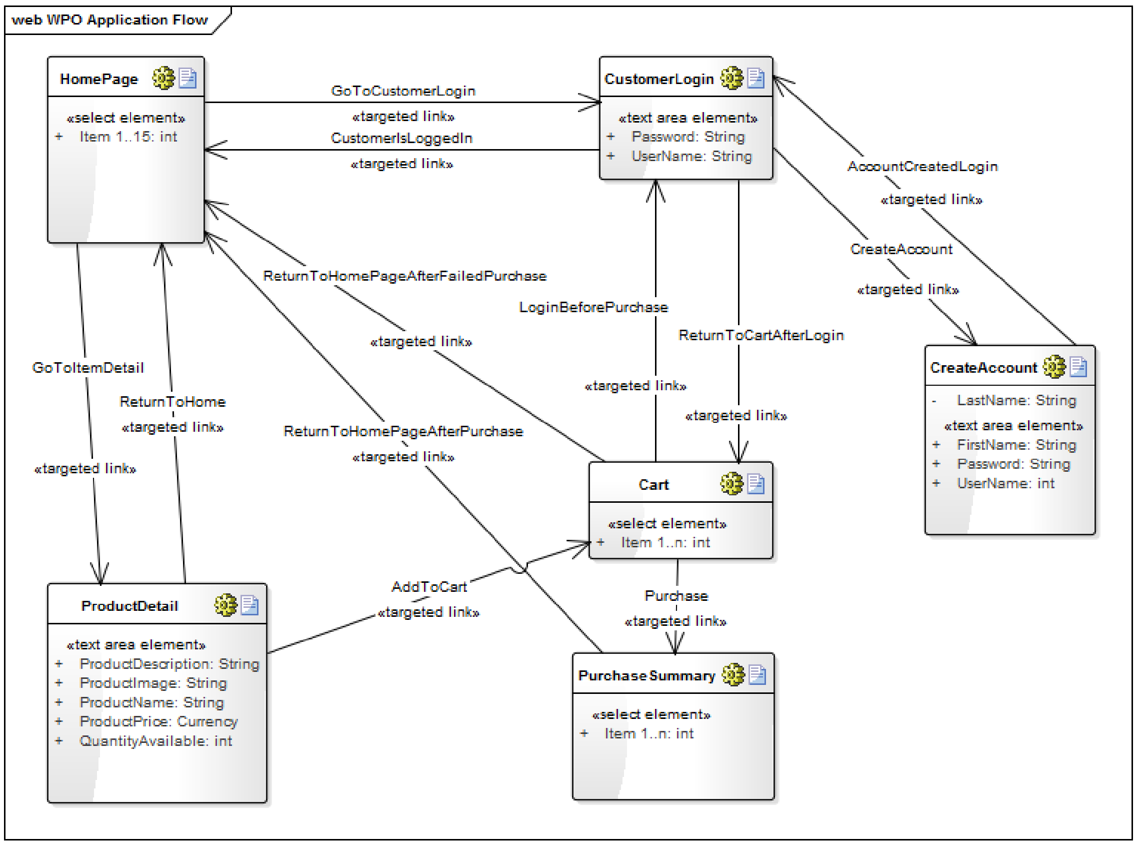


Figure 18. Web page model

The products displayed on the HomePage were populated from the SQL datastore. On the HomePage, there was an additional link to the CustomerLogin page. Once a WebClient has logged in, they were returned to the HomePage and may then Purchase products. Whenever a WebClient accessed the HomePage, a start time was recorded to capture the start of the session. This session time was referred to as the Processing Purchase Order (PPO) metric. Each WPO session that results in a purchase logged this metric to the datastore, and values averaged by minute drove Feedback loop operations. The PPO time records also included a flag to delineate preferred PPO time from regular PPO customer transaction times. Preferred customers had a different set-point for service time than regular customers, and this set-point was established at 90% of the regular PPO setting for this experimentation (Tamura et al., 2012). This 90% set point was established in the RV&V description provided by Tamura et al. (2012) as a setting that may be modified by the RV&V feedback system at runtime. For the purposes of this experimentation set points were not modified, but this capability was available to the architecture as a possible focal point for future work.

When a product link was selected on the HomePage, it took the user to the ProductDetail page where the image, name, description, and price were re-displayed. This page also displayed the product price, a link to add the product to the Cart, the quantity of products in inventory, and a quantity to be purchased entry box. The initial quantity available for each product was 20. No desired behavior or limitations drove the selection of this value for product quantities. Product quantities, or the lack thereof, had no impact performance of the experimental system. The ProductDetail page also provided a breadcrumb that allowed the WebClient to navigate back to the HomePage if the product was not added to the cart.

When a product was added to the Cart, the WebClient was taken to the Cart page. The Cart page contained a list of the selected products in the current transaction by name, price, and quantity. If the WebClient was not logged in, products may still be added to the Cart, but they could not be purchased. If a WebClient abandoned a Cart, it was then returned to the HomePage. The PPO end time was not recorded for these failed purchase sessions as PPO inherently implies a completed transaction. Other metrics could be devised to evaluate failed purchase transactions in future work, but the Tamura et al. (2012) description did not go into sufficient detail to address the impact of failed transactions due to inventory outages. If a WebClient attempted to purchase a product whose available quantity is zero, that session failed, and the page will be redirected to the HomePage. As a side effect of this failure, the item available quantity was reset to 20 as the normal way of simulating a restocking of a product for the WPO application. No costs were associated with the restocking of product in the simulation. A link to the CustomerLogin page was also be available from the Cart page. If a WebClient navigated to the CustomerLogin page, they could login or create an account from this page. Once logged in, the CustomerLogin page returned to the Cart page.

The CustomerLogin page was able to be accessed to from the HomePage or from the Cart page. The CustomerLogin page required the WebClient to provide an email-based username and password. These two text fields can't be the same value and were required to match an existing account in the SQL datastore. Upon a successful login, the WebClient was returned to the page they were on prior to navigating to the CustomerLogin page, and the name of the account holder displayed in the top right of the page banner replacing the Login link. New users to the site navigated from the CustomerLogin page to the CreateAccount page. This page contained text entry boxes for a first name, last name, email address, and password. The last item that a new user entered was a check box to determine if they are a PreferredUser. The users defined in the ExperimentalRepresentation identified whether they were a Regular or Preferred User in order to utilize this check box. Once these items were submitted, the user was redirected back to the CustomberLogin page to authenticate.

From the Cart page a WebClient that was logged in may purchase items, and view that purchase on the PurchaseSummary page. The PurchaseSummary page displayed cart information, a total price, the date that the items on which will be delivered. The WebClient considered these steps a completed transaction five seconds after the PurchaseSummary page was displayed, and then returned the user to the HomePage. The WebClient ended the transaction at this point, and the PPO timer recorded the total time of the web transaction.

Recent comparisons of customer shopping behaviors noted that the time between clicks in a completed customer transaction was insignificant to overall behavior for shopping experiences between five and 30 clicks (Gupta, Mittal, Singla, & Bagchi, 2014). Every WPO transaction completed within this stated click range, and thus the use of the five second delays throughout this research had no not impact results. Gupta et al. (2014) also noted that a typical customer transaction was a constant at 6.8 minutes, independent of the calendar or specific shopping events. The Gupta et al. (2014) work stated that treating a web shopping transaction as a directed graph with a defined end state could be evaluated by a time-homogenous model, and thus individual click-timings were not significant to shopping behavior. Thus, the click timing in this WPO application was defaulted to a single value of five (5) seconds throughout. No attempt was made with this selection to simulate or negate CPU or I/O load factors caused by this click delay. The determination of whether CPU or I/O load emerged as the dominant factor was unknown prior to experimentation, but experimentation showed that the mixed 1-idle% provided the best metric of system load.

The WPO application populated items and authenticated users from a SQL datastore consisting of a USER, PRODUCT, and ORDER tables. This simple schema shown in Figure 19 will allowed a user to place many orders, and an order consisted of one or many products. An intermediate PRODUCT_ORDER table provided the linkage between this many-to-many relationship. A separate PPO_METRIC table collected the duration of each transaction using the transaction start and end times. This information was used by the feedback system to determine a mean PPO value for each evaluation time period, which was defaulted to one minute. The one minute value was selected as a sampling period that would be greater than a standard transaction so that transactions of at least one regular or preferred customer would always occur within a sampling period. No more detailed reasoning was applied to this sampling period.

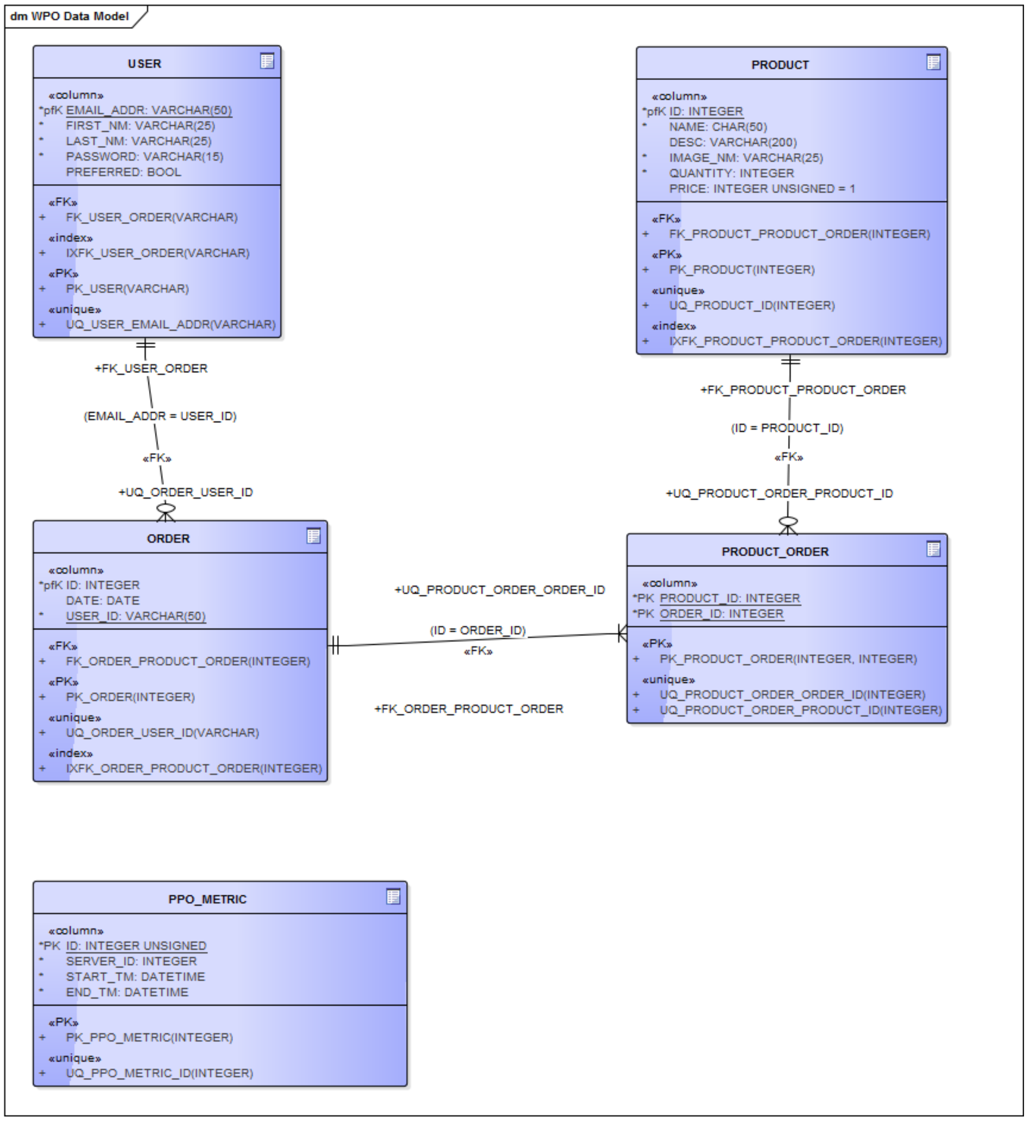


Figure 19. WPO application data model

Feedback Loops

This section described the Server One feedback subsystem shown in the Figure 17 component diagram. The entire feedback system was contained within a single Java application, aptly called Feedback. This application consisted of a main module and seven additional sub-modules that implemented the MonitorAggregator, SasMonitor, RV&VMonitor, Analyzer, RV&VPlanner, SasPlanner, and Executor subsystems from the proposed Tamura et al. (2012) RV&V feedback approach. This feedback system supported both RV&V and self-adaptive behavior through the same external interfaces. During experimentation, RV&V output was disabled in order to establish SAS baseline performance data via settings in the resource bundle. A general description of the external, component-level interfaces and subsystem performance were provided next and then details of each thread's characteristics were further expanded.

The Feedback application was manually started once Server One booted from its virtual machine image. The Feedback system initialized itself using an object representation of the ExperimentalRepresentation ontology. This representation contained all context necessary to define both SAS and RV&V feedback behavior for WPO application scaling. The Feedback application registered with the iServerManager interface at startup and then waited for a message from the iSale interface to inform the Feedback application that the simulation has started. Feedback subscribed to an input interface from the iDataStore to gather external context in the form of the PPO and CC/Svr metrics, and these metrics were gathered once a minute to align with the PPO sampling time as discussed in the previous section. The Feedback application also gathered external context from the iSale interface to determine sale day and social network events. As noted above, this same iSale interface also started and stopped the simulation via a simple Boolean variable getter. The last interface was the input/output interface to iServerManager. This interface contained the registration status functionality and a set of simple getters and setters that tell the ServerManager to log the starting or stopping each of the WPO servers.

After registration of the Feedback application was complete, the first task to be spawned was the Analyzer. This task's role was to replicate the ExperimentalRepresentation into an in-memory database for the feedback subsystem. It had no feedback logic, but kept the ontology up-to-date and distributed data to the other appropriate tasks in the feedback loop. The data in the static object model contained the SAS states, RV&V goals, and other context information needed to conduct experimentation. Each fact in the object model was tagged so that the appropriate task was notified of state changes during initialization. For the purposes of this experimentation, the entire initialization sequence using the ExperimentalRepresentation was short-circuited, and downstream tasks were pre-initialized with the data they need to conduct simulation runs. This shortcut reduced the synchronization coding, but did not impact experimentation. During simulation runs, the in-memory view of the ExperimentalRepresentation was updated with new state information from the previously discussed interfaces. The iPlannerInfo interface notified all other tasks that an updated data item was available, and each data item was tagged with the appropriate name for which the data was destined. The iChange interface aggregated changes in monitoring context from the SasMonitor and RV&VMonitor via the iSasChange and iRV&VChange interfaces, respectively. All interfaces internal to the Feedback application implemented a modified

Observer pattern (Gamma, Helm, Johnson, & Vlissides, 1995) so that the Analyzer gained and advertised the status of a changed data item, updated the model, and then announced it to all subscribers via the iPlannerInfo interface.

The next task to be activated was the SasMonitor. This task read from the iPlannerInfo interface, described above, and initialized all SAS monitoring. For this experimentation, the SasMonitor was only concerned with determining when a sale, a social network event, or black Friday event occurs. These events were delivered to the SasMonitor by the iGetContext interface from the MonitorAggregator. iGetContext messages were tagged as SAS or RV&V changes. When any of these SAS events occurred, the SasMonitor advertised the change via the iSasChange interface.

Like the SasMonitor, the RV&VMonitor only focused on the context items assigned to RV&V via the iPlannerInfo interface. The RV&VMonitor started up right after the SasMonitor and collecSted the two PPO metrics and CC/Svr metrics for each of the three WPO servers. These metrics were delivered to RV&VMonitor via the iGetContext interface. The RV&VMonitor then delivered these metrics to the Analyzer via the iRV&VChange interface. No strong system logic was contained in the RV&VMonitor, except that it delivers RV&V context changes to the Analyzer.

The MonitorAggregator was the input frontend for the Feedback application. All internal and external context inputs were delivered to the MonitorAggregator. This task polled for the metrics described above from the iDataStore service. Thus, this task managed a timer for each interface, woke up at the appropriate time, and sampled the interface for updated context. MonitorAggregator also polled the iSale interface to get changes in the sale configuration of the system, and like the downstream monitors configured itself using the iPlannerInfo internal interface. The MonitorAggregator posted all context updates to the iGetContext output interface with the appropriate routing information for either the SasMonitor or RV&VMonitor. Again, the MonitorAggregator had little system logic, but derived all behaviors from the configuration information provided by iPlannerInfo at startup.

The SasPlanner contained the simple SAS logic for the experimental example. The SAS logic and its representation were discussed in further detail in the Data Representation section that follows. The SasPlanner used the States from the ExperimentalRepresentation to move from a single server to a three server configuration based on external context input. The SasPlanner followed the rules defined in Figure 20 below. If a SaleDay status was TRUE, the SAS response was to tell the Executor to start a second instance of the WPO application on Server Four by moving to the MEDIUM_STATE using Rule1. When the SaleDay status transitions from TRUE to FALSE, the SAS response delivered to the Executor was to shutdown Server Four using Rule4.

This same approach was used when a SocialNetwork status equates to TRUE. If a BlackFriday status was TRUE, the SasPlanner told the Executor to activate Server Four and Server Five per Rule3, and then shut Server Five down when the status goes back to FALSE using Rule6. When the system is operating in the HIGH_STATE and all events are false, then it transitioned back to the NORMAL_STATE. On the subsequent metric sampling Server Four was shutdown using Rule4 or Rule5. Each of these commands was delivered to the Executor over the iCommand interface. The only input to the SasPlanner was from the iPlannerInfo that informed both Planners of context changes to evaluate.

```
RULE1: if ((NORMAL_STATE ) && (SaleDay == TRUE))
then GOTO_MEDIUM_STATE
OR
RULE2: if ((NORMAL_STATE)  && (SocialNetwork == TRUE))
then GOTO_MEDIUM_STATE
OR
RULE3: if (((MEDIUM_STATE) || (NORMAL_STATE))
&& (BlackFriday == TRUE)))
then GOTO_HIGH_STATE
OR
RULE4: if ((MEDIUM_STATE)  && (SaleDay == FALSE))
then GOTO_NORMAL_STATE
OR
RULE5: if ((MEDIUM_STATE)  && (SocialNetwork == FALSE))
then GOTO_NORMAL_STATE
OR
RULE6: if ((HIGH_STATE)  && (BlackFriday == FALSE)
then GOTO_MEDIUM_STATE

RULE6: if ((HIGH_STATE)  && (BlackFriday == FALSE &&
SocialNetwork == FALSE && SaleDay == FALSE)
then GOTO_NORMAL_STATE
:
```

Figure 20. SAS state transition rules

SAS transitions were not based on load measurements as described above. They were only based on external context changes, such as SaleDays, SocialNetwork events, or BlackFriday sales. In this experimentation the SAS feedback subsystem reacted to sale events, and the RV&V subsystem monitored customer throughput and cost to determine if additional remedial actions should be taken.

The RV&VPlanner followed a similar startup approach to that of the SasPlanner to startup and configure itself. The RV&VPlanner focused on comparing five metric values to the set points provided from the ExperimentalRepresentation. Note that each of the set points were determined in the Phase One experimentation. For this purposes of these experiments, the Planners did not change the set points during execution, but only reacted to goal violation. As noted previously, reduced settling time will not be addressed in this experimentation; however, *Until* state transitions did provide a hook by which an intermediate MinimizeSettlingTime rule between the MaximizePPO and MinimizeCost could be applied to reduce the perceived settling time caused by second server spin up. This rule also suppressed the MinimizeCost rule during state transitions. This rule was not demonstrated in this research, but the architecture easily supports such a rule construct. It should also be noted that the spin up and spin down time associated with the WPO application did not provide a sufficient structure by which settling time improvements could be clearly demonstrated.

The equations that verify whether a goal has been achieved were called MonitoringConditions, using the Villegas et al (2011) terminology, and each rule correlated to a ServiceLevelObjective (SLO). The RV&VPlanner first serviced the highest goal – MaintainCustomerThroughput. This goal was verified by making sure the RegularPPO was less than the RegularPPOSetpoint. Each of the customer throughput goals were shown in Figure 21 below. The second verification rule was that the PreferredPPO was also less than the PreferredPPOSetpoint. A violation of these goals forced a change to the SAS state model by moving to the NEXT_STATE. When RV&V is active, the RV&VPlanner commands took priority over the SasPlanner commands.

```
RULE1: if (RegularPPO >= RegularPPOSetpoint)
then GOTO_NEXT_STATE
OR
RULE2: if (PreferredPPO >= PreferredPPOSetpoint)
then GOTO_NEXT_STATE
```

Figure 21. Maintain customer throughput monitoring condition

Thus, the SAS commands to activate a server on a sale day may be suppressed if the PPO metrics were not violated, saving server runtime costs. If a PPO metric was violated, then the RV&VPlanner issued a command over the iRV&VCommand interface to spin up another WPO server.

The second goal that the RV&VPlanner enforced was the MinimizeCost goal. This secondary goal looked at the CC/Svr metrics for each server and executed the MonitoringCondition logic of Figure 22. The actions of secondary goals may only be executed if they did not violate the rules of a primary goal, and therefore the rules of Figure 21 always took precedence over Figure 22's goals. This approach was the key outcome of

```
RULE3: if ((Svr4 == ACTIVE && Svr5 != ACTIVE) &&
(CC/Svr3 + CC/Svr4 < MaxCC/Svr)) then SHUTDOWN Svr4
OR
RULE4: if ((Svr4 == ACTIVE && Svr5 == ACTIVE) &&
(CC/Svr4 + CC/Svr5 < MaxCC/Svr)) then SHUTDOWN Svr5
```

Figure 22. Minimize cost monitoring condition

the Hierarchical, goal-based governance approach over the standalone Villegas et al. (2011) approach. The MinimizeCost goals shutdown a server as soon as the number of concurrent connections on two servers was less than the maximum number for a single server. With RV&V enabled the MinimizeCost goal reduced the total server activation time over SAS-only operation.

The Executor was the last task in the Feedback subsystem. This task simply took commands from the SasPlanner and RV&VPlanner via the iCommand interface, made sure they were not conflicting, and used the iServerManager interface to activate or shutdown WPO server instances. The iServerManager interface had a simple setter interface that activated or shutdown a WPO server. Conflicts could occur when a SAS command conflicts with an RV&V command. For this experiment, RV&V commands always took priority over SAS commands. While the Executor also had an iPlannerInfo interface, this interface was not needed for this experimentation.

Simulator

The SimulationManager was the last component in the experimental system shown in Figure 17. This was a standalone application that performed three major tasks. First, it read in the test case description file, and initialized itself. This file utilized the Javascript Object Notation (JSON) file format to define a fractional load-rate value per-hour as a factor of the maximum number of concurrent connections per server determined in the Phase one experimentation. This load rate was used to determine the number of web client threads that were utilized to appropriately load the WPO application. The second feature of the JSON file was that it defined when sale events occurred. A sample of the object data for both load and event timing was shown in Figure 23 below.

The second major task of the SimulationManager was to create and maintain a pool of threads that will act as HTTP clients to the WPO application. At startup, this pool was initialized to three times the number of maximum concurrent connections per server that was determined in the Phase one experimentation. Since Phase One experimentation was focused on a single WPO server, it was assumed that three times the number of threads necessary for Phase One test cases was sufficient to execute the test cases across the three WPO servers. A web client thread waited until the main timing system activates a session. The timing system then passed in a random number that the web client used to associate itself with one of 25

Figure 23. Load and event timing JSON example

```
{
“load” :  {
“0000” : “0.25”,
“0100” : “0.35”,
….
“1600” : “1.25”
},
“events” : {
“saleDay” : {
“start” : “0000”,
“end” : “2359”
},
“socialNetwork” : {
“start” : “3600”,
“end”  : “3630”
}
}
}
```

possible users. No behaviors or limitations were implied by the selection of 25 users. The only expectation on the selection of the number of 25 users was that user account creation more highly loaded the system early in test cases, but this lower number of users caused account creation to cease early in each test case. Users were not created prior to the simulation, but were created as part of a WPO transaction, if they don't exist. This approach provided some variation in transaction timing and more realism for the simulation. Web clients were also initialized with one of three possible approaches to the WPO application. The first approach was be a login, a random selection of a number of items, and a purchase.

The second approach was browsing for items, attempting to purchase items, a forced login, and then finally a purchase. The last approach was to browse items, but abandon the session before a purchase. The last approach was executed much more often as the first two to simulate browsing, but not buying, activity being the predominant session for a purchasing site. Further discussion of this browsing behavior was addressed in the test cases section below.

The last major function of the SimulationManager was to announce sale day, social network, and black Friday events via the iSale interface. A setter function allowed the application to set each of these events to a TRUE state, and the interface retained that state until the end time arrived. The main thread of the SimulationManager managed the time, per the test case definition file, and determined when a particular sale event was in force or expired. The SimulationManager also logged all activities in a local log file in order to provide a basis for debugging and subsequent analysis. The SimulationManager provided the capability to simulate the entire experimental platform via a file-based, configurable interface, which was used in both phases of experimentation.

Test Cases

Two forms of test cases were defined in this section. In the first phase of experimentation the proposed test cases were designed to establish settings so that a programmable load can be generated in phase two. Phase one also established the maximum PPO and CC/Svr set points needed to instrument the second phase of experimentation. Phase one test cases were executed concurrently, and the experimentation section further defined how they are employed to establish the measures needed for subsequent experimentation. The second phase test cases were designed specifically to address the stimulus events described in the Tamura et al (2012) application example. The following sections described each test case in detail and the reasons why their characteristics were necessary for this research.

Phase One Test Cases

The WPO application modeled a simple sales website with common features like a shopping cart and purchase pages. Chung and Park (2009) evaluated over 3,000 weblog entries for the Amazon.com site, and inferred that the ratio of browsers to buyers for a shopping web site was 50 to 1. While the customer motivation to purchase items varied depending on whether a user typed a URL directly or was referred to a site from another, this browsing versus buying ratio held (Chung & Park, 2009).

In phase one only a single WPO application server was loaded with web clients from the SimulationManager. Sessions traversed the load balancer, but no session management was performed since there is only one application server. As well, no feedback loop behavior was utilized in phase one. The goal of these test cases was to determine the parameters necessary to adjust WPO performance through the identification of the Load Increment concept and its associated parameters. Each of the Load Increment parameters was described below. Phase one also established the maximum PPO and CC/Svr metrics. While CC/Svr included browse-only and purchase sessions, PPO metrics were only be gathered from sessions that resulted in a purchase.

In order to realistically load the WPO application, the vast majority of client sessions only browsed the site, but did not make purchases per the ratio from Chung and Park (2009). This browse-only behavior was reflected as Test Case One in Table 1. In this test case the automated web client accessed the HomePage shown in Figure 17 and selected an item. This selection took the client to the ProductDetail page for that item. Each transition in this browse-only test case implemented a five-second delay between link selections to emulate human interactions, and not the speed of an automated tool. From the ProductDetail page, the client returned to the HomePage and randomly selected another item in the list. This random selection of an item had no impact on the research outcomes, but better emulated human behaviors across a user base. Gupta et al. (2014) confirmed that for sales web sites where the click-rate for a purchase is between five and 30 clicks, the time between clicks was homogenous – or had no significance. The product selection, browse the ProdcutDetail page, and return to the HomePage cycle continued for three iterations. After the five second delay on the third ProductDetail page selection, the web client returned to the HomePage and the session was deemed complete when the HomePage screen was delivered to the automated web client.

Table 1. WPO test case one

Step	Description
1	Navigate to the WPO URL HomePage and delay five seconds
2	Select a product link and view the ProductDetail page for five seconds
3	Return to Step 1 until three (3) views are complete
4	Select the HomePage after delaying for five seconds, and terminate the session when the HomePage is displayed

The second and third test cases were not based on external web purchasing research, but on the two methods by which the WPO site may be traversed to purchase a product. The three test cases were summarized in Table 2 below. These test cases were randomly interleaved into 49 instantiations of Test Case One, and the execution of 50 test cases was described as a single Cycle. Random execution of test cases two and three with the browse test cases introduced more realism and reduced load spikes during purchase events. Test Case Two was executed on odd cycles, and Test Case Three on even cycles. Cycles overlapped as threads from a previous cycle were still executing when a new cycle began. This behavior was necessary so that a constant or increasing load was presented to the WPO application.

Table 2. WPO phase one test cases

Number	Description	Cycle Detail
1	Browse three items	49 executions per cycle
2	Login then purchase	1 execution per even cycle
3	Attempt to purchase, then login	1 execution per odd cycle

The Test Case Two method of execution was summarized in Table 3 below. A web client retrieved the HomePage and then traversed from the HomePage to the CustomerLogin page. The client then attempted to login, and navigated to the CreateAccount page if no customer login existed. Each web client was provided with a randomly selected user account upon creation. The client attempted to login with this credential, and if unsuccessful created the appropriate user account. New users then executed a login. Upon a successful login, the client was redirected back to the HomePage. Each link selection was again separated by a five second delay as in Test Case One. The web client selected a product link and navigated to the ProductDetail page. The product was added to the Cart and subsequently purchased. The web client was then be presented the PurchaseSummary page, and terminated the session after the standard five-second delay.

Table 3. WPO test case two

Step	Description
1	Navigate to the WPO URL HomePage and delay five seconds
2	Select the CustomerLogin link and delay five seconds
3	Login with a random selection of user account credentials with the assigned username / password combination. If successful, go to Step 5
4	Create a new customer account and return to Step 3 after a five second delay
5	Return to HomePage and select a product after a five second delay
6	Select the add to cart link on the ProductDetail page and navigate to the Cart page after a five second delay
7	Select the purchase product link from the Cart page, and view the PurchaseSummary page. Terminate the session after a five second delay

Test Case Three provided a variation on Test Case Two and traversed the WPO site to make a purchase through a different path. After navigating to the HomePage, a product was selected and the ProductDetail page was displayed. The product was then added to the Cart, and a purchase was be attempted. Since the user was not logged in, the web client was redirected to the CustomerLogin page. The web client attempted a login, and a successful login redirected the customer back to the Cart page. If the web client's credentials were not recognized, then the client was directed to the CreateAccount page. As in Test Case Two a new account was created and the new accountholder logged in. Upon a successful login the web client was redirected back to the Cart page, and the client purchased the product. The client was presented with the PurchaseSummary page, and the session ended five seconds after the transition. As in the other test cases a five second delay was implemented between actions. These steps are detailed in Table 4 below.

Table 4. WPO test case three

Step	Description
1	Navigate to the WPO URL HomePage and delay five seconds
2	Randomly select a product and navigate to the associated ProductDetail page and delay for five seconds
3	Add the product to the Cart and delay five seconds
4	Attempt to purchase a product, and then be re-directed to the CustomerLogin screen after a five second delay
5	Login. If successful, go to Step 6
6	Create a new customer account, and then return to Step 5 after a five second delay
7	Purchase a product from the Cart page after a five second delay
8	View the PurchaseSummary page, and terminate the session after a five second delay

Each WPO server consumed some small percentage of CPU load to run the basic operating system, Apache Tomcat, and other associated infrastructure applications. This quiescent load was defined as MinLoad, and may be identified by CPU, IDLE, or I/O wait times as identified by sar data. In this phase of experimentation it was determined which was dominant factor in WPO application session performance. The remaining IDLE, CPU, or I/O capacity was divided into equal increments, called Load Increments or L_i. Each L_i represented 5% of the available WPO processing capability, divided into 20 unique increments. Each test case Cycle consisted of 50 test case runs, and each test case run was interspersed with a short delay between the start of sequential test cases. This delay was called the intra-cycle delay (C^a), and was defaulted to five seconds. No benefit or limitation was implied by the five second default. Since each test case run spanned many seconds, and file service access was sporadic within the test case steps. C^a was a factor used to tune a Cycle to achieve a relatively flat or even CPU utilization throughout the Cycle. Purchase test cases were randomly interspersed between browse-only test cases and resulted in increased processing load as shown in the CPU usage spikes of Figure 24. In order to increment load on a WPO server in 5% chunks, multiple Cycles had to be run simultaneously. The number of simultaneous Cycles that had to be run to achieve a single L_i was referred to as Concurrent Cycles or C_c. Lastly, when multiple cycles were run to achieve an L_i increment, a delay could be required between these incremental Cycles to achieve a smooth step function of load. This delay between the start of Cycles was referred to as inter-cycle delay or C^r. The formula in Figure 24 defined the triple of the factors of L_i. Figure 25 showed a graphical representation of each of these factors in L_i as well.

$$\boxed{MaxLoad := 20 \times L_i \langle C_c | C^r | C^a \rangle}$$

Figure 24. Max WPO server load equation using load increments

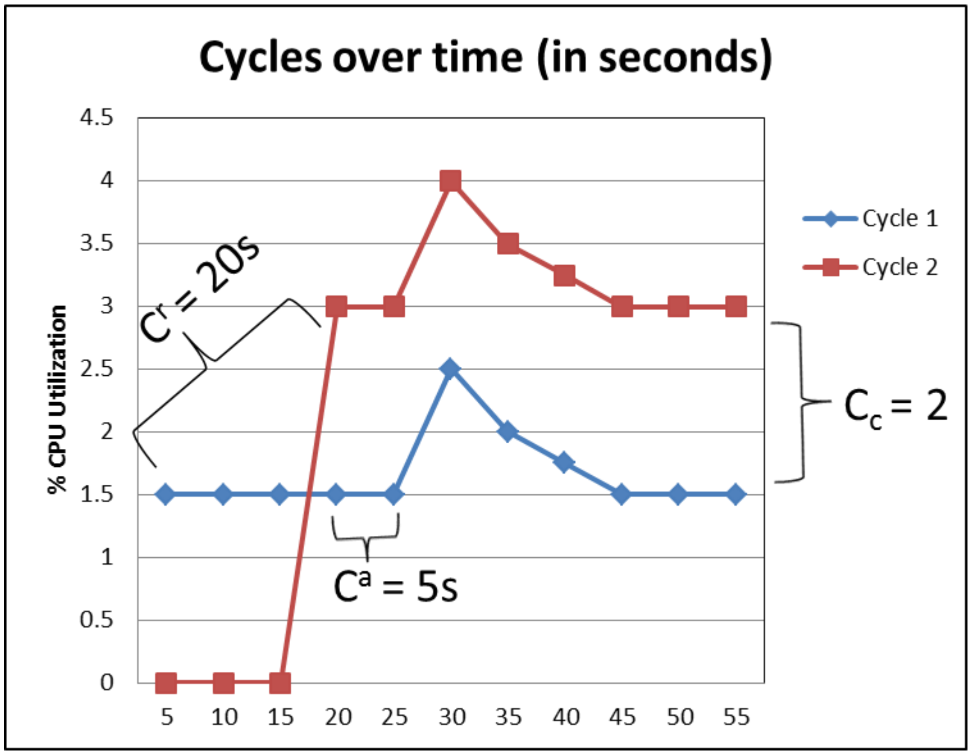


Figure 25. Load increment factor visualization

Phase two test cases

As has been previously stated, the purpose of experimentation in this phase was to activate SAS feedback behavior, and subsequently measure the impact of a second RV&V feedback layer while re-using the same inputs. To that end, phase two test cases differed in structure and form from those in Phase One, but they utilized the phase one Cycle approach of browsing and buying to apply a realistic WPO load. Time-based load rates in the form of L_i multiples formed the basis for phase two test cases, along with external context events at specific points in time. These external context events were injected into the feedback subsystem, shown in Figure 17, in order to produce expected SAS responses. These multiples of L_i provided a descriptive tool to instrument increasing or decreasing browsing/buying load.

The distinctions between the L_i load concept and the PPO metrics were further expanded below. While both metrics were baselined in phase one, they served two different purposes. L_i defined a measure of concurrent web sessions that produced a 5% load on the WPO application. The L_i approach encapsulated the number and frequency of WPO sessions. The PPO metric exclusively defined the number of seconds it took to execute a purchase. L_i was not a metric that was measured or tracked by the context feedback system. It was exclusively used to drive simulated traffic. PPO focused on purchasing response behavior and RV&V interactions within the feedback loop.

For the purposes of the phase two test cases, a maximum PPO value (MaxPPO) was established, and maintained throughout the life of the test case in this experimentation. Varying the MaxPPO metric based on feedback was also a possibility to further hone performance, but this additional facet of the SAS RV&V problem was reserved for future work. L_i multiples were defined in the test cases themselves, varied with time, and spurred SAS and RV&V behavioral changes in concert with context events. MaxPPO was an entity defined in the ExperimentalRepresentation ontology

Three test cases were executed in this phase, and each was inferred from the description in Tamura et al. (2012) of a concrete industrial case study. Additional detail and constraints were applied to each of these test cases in order to better emulate a realistic shopping site behavior. The first of these constraints was a consistent load pattern that varied based on the time of day for a shopping website. Figure 26 showed a representative hourly load gleaned from Rosenstein (2000) of measurements taken from a United States e-commerce web site operating primarily in the Eastern Time zone. In this graphic the black bars represented true customer traffic versus the total traffic that included web-bots or other automated tools. This general pattern of web traffic was used in the phase two experimentation.

In Table 5 below, the percentage utilization values were achieved by incrementing L_i multiples and applying these values to each test case time period. From midnight until 8am server load stayed within a range of 10 to 20% of MaxLoad. This maximum load for a single server was 20 times L_i as shown in Figure 24. At 8am, server load increased in a pseudo-linear fashion until 10am when it achieved 50% of the MaxLoad. From 10am until 5pm the WPO load oscillated between 50% at the start of this period to 90% of the MaxLoad as determined in phase one. From 6pm until 10pm, load was reduced in a pseudo-linear fashion back down to 10-20% of the MaxLoad. Lastly, from 10pm until midnight server load was maintained at 10% of the MaxLoad.

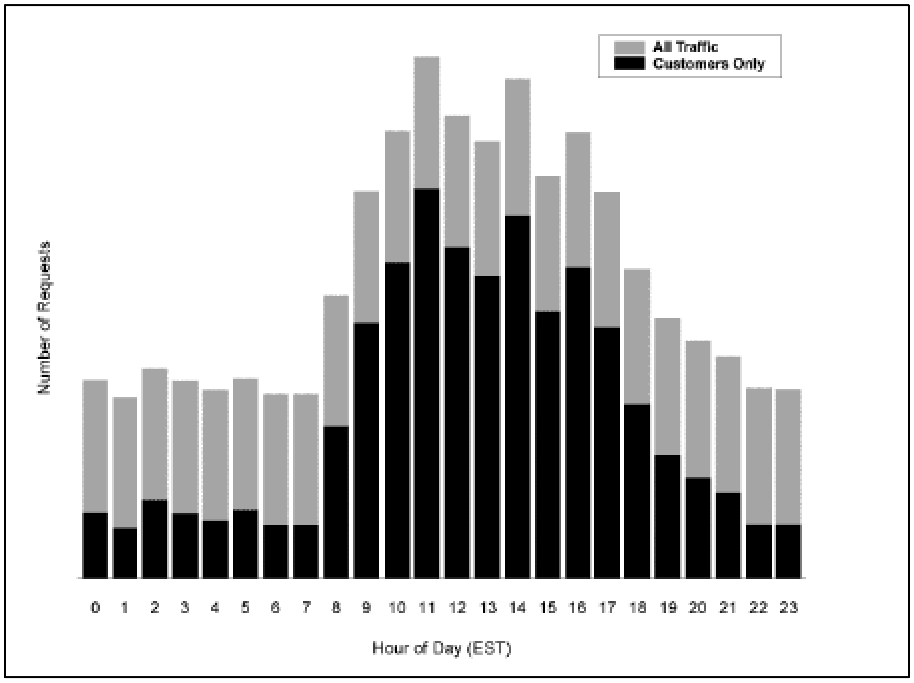


Figure 26. Hourly web traffic (Rosenstien, 2000)

The first test case covered a 24-hour period that spanned a normal day of operation, with varying loads by shopping hour. This first test case verified that the system operated

Table 5. Periodic server load

Time Period	Customer Load Description
12am to 8am	Server load set at 10% and 20% of the MaxLoad
8am to 10am	Pseudo-linear increase to 50% MaxLoad
10am to 5pm	Hourly oscillations from 50% MaxLoad load up to 90% of the MaxLoad
5pm to 10pm	Linear decrease to 10% of MaxLoad
10pm to 12am	Maintain 10% of maximum MaxLoad value

properly without state transitions in the Normal state, roughly following the Figure 26 curve. The second test case modeled a 24-hour period that spans the day before, during, and after a holiday sale. This test case demonstrated the Normal to Medium transition and back, based on a changing calendar context. The third test case covered a 24-hour period where a Normal day included a four-hour social networking event sale and corresponding transition to the Medium state.   The last test case encompassed a 56-hour period bracketing a Black Friday sales event where the system expected maximum loading, but this test case was not executed during experimentation. Table 6 below described each test case.

Table 6. Phase two test case descriptions

Number	Name	Period	Number of WPO servers expected	Description
1	Basic	24 hours	1	Demonstrates Normal state behavior of the WPO without state transitions, but changing load throughout the day with a fraction C/s value
2	Sale Day	24 hours	2	Demonstrates Medium state behavior of the WPO with state transitions based on a Calendar event. The system will execute the test case where a SaleDay event begins at 0800 and concludes at 1800. A Normal to Medium state transition is expected.
3	Social Network	24 hours	2	Demonstrates Medium state behavior of the WPO with state transitions based on a Social Network sale event. The system will start out in a Normal state and ramp up to Medium capacity during a four-hour sale, and then conclude the Normal day.
4	Black Friday	26 hours	3	Demonstrates complete state transition behavior with transitions from a Normal state on the day prior to a High state during Black Friday, and then back down to a Medium state with a following day sale. This behavior is based on a Black Friday calendar event and a following day Calendar sale event.

Test case one – called Basic – executed completely within the Normal state. It closely mimicked the load profile shown in Figure 26 and Table 5, and spanned a 24-hour period. Table 7 displayed the content that was encoded in the JSON format discussed in previous sections and displayed in Figure 23. No external context events were enabled during this test case, and no SAS state transitions nor was RV&V intervention expected.

Table 7. Basic test case definition

Time Period	% of CPU
0000	10
0100	10
0200	15
0300	10
0400	10
0500	15
0600	10
0700	10
0800	30
0900	40
1000	50
1100	90
1200	60
1300	55
1400	70
1500	55
1600	60
1700	40
1800	35
1900	25
2000	20
2100	15
2200	10
2300	10

The second test case, called Sale Day, again spanned a 24-hour period and introduced an external context change of a sale day between 0900 and 1700. The test case started at 0000, which was the beginning of a sale day. As Figure 23 denoted the sale day event was encoded in the JSON file, and the SAS system reacted by activating a second WPO application server environment slightly in advance of the beginning of the sale day. Using the basic hourly profile discussed above, load increased by using the L_i multiples, except that the two-server load was no longer a fractional value less than 1.0. This load followed the Basic test case structure, but scaled to a maximum value of 175% of MaxLoad throughout the test case. As in the Basic test case, the PPO value set at the start of the test case was maintained throughout. Table 8 defined the associated load values for each hour. The Sale Day test case provided a structure by which the SAS behavior of reconfiguration to a two-server environment was evaluated to comply with the description in Tamura et al. (2012). This same test case caused different RV&V behavior to be exhibited to maintain system goals.

Table 8. Sale day test case definition

Time Period	% of CPU
0000	15
0100	20
0200	25
0300	20
0400	15
0500	25
0600	20
0700	20
0800	55
0900	70
1000	90
1100	160
1200	105
1300	100
1400	125
1500	95
1600	105
1700	70
1800	60
1900	45
2000	35
2100	25
2200	20
2300	20

The third test case, called Social Network, also spanned a 24-hour period and roughly followed the Basic test case curve; however, between 0900 and 1259 a social network event was introduced that attempted to increase MaxLoad by a factor of 2.0. This caused the SAS behavior to adapt to a two-server configuration prior to the start of the event and then reverted to a single server configuration after the social network event concluded. The input profile for the Social Network test case was displayed in Table 9 below. The Social Network test case again provided a format by which the Tamura et al (2012) description of SAS adaption was evaluated. As in the Sale Day test case, different behavior was exhibited when RV&V was enabled.

Table 9. Social network test case definition

Time Period	% of CPU
0000	10
0100	10
0200	15
0300	10
0400	5
0500	15
0600	10
0700	10
0800	5
0900	10
1000	100
1100	180
1200	120
1300	55
1400	70
1500	55
1600	60
1700	40
1800	35
1900	25
2000	20
2100	15
2200	10
2300	10

The last test case, called Black Friday, spanned a 26-hour period starting at 2300 on day one and ending at 0059 on day three. This test case was developed for experimentation but held for future work. The following description shows expected behaviors, but none of these outcomes were measured by experimentation. This test case, shown in Table 10, would exhibit a combination of the Black Friday event from 0000 to 1759 and a Sale Day event from 1800 to 2359 as would typically occur on the day after Thanksgiving. Black Friday load should activate the High state in the SAS state machine for the WPO application, and then revert to the Medium state after Black Friday concludes at 6pm. At the end of the Sale Day, the system should revert to Normal state behavior. This behavior should also engage RV&V behaviors that are unique from the SAS decisions. During the Black Friday period the MaxLoad factor will be multiplied by 2.8, and during the Sale Day period, the MaxLoad factor will again be multiplied by 1.75.

Table 10. Black Friday test case definition

Time Period	% of CPU
2300	10
2400	22
2500	30
2600	40
2700	30
2800	20
2900	35
3000	30
3100	30
3200	85
3300	115
3400	140
3500	255
3600	170
3700	155
3800	195
3900	155
4000	170
4100	115
4200	60
4300	45
4400	35
4500	25
4600	20
4700	20
4800	10

Each test case defined for phase two exhibited a specific context event and associated load profile identified by Tamura et al. (2012). The combination of context changes and varying load generated a dataset required to determine if the addition of SAS RV&V in a second feedback loop provided improved QoS performance over a single SAS feedback loop.

Experimentation

The proposed experimentation was conducted in two phases and consisted of three major steps. The first step was to collect load measurements and establish set-points using the phase one test cases in a single-server WPO configuration. The second step involved running the SAS simulation without RV&V responses enabled in a full WPO architecture to verify that the system demonstrated appropriate SAS behaviors. The last step was to introduce RV&V management of the second feedback loop in order to determine if RV&V improved overall system adherence to QoS metrics as ordered goals.

Analysis was conducted throughout phase one experiments to determine the appropriate settings for the L_i increments. Subsequent analysis after phase one testing was complete determined the maximum PPO settings for phase two. Some rudimentary analysis of SAS behaviors was also performed after step two to determine the total WPO application server up-time and if the MaxPPO metric was violated; however, the majority of phase two analysis was performed following the RV&V experimentation to compare SAS performance to that of SAS RV&V performance. The following paragraphs described each experimentation step in further detail.

Phase one experimentation

The purpose of this phase of experimentation was to baseline the performance of a single-server WPO application and the web load simulator. To achieve a repeatable method of increasing and decreasing load in 20 L_i increments, the quiescent server load was first base-lined as MinLoad. Then, the parameters for the proper implementation of L_i were tuned to establish a single 5% load increment. Next, the WPO server load was incremented up to the 50% level and back down. Any modifications needed in L_i parameters were made based on analysis results from these increasing and decreasing profiles. The system was then fully loaded and MaxLoad was determined. At any point up to MaxLoad the system performed within an acceptable PPO performance range. Beyond MaxLoad, PPO was not guaranteed. In each of these phase sub-steps PPO, preferred PPO, and CC/svr values were written to storage.

Analysis after phase one determined when PPO values degraded with increasing server load, and these analyzed values became the MaxPPO setpoint. The maximum preferred PPO (MaxPPPO) was also determined based on the results of the MaxLoad testing, and its value was at least 10% less than the regular PPO value. This 10% factor applied to MaxPPPO came from the Tamura et al. (2012) definition of a preferred customer having settings that resulted in a 10% preference over Regular users. The CC/svr value was sampled each minute, and was established as the mean of the last five samples before the system achieves MaxLoad. With the load simulator calibrated, and set points established in this first phase, the subsequent SAS RV&V experimentation proceeded. The following paragraphs detailed the steps of phase one.

This first measure to be established was the MinLoad value previously discussed. The MinLoad value was a measure of load with no user simulated web client traffic applied to the system. It was determined as follows. After an AWS Small instance was started with the WPO application running, five measures of quiescent system utilization were logged over a one hour period using the Unix sar command that accessed sysstat library data. One of the factors to be determined was whether I/O utilization, CPU utilization, or a combination of multiple factors drove the overall WPO performance; thus, using sar will allow phase one analysis to determine the overall drivers of system performance, whether they be cpu, iowait, or a combination of these times. Measures were taken in reference to the experiment start time at 10 minute increments, and the averaged value was established as MinLoad. This MinLoad value was determined to have a negligible value in baselining overall system performance.

Once MinLoad was established by identifying the CPU, I/O, or a combination of factors in quiescent operation, a single repeating Cycle of web transactions was introduced using the defaults shown in Table 11. The C_c, C^r, and C^a values were modified to achieve the first 5% L_i increment by increasing or decreasing C^a to flatten out load spikes. Secondly, C_c was increased if a single Cycle of repeating transactions did not produce enough load – CPU or I/O – to reach the 5% increment. The tuning of L_i was accomplished in 10 minute intervals. sar dumps were collected each minute and evaluated after each 10-minute interval to determine if further modifications need to be made to the C_c, C^r, or C^a values. The sar values of %user, %iowait, and %idle were the focus of analysis to determine the load increment values.

This initial value of L_i was now used to move the system to a load mid-point. The system was loaded with increasing L_i values to reach a 50% utilization level, and then back down to MinLoad over a one-hour period. In each 10-minute period the system was brought to 50% load at the five-minute point, held at 50% load for two minutes, and then back down to MinLoad. sar readings were taken every minute for subsequent analysis with the same focus on %user, %iowait, and %idle as previously described. C^r values would have been incremented to achieve a smooth, step transition in this step, if needed.

Table 11. Initial parameters for L_i

Parameter	Initial Value	Units
C_c	1	Scalar
C^r	0.0	Seconds
C^a	5.0	Seconds

Lastly, the system utilized L_i increments to fully load the single-server configuration over a one-hour period. Again, testing was broken into 10 minute increments. Load was immediately ramped to the 50% level and then brought up to a maximum utilization at the five-minute point, remain at maximum load for two minutes, and then be reduced to 50% at the end of the 10 minute period. sar data will collected each minute. The system did become unresponsive at some point below 100% cpu utilization due to locks or iowait times, and performance became non-linear above 50% CPU utilization. Therefore, the MaxLoad value was positioned to be a value just below where the server became unresponsive. Again, L_i parameters were adjusted after this step to achieve as smooth a step function to MaxLoad as possible. The output set points from this phase were listed in Table 12. The method by which they were determined was also discussed in further detail in the Analysis section.

Table 12. WPO load criteria

Measurement Set Three			
Metric	Description	Unit of Measure	Criteria
MinLoad	Steady state CPU utilization of an AWS small instance with Tomcat, and the WPO application running but no client connections	%	Quiescent load tests executed five times and a mean calculated using sar data output
L_i	Load increment determined by modifying the parameters in Table 11	Function of C^a,C^r, and C_c	Determined from a mean of five sar samples of the 5% load tests and then tuned in subsequent testing
MaxPPO	Maximum number of seconds that is system is permitted to process a purchase transaction for a regular customer	Seconds	Determined from a mean of five sar samples of the 50% - 100% load tests
MaxPPPO	Maximum number of seconds that the system is permitted to process a purchase transaction for a preferred customer	Seconds	Determined from a mean of five sar samples of the 50% - 100% load tests
MaxLoad	Maximum CPU utilization of an AWS small instance; 20 * L_i load is generated; Also, the point at which the server and WPO application become unresponsive may require MaxLoad to be tuned down from absolute maximum server load	%	Determined from a mean of five sar samples of the 50% - 100% load tests
CC/Svr	Number of concurrent connections to the WPO application at the time that MaxLoad is achieved	Scalar	Determined from transaction logs

Phase two experimentation

In this phase of experimentation three test cases were executed against the full three-server WPO architecture in two different steps. In the first step RV&V feedback behaviors were disabled and only SAS feedback actions occurred. The RV&V system performed data collection, but the Executor module suppressed RV&V execution commands. In the second step, RV&V feedback was enabled, and the same three test cases re-run. The test cases spanned the time periods defined in Table 6, and each test case was run five times. sar performance monitoring was also conducted on each WPO application at 10 minute intervals. Further detail on how the test run data was evaluated was described in the validation section.

This first step in the phase two experimentation was to produce baseline SAS data that was used in the determination of whether SAS RV&V behaviors reduced the cost of operating the WPO system, yet maintained expected performance. The feedback server was the first server started in the environment. The MaxPPO, MaxPPPO, and CC/svr values from phase one were added to the ExperimentalRepresentation, and a single WPO server was started. The simulation server was then started, and the Basic test case loaded. Based upon successful registration of all participants in the ServerManager, the environment was ready to execute test cases. As each WPO application server started, it logged its start time to a local file, and did the same when it shut down. These time reference points were used to determine the total WPO server up-time. The cumulative server up-time for all WPO servers in each test case was the baseline measure of performance for the QoS measure of Minimize Cost.

The first test case to be executed was the Basic test case. It executed for 24 hours of simulation time and was repeated five times. After each simulation run, WPO server start and stop times were be collected, and the database was queried for all relevant PPO performance data. If the feedback loop issued any commands to the ServerManager, those commands were also be logged. The expected behavior of the Basic test case was that no self-adaptive behavior should be exhibited as the system should not have experienced MaxLoad. PPO values were also collected, but since RV&V feedback is suppressed, no PPO-based transitions can occur. As noted above, server activation and deactivation times for each WPO server were also collected.

The second test to be executed was the SaleDay test case. Once initiated with the values derived from phase one, the ExperimentalRepresentation was not modified for any of the SAS test case runs, and RV&V feedback behaviors continued to be suppressed. The test case input to the simulation server was set to load the SaleDay content, and all servers were re-initialized in the same way as the Basic test case described above. The SaleDay test case was executed for 24 hours of simulation time, and then repeated five times. The SaleDay test case simulated a SAS transition from the Normal to the Medium state and two WPO servers were activated to address this SaleDay context change. Again, PPO values were collected, but no PPO-based transitions were possible in this SAS-only mode of operation. PPO violations were possible during this test case, as no RV&V feedback is enabled.

The third test to be executed was the SocialNetwork test case. Initialization was performed similarly to the previous two test case descriptions, and this 24-hour test case will be executed five times. The SocialNetwork test case mirrored the Basic test case, except for the SocialNetwork context change of a few hours. The self-adaptive feedback system enabled a second WPO application server for the SocialNetwork period, and the system transit from the Normal state to the Medium state. Once the SocialNetwork event concluded, the system eventually resumed a Normal state. It was again possible that PPO values were violated during the execution of this test, and these values were logged for comparison to the RV&V runs in the RV&V step.

In this last series of steps in experimentation, three of the four test cases were re-run with SAS RV&V execution commands enabled. At the beginning of each test case run the metric collection database was re-initialized, the ExperimentalRepresentation file was re-loaded, and the appropriate test case loaded into the SimulationManager. Initialization actions follow the same order as in the SAS steps, except that the global setting for SAS RV&V was enabled. The PPO values for each run were collected as well as server activation and deactivation times. Each test case was executed five times over the same test case periods described above. System behaviors were different from those behaviors recorded from the SAS-only experiments.

Validation

To determine if the experimentation demonstrated that the inclusion of a secondary SAS RV&V feedback loop reduced cloud costs while maintaining system performance, the following validation approach was proposed. The first task of validation was to analyze the output from the phase one experimentation, and populate the values required by Table 11. Next, load rates for each SAS test case execution, shown in Table(s) six through nine, were compared to the sar output for each of the five runs, and the mean of the five runs. The evaluation of sar data focused on %user, %iowait, and %idle values, seeking to determine if CPU, I/O, or a combination of these drivers dominate WPO load. This step was necessary to verify that proposed input was consistent with actual measured load. These results were presented in both tabular and graphical form. The processor utilization and PPO/PPPO times were also displayed compared to the time of each test case. These same values were then displayed for the SAS RV&V experiments. Last, a comparative summary of results was presented to show how SAS RV&V experiments performed versus SAS-only experiments.

The first phase one step was to establish MinLoad and this was accomplished by displaying the five sar outputs and an average of the %user, %iowait, and %idle results. The L_i increment was then initially established, and tuned over a 50% load. Finally tuning over a maximum load was also documented in tables. By using the maximum load experiments PPO, PPPO, MaxLoad, and CC/Svr were established and shown in tabular format.

The QoS goal of maximize throughput was applied to the metrics of PPO and PPPO. The secondary SAS RV&V goal was to minimize cost. Each of these goals used terms that communicated the slope or direction of the goal, but not a mathematical maximum or minimum. Thus, the highest goal named maximize QoS throughput informed the system to maintain PPO and PPPO. The secondary goal of minimize QoS cost attempted to reduce system cost. These two goals were clearly in conflict, and this experimentation demonstrated that the SAS RV&V feedback sub-system attempted to manage these competing goals.

PPO and PPPO values were extracted from the database by time, and displayed in tabular and graphical form. A mean value for each test case time period was tabulated. Notation was made of the number of times that the QoS performance metric was violated and an average duration of the violations per measurement period in each test case. With PPO and PPPO means calculated for each test case, the SAS versus SAS RV&V values were shown. It was expected that SAS-only tests resulted in few PPO violations, but that the SAS system will not react to these PPO violations. It was expected that the SAS RV&V system violated the PPO set points more often, but reacted more quickly to these violations and reduced their durations.

The next activity of validation was to compare SAS server utilization to SAS RV&V server utilization. Cloud providers have many different charge rates schemes for the time that a server instance was utilized, but server uptime formed the basis for all cost models. Therefore, the simple measure of server uptime determined cost for this experimentation. Cost savings were achieved when – for the same test case inputs – one configuration executed with less overall WPO server uptime. For each test case a measure of server uptime was established based on the start and stop times in the log files. Again, a mean was calculated for each test case and experimentation step – whether SAS or SAS RV&V. Then, the ratio of mean SAS RV&V uptime to the mean SAS uptime for each test case was calculated to determine if system costs are reduced. It was expected that no savings were achieved in the Basic test case, but it was possible that a PPO violation could occur and a second server would be added in the SAS RV&V test case runs. A SAS RV&V PPO violation that caused a second server activation will result in SAS-only cost savings over the SAS RV&V experiments. It was expected that SAS RV&V uptime for the SaleDay and SocialNetwork test cases demonstrated savings over SAS-only test case runs.

The combination of mean PPO measures and mean utilization measures by test case validated whether the SAS RV&V feedback system improved QoS performance over a SAS-only system. This determination required an analysis of both QoS performance measures over time. These results were again graphed and highlights were made where performance differs between SAS-only and SAS RV&V results. It was expected that the SAS-only system achieved the highest goal of maximize throughput, but failed to reduce costs over all test cases. The SAS RV&V system was expected to have more PPO performance violations, but to react and restore PPO more quickly, while executing test cases at an overall reduced cost. If these expectations hold, the system will have demonstrated that hierarchical goal-based SAS RV&V improves QoS performance over a single SAS feedback loop approach.

Resources

The resources shown in Table 13 were required to develop the WPO application, RV&V simulation, and conduct experiments for this research.

Table 13. Resource list

Resource	Description
Java language	Language utilized for WPO example and RV&V simulation
Apache Jena	Java Ontology Library
Eclipse platform	Development environment
Junit	Test library
Apache Wicket	Library used to develop the WPO application
Stanford Protégé Ontology developer	Ontology development tool
Windows desktop (development)	Development workstation
Linux Server (test)	Testing workstation for simulation
Ubuntu operating System	OS used for all experimentation
Apache Tomcat	Java web server
Apache Commons	HTTP client side library
Log4j	Logging library
VMWare	Virtual machine application
Amazon Web Services	Cloud services provider
MySQL	SQL Database

Summary

This experimentation quantitatively explored the value of expanding SAS with a second SAS RV&V feedback loop. The integration of QoS goals into the RV&V structure allowed for an implementation of an independent SAS RV&V documentation construct. The research effort began by developing a fully functioning WPO application in the AWS environment ("Amazon web services," 2013). This WPO application itself did not have SAS or SAS RV&V components. Baseline measurements were collected from this basic environment and then those measurements were used to instrument the simulation and feedback behaviors. The simulation was then employed in a SAS-only configuration with four test cases that execute the intent of the Tamura et al. (2012) example problem. The Basic 24-hour test case demonstrated normal, single-server behavior. The Sale Day test case forced the addition of a second server to the WPO configuration, and the Social Network test case demonstrated the temporary addition of a second server to the WPO configuration.

Each of the test cases described above were executed using simulated web clients in SAS-only and then SAS RV&V configurations. Both the SAS-only and SAS RV&V simulations utilized the addition of states, temporal logic, and goal-oriented behavior by expanding the Villegas et al. (2011) SmartContext taxonomy. The results of each SAS-only and SAS RV&V test case run were compared to determine if the introduction of SAS RV&V quantitatively improves goal achievement by reducing server costs, while maintaining QoS throughput goals. This research demonstrated that the integration of these additional components to the Villegas et al. (2011) SmartContext provided a generic method for documenting SAS RV&V systems.

Chapter 4

Results

Overview

SAS systems make it inherently difficult for users to establish trust in these architectures (Dahm, 2010; Tamura et al., 2012), and therefore a new construct to verify non-functional requirements was needed. SAS RV&V was proposed by Tamura et al. (2012) as a method to reduce the complexity of confirming that SAS systems can maintain their non-functional requirements. The problem statement also highlighted that a baseline reference model for SAS RV&V, like the one proposed by Tamura et al. (2012), was not available to the research community. This proposed baseline model (Tamura et al., 2012) also defined a series of test cases with which to evaluate SAS RV&V performance.

This dissertation provided a method to document SAS and SAS RV&V behavior by extending the Villegas et al. (2011) SmartContext taxonomy. It then quantitatively verified that this SAS RV&V monitoring method improved overall non-functional performance. The baseline reference model and test cases were also implemented so that the community can further extend this SAS RV&V experimentation.

The many different paths of execution that a SAS system may take requires that SAS RV&V requirements languages allow for a broad syntax of non-functional goal criteria, generalization in defining system limits, and a decoupling of design versus implementation. The Fu et al. (2007) QoS taxonomy provided the structure for a goal description language. Lapouchnian et al. (2005) proposed a positive and negative goal approach for autonomic systems, and this Lapouchnian et al. (2005) directedness was required by SAS RV&V so that the feedback loop can make tradeoffs between competing non-functional goals. The Fu et al. (2007) goals were then decorated with Maximize or Minimize qualifiers to guide the feedback subsystem. This dissertation quantitatively demonstrated that a SAS RV&V goal tree of directed, static rules improved the performance over SAS-only experiments.

The extension of the Villegas et al. (2011) SmartContext language was accomplished by adding states connected with temporal logic transitions and a hierarchy of directed, RV&V goals. These new language constructs allowed for the definition of SAS-only and SAS RV&V behaviors to be generically defined during design. The conversion of the temporal logic state machine into an executable feedback system was found to follow that of a non-deterministic, finite automata transformation (Sipser, 2006). New synthetic states were required to capture temporal logic transitions and establish deterministic feedback behavior. This transformation from the requirements representation to the operational representation is discussed further in the data analysis section.

A realistic baseline model was needed to test quantitative SAS RV&V behavior, and an implementation of the Tamura et al. (2012) web-based purchase order application in the Amazon cloud environment was employed. A tunable load generation application was also produced that simulated the WPO responses through three different test cases. The combination of these test case results showed that the implementation of SAS RV&V improved performance using a goal tree of RV&V rules.

This chapter will first analyze the baseline settings required to configure the web load generator and simulate the WPO application to move through a CPU load profile consistent with that of Figure 27. In order to establish these settings for stable SAS and SAS RV&V operation, key variable values were established. Since the WPO application are required to scale across multiple web servers in a cloud-based environment, verifying that a stable load generation profile was established was key to gathering quantitative metrics for measuring SAS RV&V performance.

Next, the Basic test case data was analyzed. The SAS-only test cases were evaluated to capture a mean of the best possible performance that the WPO application can deliver under single server load. Enabling competing goals resulted in degradation of some performance metrics compared to unconstrained SAS-only operation. Performance was analyzed for all Basic test cases by looking at the number of RV&V violations to identify where the highest goal of customer satisfaction is violated, and then server uptime was evaluated to determine the impact of the subordinate RV&V goal of minimizing cost.

The data analysis section then focused on the results of the Social Network test case results. In this test case SAS-only behavior was compared to SAS RV&V behavior for a short-term load increase caused by a social networking announcement. The evaluation of these data determined whether SAS RV&V introduction maintained customer satisfaction yet reduced cloud-server costs when load increases through a temporary spike in purchases.

The Sale Day test case was then analyzed to evaluate the performance differences when the normal load profile was augmented by an extended sale period. SAS-only behavior established values for RV&V violations, purchases, and server uptime hours. With the introduction of SAS RV&V, Sale Day results were evaluated to determine the mean of each of the previously mentioned performance criteria. A comparison of cost savings and overall performance was then provided.

The last data analysis section provided a description of the transformation required for a temporal logic state machine defined in Figure 20 to be utilized as the SAS RV&V feedback loop. It was determined that the use of temporal logic required additional states that the system must implement for both the SAS and SAS RV&V feedback system to meet the requirements defined in the SmartContext extensions (Villegas et al., 2011).

The chapter concludes with a summary of the SAS-only and SAS RV&V performance results to show that the RV&V goal tree, integrated in a temporal logic state machine, does improve the performance of the WPO reference application.

Baseline Data Analysis

The Phase One effort described in the Methodology section outlined the tasks necessary to establish web load to roughly conform to the curve shown in Figure 26. A number of observations were made during this phase of experimentation that resulted in minor modifications to the experimental architecture. First, the use of AWS Elastic Load Balancer ("Amazon elastic load balancing," 2013) introduced an unwanted side effect. Sporadic load behavior was observed because the load balancer forced TCP connections to Tomcat instances to implement HTTP Pipelining. Pipelining defeated the web load client approach of creating new connections to simulate a smooth load curve. Figure 27 below shows that the elastic load balancer introduced staccato behavior because the load balancer was continually attempting to combine new transactions with existing connections. The elastic load balancer was then removed from the experiment and the web load generator was modified to perform software load balancing internally. After simulating the load balancer in the web client, L_i increments were achieved in a more compressed form than was predicted.

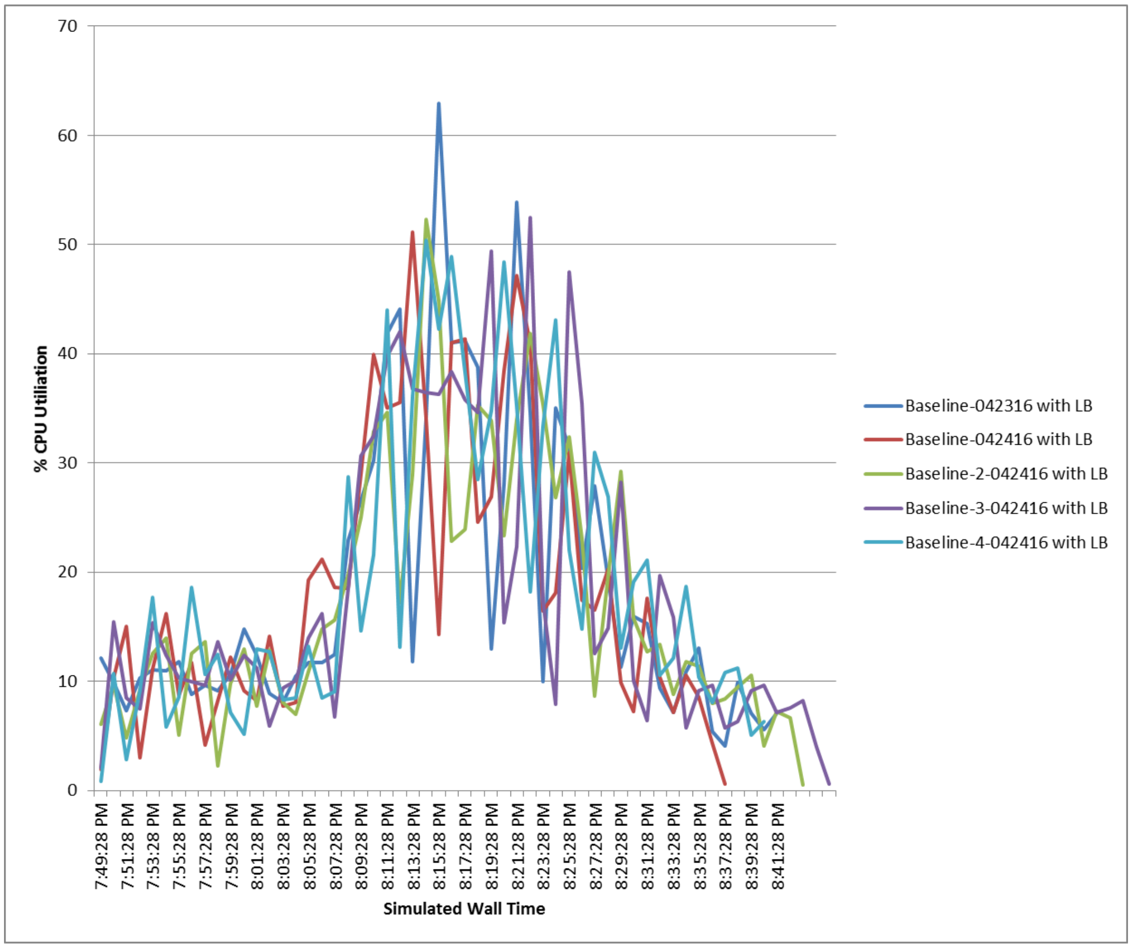


Figure 27. Basic test case output with load balancer

Web load peaks at the times proposed, but the overall increments compressed as load increased, especially beyond 50% load. The mean curve of Figure 28 more closely aligns with the Rosenstien (2000) curve of Figure 26, and there are no staccato movements. The curve shown in Figure 29 also shows that web load changes in amplitude are more gradual due to residual load from past transactions.

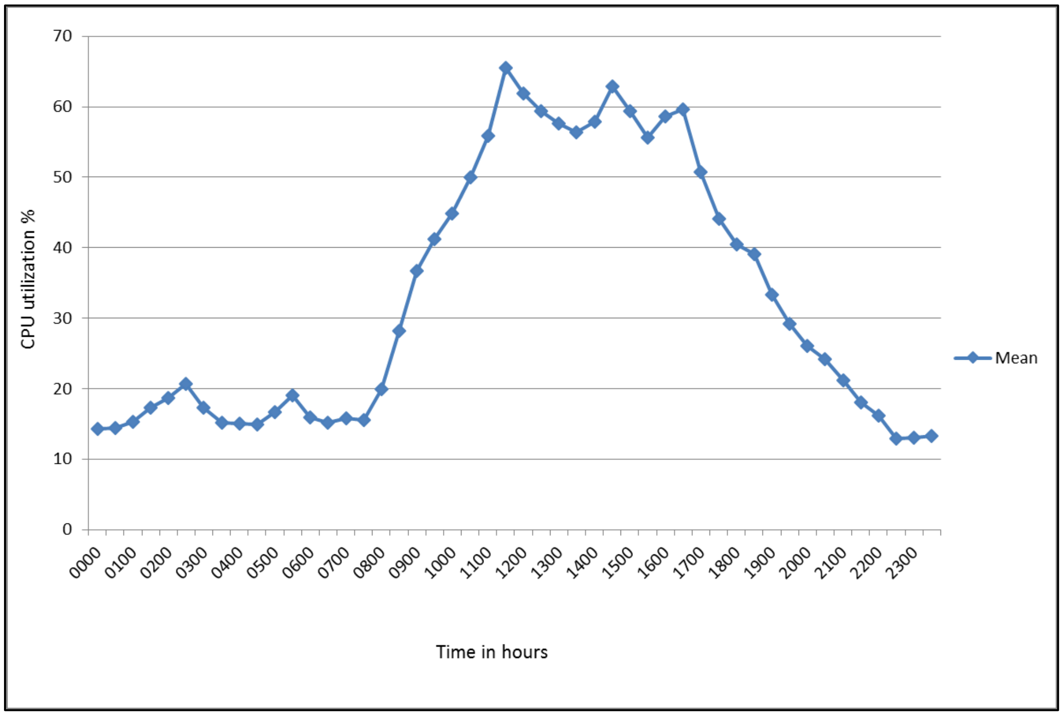


Figure 28. Basic SAS output mean

To achieve the curve in Figure 29, the load parameter settings shown in Table 14 were established. By executing the Basic test case and repeatedly increasing or decreasing values shown in the table below, a stable movement through the test cases was achieved without HTTP Client Protocol Exceptions, I/O Exceptions, or Client I/O Exceptions. The use of the Cr setting to ramp into a Li transition proved unnecessary and was left at the default value of 1. Movement of Cc or Ca to values greater than those shown in Table 14 resulted in the errors previously mentioned.

Table 14. Baseline web load settings

Parameter	Setting
C_c	2
C^a	4
C^r	1

Using the web load settings shown above, PPO load values were measured over five runs of the Basic test case and are shown in Table 15. The maximum value for RegularPPO value of all runs was 62 seconds under nominal load. The SocialNetwork and SaleDay test cases were expected to greatly increase PPO transaction times over unloaded test cases. Thus, the setting selected for Regular MaxPPO was 5% greater than the maximum measured value, or 66 seconds. The Preferred MaxPPO was then set to a value of 10% less than the Regular MaxPPO, or 60 seconds.

Table 15. Baseline settings for Regular and Preferred MaxPPO

Run	Preferred Avg Second Count Max	Regular Avg Second Count Max
1	52	47
2	46	60
3	51	47
4	49.5	62
5	51	48
MAX()	52	62
MAXPPO	**60**	**66**

The maximum CPU value predicted in the methodology section of 90% was not achieved, but the load curve of Figure 29 did provide a reference load necessary to conduct the phase two test cases. Like the Amazon Elastic Load Balancer (2013), the Tomcat server also attempts to pipeline all HTTP connections and this behavior degrades the use of new connections to linearly increase load. Maximum CPU utilization varied between 63% and 70% in the Basic test cases. Because of the pipelining behavior a different metric had to be chosen to replace the CC/Svr metric that was proposed in the Methodology. CC/Svr did not provide a measure of server utilization on each server because of pipelining, and a replacement metric was required. By using the %idle value of SAR output, a stable metric that combines CPU utilization, I/O utilization, and operating system waits properly instrumented the RV&V rules. Minimum CPU averaged less than 1% and proved negligible throughout the experimentation.

Basic Test Case Data Analysis

The Basic test case provided a worst case analysis for SAS RV&V behavior. The SAS-only performance never loaded the WPO application sufficiently to transition to a second server, and therefore SAS RV&V behavior could only provide poorer performance than the SAS-only test cases. The feedback loop was active during test case execution, but no SAS RV&V transitions were activated for this test case. SAS-only CPU utilization over the 24 hour period is shown in Figure 29 below. CPU load performance was consistent across all five test cases. Similarly, server uptime hours were exactly the same across all five test cases at 24 hours as shown in Table 16 below. Even without server transitions RV&V violations do occur in the experiment, as would be expected in any web application due to network latency and cloud-server constraints applied by the cloud provider. Maximum CPU utilization was consistent across all test cases, and transaction counts also showed consistent performance at a mean of 718 transactions. These results provide a best-case view of the WPO application with competition between the goals of minimizing cost versus maintaining

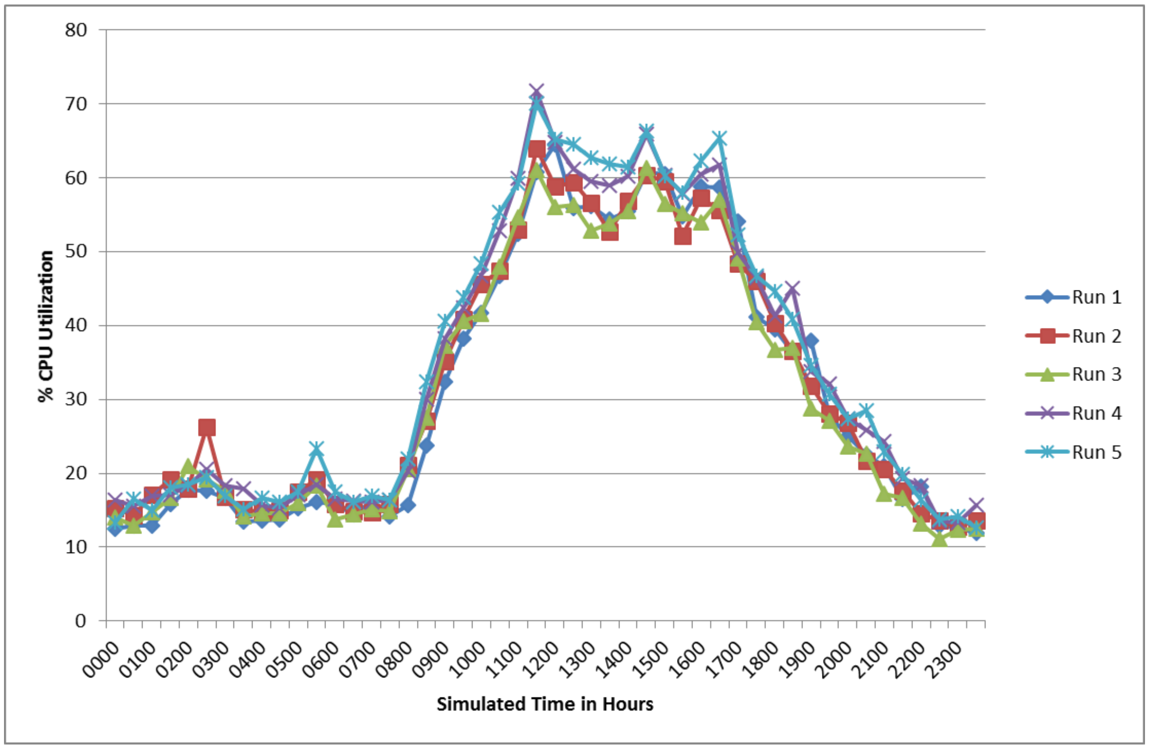


Figure 29. Basic SAS-only test case runs

customer satisfaction.

The same five test cases were then executed with the SAS RV&V feedback loop

Table 16. Basic SAS-only test case statistics

	Basic Test Case Runs					
	1	2	3	4	5	Mean
MaxTransSecCnt	63	63	60	61	65	62
RV&VViolations	10	9	9	11	12	10
MaxCpuUtilization	64	64	61	61	70	64
Server Uptime (Hrs)	24	24	24	24	24	24
Transaction Count	724	737	700	718	712	718

activated. While there were still no server transition rules active, the SAS RV&V feedback loop activated when SAS RV&V MaxPPO violations occurred. Note that this research implements static SAS RV&V rules that did not learn or derive heuristics to minimize the impact of periodic PPO violations. Any PPO violation that occurred twice, in consecutive samplings, triggered a PPO violation.

The approach of setting static limits for rule violation turned out to be the most significant weakness in the experimentation. For example, the consecutive MaxPPO violation count of two was established in the ExperimentalRepresentation, not at runtime. If the feedback sub-system were permitted to establish this value during SAS-only operation, it would have noted that MaxPPO violations do tend to occur in clusters, likely due to concentrated I/O periods that are a natural side effect of a magnetic disk backing store. While SAS RV&V must always have a starting point, the collection of runtime behavior within a viability zone would have better established the MaxPPO limit likely eliminated the performance degradation in this Basic test case set of experiments. PPO violations during the activation and deactivation of the second server also demonstrated the quandary of determining what a valid period of settling time was. Again, this settling time period was statically established in the representation, but could have been better determined by monitoring server four activations. This probably would have resulted in suppression of server four activations in the subsequent test cases. Feedback memory is a topic for future research.

Table 17 below shows the impact of two additional PPO violations over the five test cases. Consecutive RV&V violations in test case one and five resulted in server four activation and a 1% average increase in server uptime over SAS-only performance. The impact of these server four activations was amplified because a two-minute period simulates an hour of wall time. The server four activations actually had no impact on transactional performance as the overall average transaction count actually increased from 718 to 723 with SAS RV&V feedback. Thus, the activation of a second server did not leave transactions in an orphaned state during server activation in this test case. The slight increase in RV&V violations can be accounted for by the activation of server four in test cases one and five, but the maximum transaction second count mean remained exactly the same.

Table 17. Basic SAS RV&V test case statistics

	Basic RV&V Test Case Runs					
	1	2	3	4	5	Mean
MaxTransSecCnt	65	61	58	63	65	62
RV&VViolations	17	13	10	9	12	12
MaxCpuUtilization	133	65	76	77	151	100
Server Uptime (Hrs)	24.5	24	24	24	24.5	24
Transaction Count	709	729	734	724	718	723
Savings	-2%	0%	0%	0%	-2%	-1%

Like static rules, the expansion of simulation time would likely have resulted in better performance for this worst-case test case. Because a one-day test period was simulated by 24 two-minute periods, the impact of server activations was greatly amplified. While the two-minute period was selected so that data collection from all test cases could be accomplished in 30 hours, the expansion of the simulated time of one hour to even five minutes would have drawn down the impact of a second server activation significantly.

Based on these results it can be concluded that the activation of SAS RV&V feedback on the Basic test cases had only a minimal negative performance impact, and one that is likely greatly amplified by the simulation aspects of testing. Figure 30 below showed the impact of the two, server four activations on combined CPU utilization. The two spikes are largely anomalies accounting for the startup of the AWS instance. Each instance was deactivated utilizing the RV&V server shutdown rules as soon as the settling time setting had expired after an RV&V decision was taken. It should be noted that the tuning of settling time – either manually or through a learning algorithm – would have increased the positive impact of SAS RV&V rule activation in every test case. Both activations do correlate to the two peaks in the test case where load should be at its highest level. The SAS RV&V feedback loop had no knowledge of CPU load for server activation, but the MaxPPO readings do directly correlate to these local maxima at 1200 and 1600.

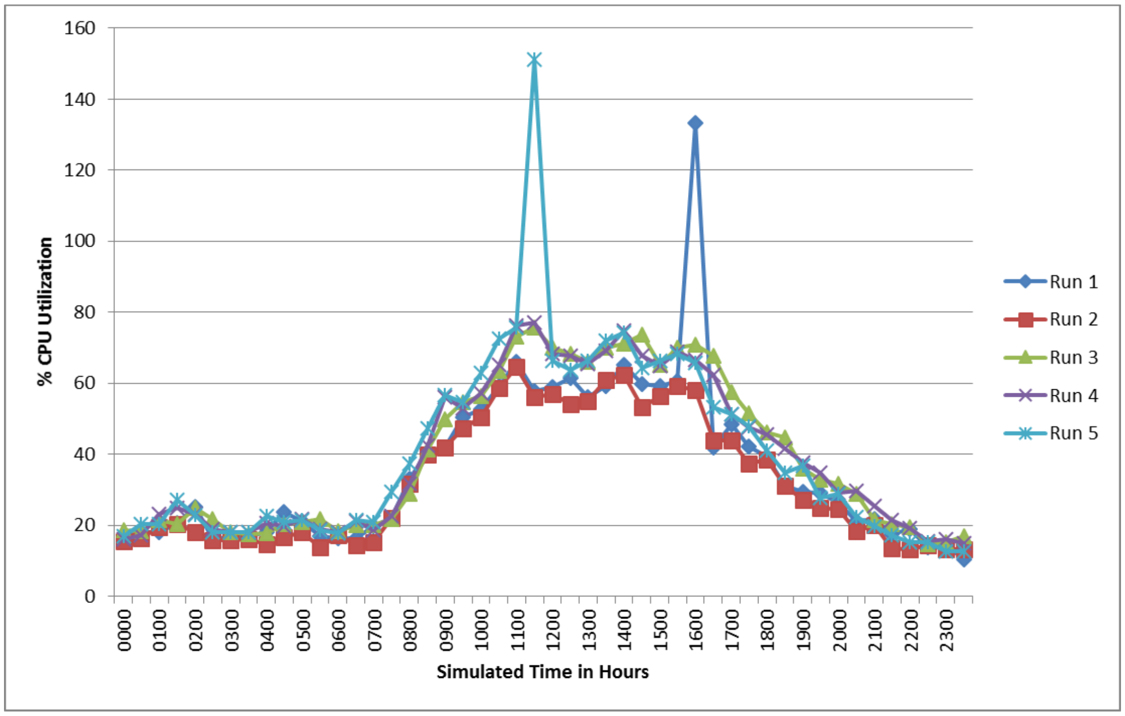


Figure 30. Basic test case SAS RV&V combined CPU utilization

The Basic test case data demonstrated that the addition of SAS RV&V feedback had only a minimal negative server uptime impact. Because the Basic test case with SAS-only behavior contained no SAS server activations, it can be considered the worst case on which to measure the impact of SAS RV&V performance. The number of transactions processed increased, and the maximum processing time for any transaction was consistent between SAS-only and SAS RV&V test cases. The impact of SAS RV&V server activations was likely amplified by the shortening of a simulation hour to two minutes because in actual clock time the second server would deactivate far more quickly than a half hour, which is the smallest unit of measure in simulation time.

Social Network Test Case Data Analysis

The Social Network test case was the first to exercise SAS-only, multi-server operation. It also provided insight into the short-term activation impacts of SAS RV&V behavior. The Social Network test case activated a server for a short duration prior to the start of a sale, triggered by advertisements on a social networking site. In the SAS-only test cases, server activation was triggered at a set time. In the SAS RV&V test cases, server activation was retarded until PPO violations triggered activation.

The Social Network SAS-only CPU performance is shown in Figures 31, 32, and 33. Figure 31 displays the primary server performance, and Figure 32 shows the activation of server four during the social network event. Figure 33 shows the combined CPU utilization

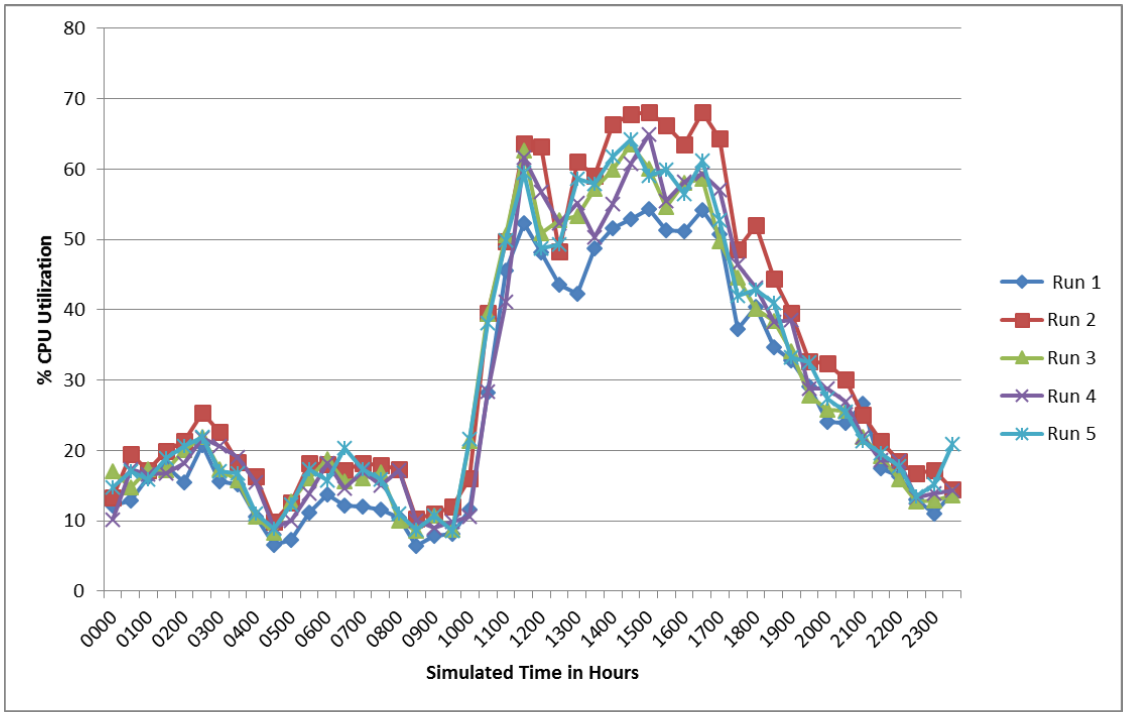


Figure 31. Social Network test case SAS-only CPU performance for server three

of both servers. A similar overall CPU performance curve is observed in the Social Network test case as that of the Basic test case, except for the social network event where server four is activated as shown in Figure 32. The server four performance curves are also very consistent, showing a spike in user load at 1200 and then a rapid degradation of server utilization until the server is deactivated at approximately 1500.

The combined CPU performance shown in Figure 33 demonstrates that server load was properly balanced across two servers even over the short period of the simulation, with a mean maximum CPU utilization for the SAS-only test case being 120%. Since the single

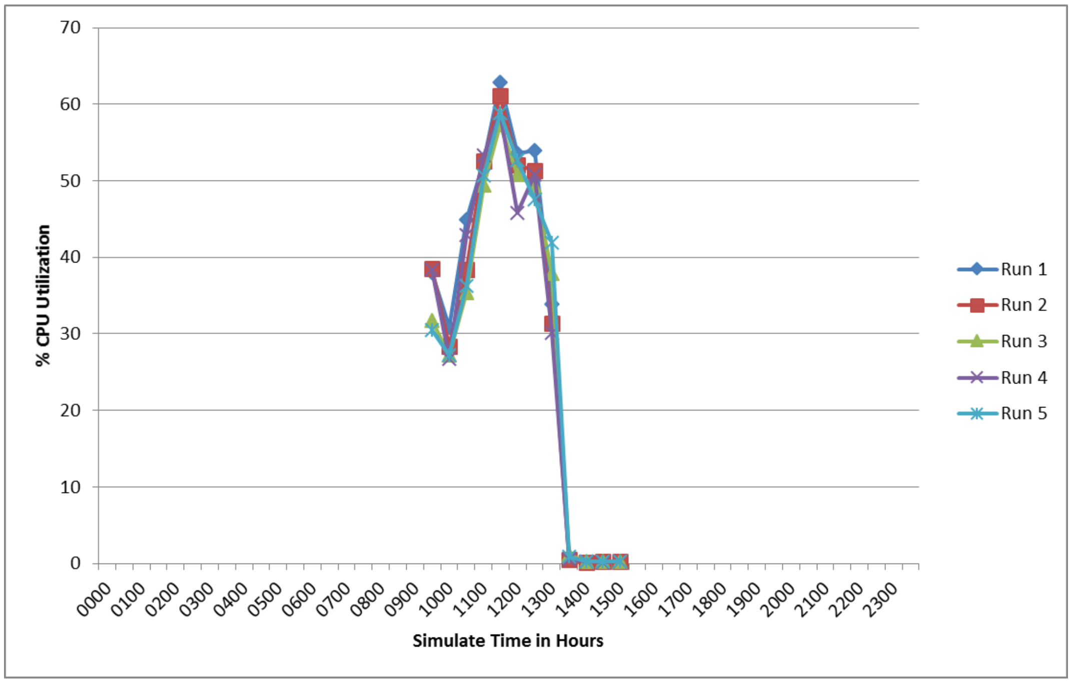


Figure 32. Social Network test case SAS-only CPU performance for server four

server baseline fluctuated between 60-70%, this result shows that the SAS-only simulation tracked very closely to expected WPO system performance. The period of the social network sale event can be clearly determined by the CPU load curve.

The overall performance of the SAS-only test cases is displayed in Table 18 below. Over the same 24-hour simulation period only two additional RV&V violations occurred over the Basic test case, and this result again showed that server transitions had a minimal impact in creating orphaned transactions in the simulation at this load value. The maximum

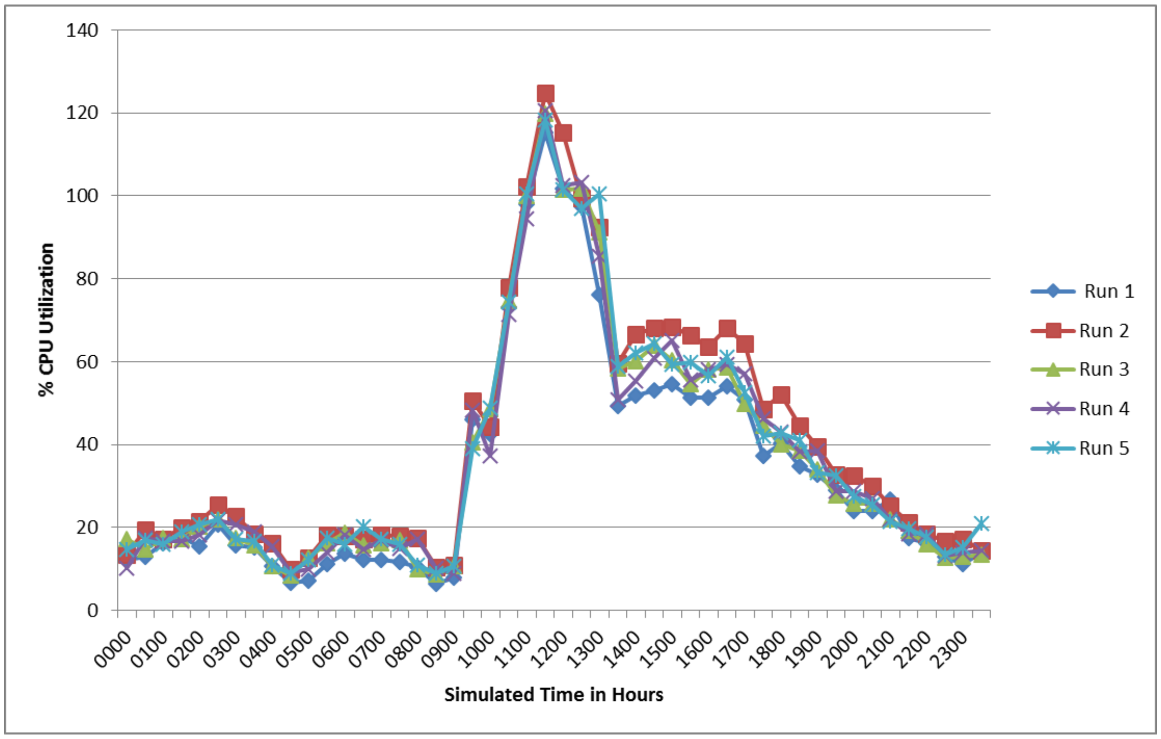


Figure 33. Social Network test case SAS-only CPU performance combined

transaction times for each test case did increase and demonstrate that the system was fully loaded during the spike shown in Figure 33. Server uptime – shown in the graph of

Table 18. Social network SAS-only test case statistics

	Social Network Sas Test Case Runs					
	1	2	3	4	5	Mean
MaxTransSecCnt	67	76	78	80	90	78
RV&VViolations	20	15	12	14	10	14
MaxCpuUtilization	115.2	124.82	119.85	120.52	118.32	120
Server Uptime (Hrs)	30	30	30	30	30	30
Transaction Count	790	719	759	796	751	763

Figure 34 – was consistent as the combination of the 24-hour period of server three and the six hour period of server four. It was clearly demonstrated that the SAS-only performance

reacted only to the social network sale event, and that CPU load had no impact on server uptime decisions.

In contrast to SAS-only Social Network performance the SAS RV&V test case results demonstrated the first benefit of the RV&V feedback loop behavior. Figure 35 demonstrated a substantially different CPU utilization curve for server three from that of server three in

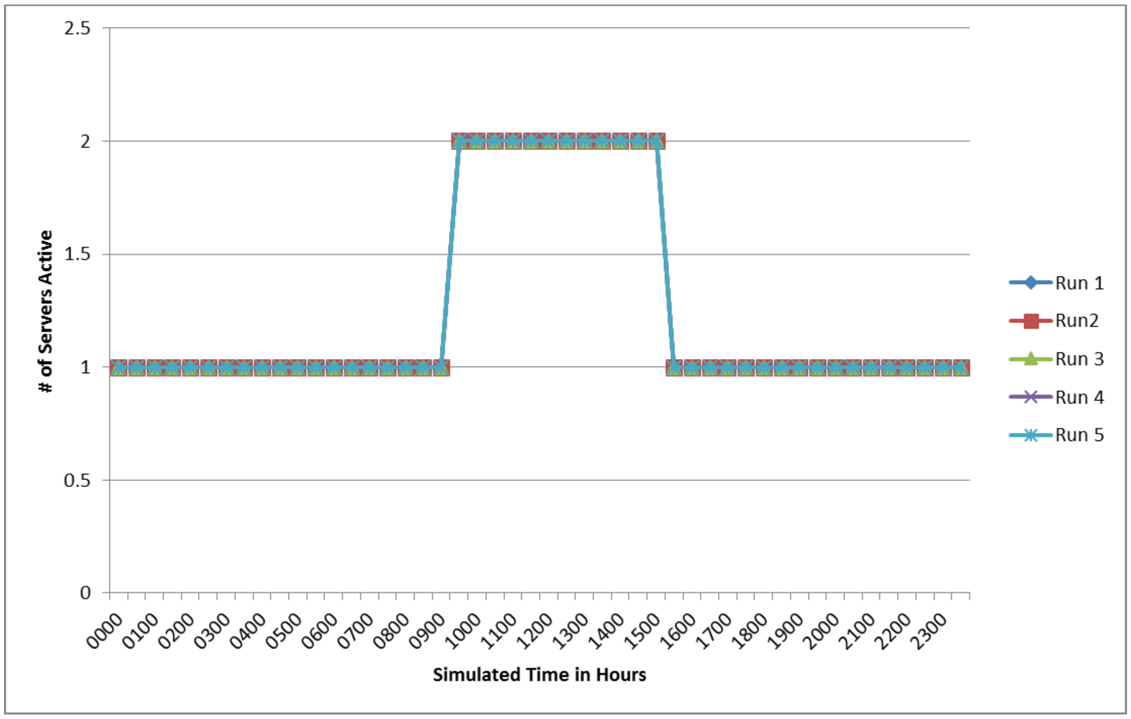


Figure 34. Social Network test case SAS-only server utilization

Figure 31. SAS RV&V feedback had a definite impact on server utilization, as server utilization at the peak point of 1200 hours was actually a trough compared to the SAS-only test case. The addition of server four and server five CPU utilization showed an actual increase in total CPU utilization for the system.

In Figure 36 below, server four CPU performance showed how the server three trough was supplanted with server four transactions. All server four and server five activations began with very high CPU utilization as basic operating system services consumed the CPU until server initiation processes quiesced. Comparing each test case run to its peers, performance was very consistent. The activation of server four was delayed more than an hour in every test case, but deactivation occurred around 1500 for both SAS-only and SAS RV&V tests.

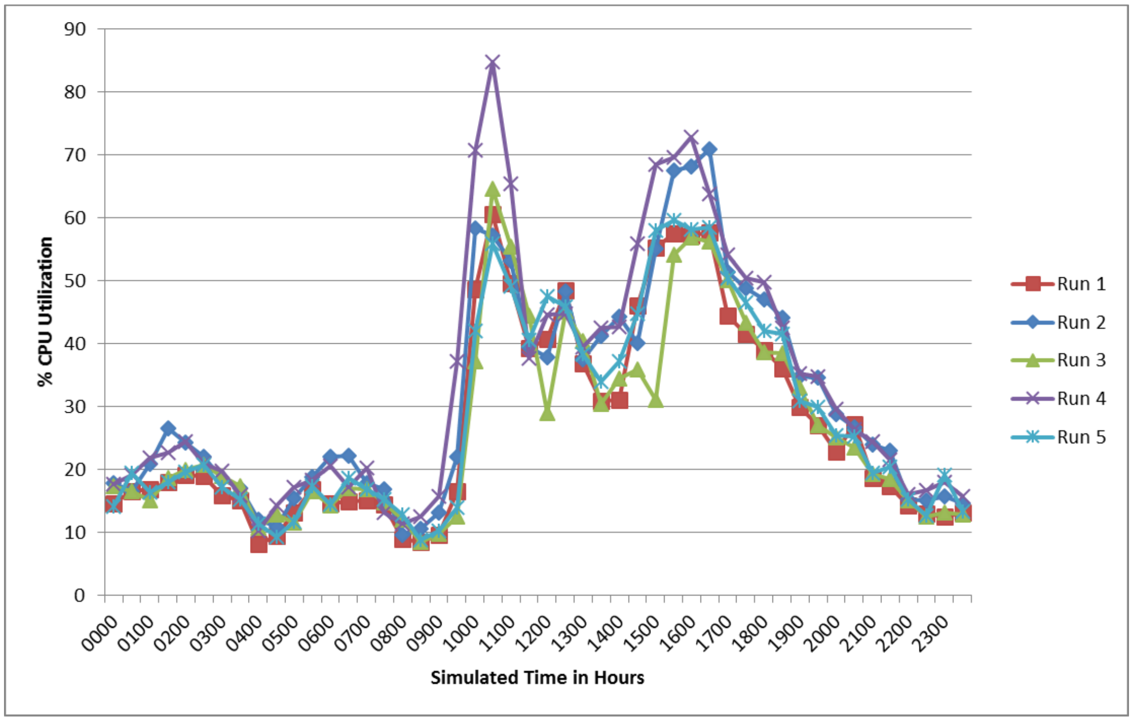


Figure 35. Social Network test case SAS RV&V CPU performance for server three

The weakness of SAS RV&V static rules was also demonstrated with this test case as consecutive RV&V violations were not tuned during run time to address periodic performance failures that should not have triggered RV&V intervention. Figure 37 shows that the Social Network test case triggered SAS RV&V activation of server five in every test case due to PPO violations; however, server five activation has no positive impact on transactional performance. As soon as the settling time clock expired, server five was deactivated as total CPU utilization did not require a three-server configuration. Based on a measurement of failed transactions comparing SAS-only versus SAS RV&V transaction counts, server five deactivation caused transactions to be aborted because the server was available for such a limited time. In only run two was server five active for more than a

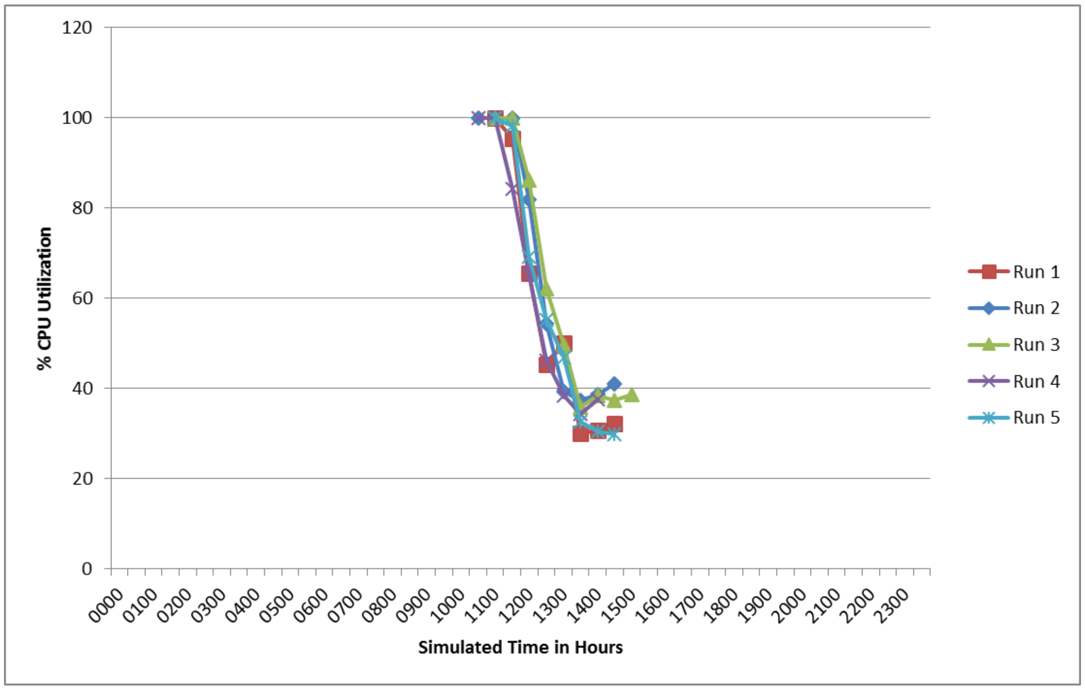


Figure 36. Social Network test case SAS RV&V CPU performance for server four

single measurement cycle of a half an hour.

The combination of all three server CPU performance graphs is shown in Figure 38 below. This figure clearly shows that the social network sales event does drive server performance to the limit of two-server operation. Except for one outlier spike in run two, all five test cases show a consistent single spike during the social network sale event, with the rest of the graph matching baseline performance. SAS RV&V combined performance shows that the entire system exhibited higher CPU utilization in spikes and in the overall area under the curve compared to SAS-only performance shown in Figure 33. This increased CPU utilization is likely due to the increased number of server transitions. No other WPO behavior should have been modified on each server due to the implementation of SAS RV&V feedback. It should be noted that the increased CPU utilization had no impact on server uptime as a cloud-based server cost was incurred no matter whether the server was fully utilized as they were with SAS RV&V test cases, or incurred more idle time with SAS-only

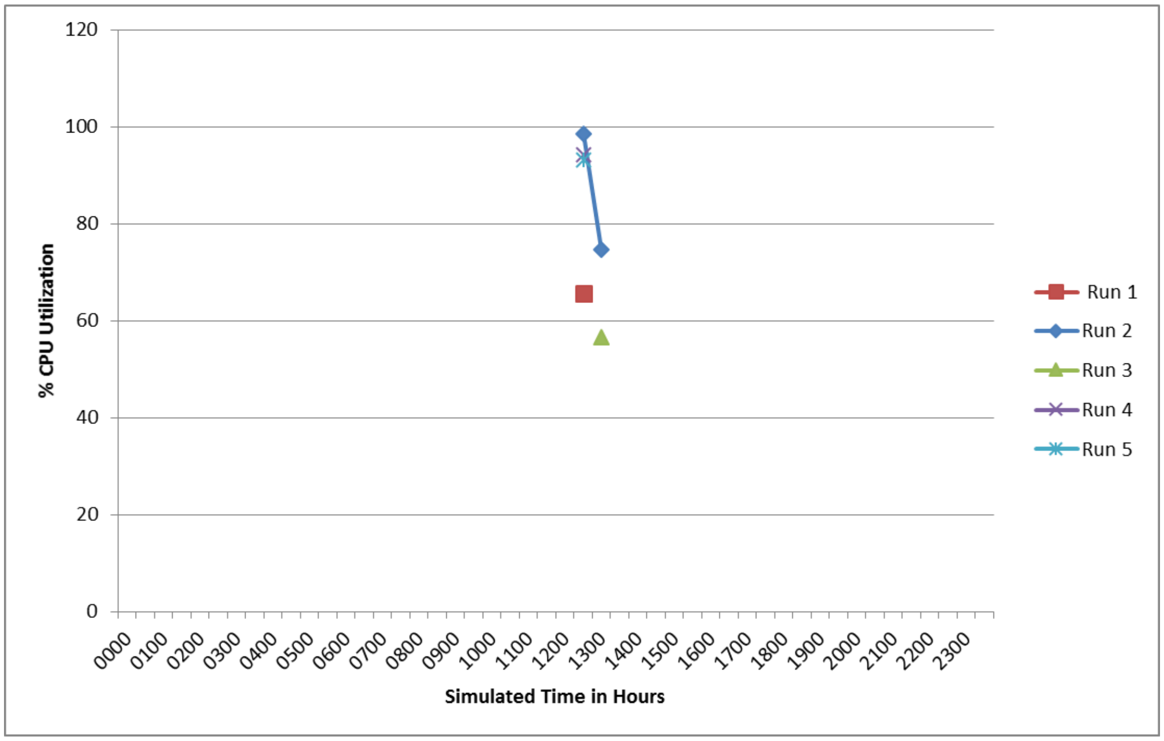


Figure 37. Social Network test case SAS RV&V CPU performance for server five test cases.

Competing SAS RV&V goals in a hierarchy where maximizing customer satisfaction measured by PPO responsiveness was a higher-level goal than minimizing cost was expected to reduce transactional performance in some manner. The feedback loop with competing goals must attempt to balance cost minimization with maximizing user satisfaction. This competition was demonstrated in the results of Table 19. The implementation of SAS RV&V reduced server costs by an average of 4% over SAS-only test cases. Transaction counts were also reduced by 7%. These two results demonstrated the competing behavior of maximizing transactional performance versus minimizing server uptime costs. RV&V violations were comparable with an average of 14 for SAS-only behavior versus a mean of 16 for SAS RV&V. Thus, RV&V violations remained consistent across both feedback approaches; however the SAS RV&V feedback loop attempted to reduce these violations by spinning up new servers. The major change between the two profiles was that the maximum

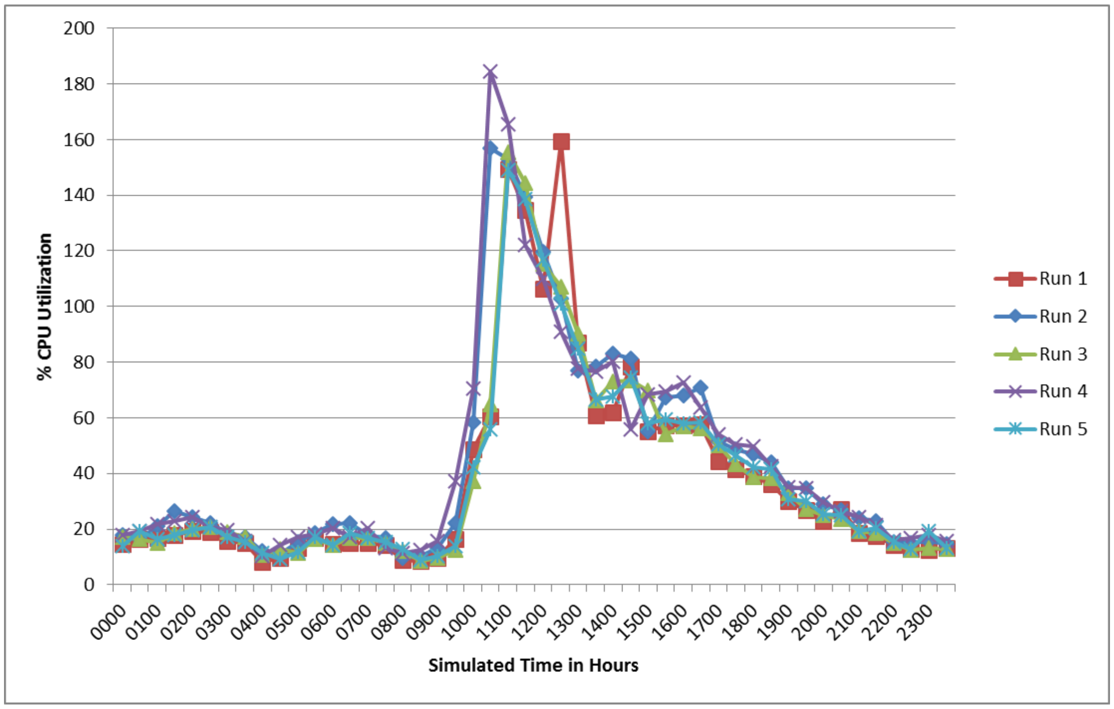


Figure 38. Social Network test case SAS RV&V CPU performance combined

transaction second count went up by almost 100 seconds. This is likely due to the activation and deactivation impacts of server five for such a limited time. While this maximum transaction second count was a negative metric, it accounted for only a single reading that mapped to server five deactivation.

The last data analysis artifact in this section was Figure 39 where the mean values of server uptime were compared. This graph shows that SAS RV&V narrows the uptime of server four substantially, and that cost savings would have been greater had not server five been activated for this short period due to static SAS RV&V rules. The SAS RV&V mean

Table 19. Social network SAS RV&V test case statistics

	Social Network RV&V Test Case Runs					
	1	2	3	4	5	Mean
MaxTransSecCnt	141	210	203	174	127	171
RV&VViolations	24	9	18	17	10	16
MaxCpuUtilization	159.41	157.05	155.3	184.59	149.09	161
Server Uptime (Hrs)	28.5	29.5	29	28.5	28.5	29
Transaction Count	719	667	659	740	773	712
Savings	5%	2%	3%	5%	5%	4%

clearly demonstrated that server activation was delayed by an hour and half. Deactivation also demonstrated one of the negative impacts of cloud-based server behavior. Cloud-server deactivation often took far longer than activation as shown by the SAS RV&V mean line. SAS RV&V attempted to deactivate an hour before SAS-only performance, but both lines meet at 1600. This same deactivation delay was shown between 1300 and 1400 for server five where the graph did not demonstrate linear deactivation.

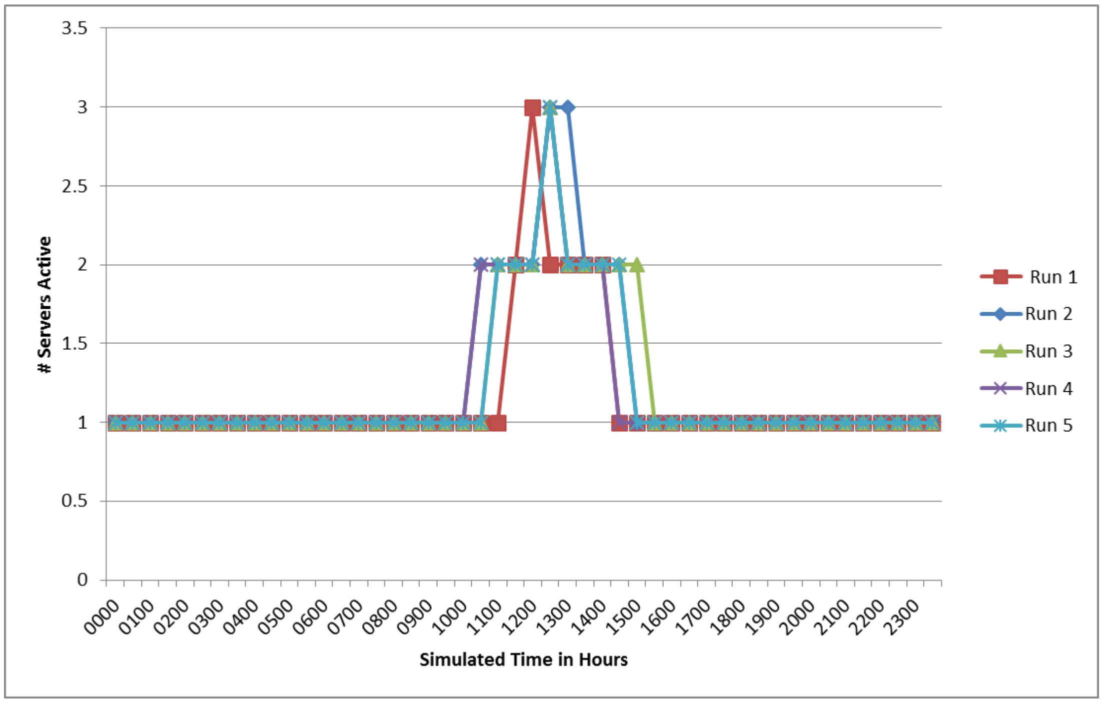


Figure 39. Social network server uptime mean comparison

Social Network sale data analysis demonstrated the conflict between competing SAS RV&V goals, where PPO violations were addressed by spinning up new servers, while the SAS RV&V cost savings rules attempted to retard server activations until customer performance was impacted. SAS RV&V demonstrated a 4% cost savings over a limited multi-server activation period. Total transactions decreased by 7% due to the short activation time of server five that likely caused aborted transactions. Server logs showed substantial transaction aborts at the same time that server five began termination. Overall, the Social Network test case showed that SAS RV&V behavior did improve cost savings with only a nominal increase of RV&V violations at 2% between SAS-only and SAS RV&V feedback.

Sale Day Test Case Data Analysis

The Sale Day test cases demonstrated a more realistic comparison between SAS-only and SAS RV&V behavior over a full day, sale event. Like the Social Network test case SAS-only behavior activated server four, but for a full eight-hour period. The SAS-only system did not pay attention to RV&V violations or cost minimization. Evaluation of SAS RV&V feedback for the Sale Day test case expanded on the results of the Social Network test case in the positive and negative impacts of a fixed set of SAS RV&V rules over an extended time period. Like the previous data analysis sections, CPU performance of each server was analyzed and then a comparison of SAS-only versus SAS RV&V performance was provided. In Figure 40 below, the SAS-only server-three graph showed the same trough created in the Social Network test cases when a second server was activated to share the highest load. For the first time the basic shape of the server three curve was no longer distinguishable because server four played an active part in balancing overall load. It should also be noted that this graph showed more variability between test cases as server three performance varied between each sample data point because two servers were balancing the load for the majority of the test cases.

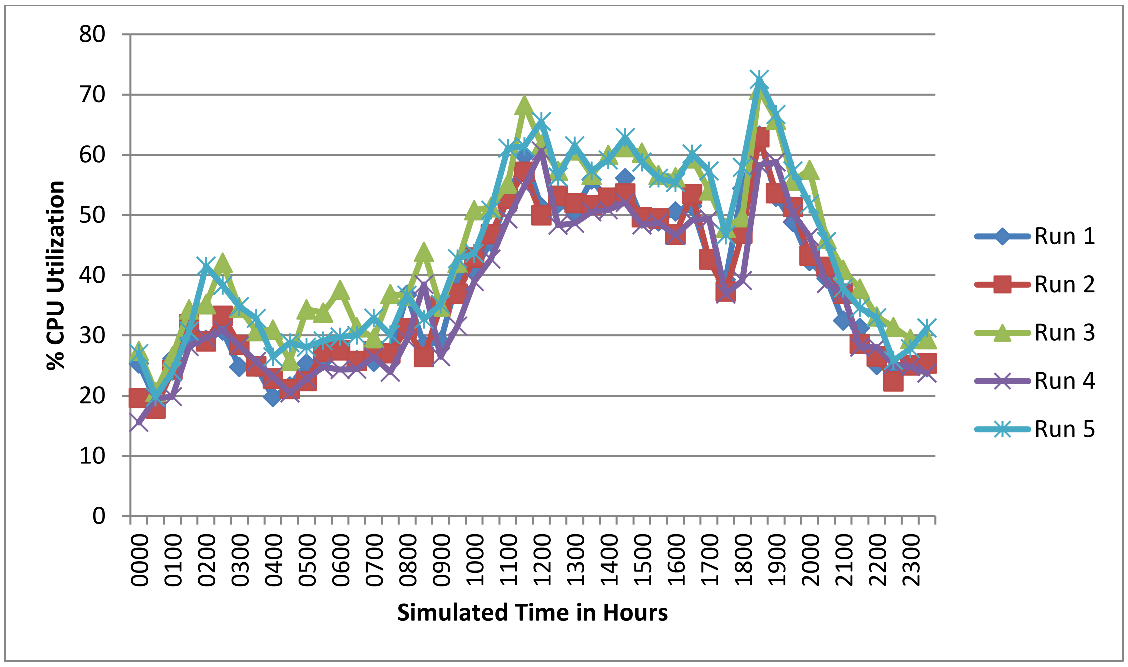


Figure 40. Sale Day test case SAS-only CPU performance for server three

Figure 41 shows the accompanying server four performance curve for SAS-only behavior. Two characteristics of this curve deserve note. First, all five test case runs show that the CPU utilization during server initialization begins at a maximum level and rapidly decrease to a stable state around 50% utilization. This observation demonstrated that the load balancing algorithm of the web client was appropriately balancing load in a stable manner across the two servers. The second notable item in this graph was that it took almost two simulation hours for server four to deactivate further reinforcing that server shutdown in AWS environments was much less predictable than server activations. There was really no other significant variability between the five test case runs for server four behaviors.

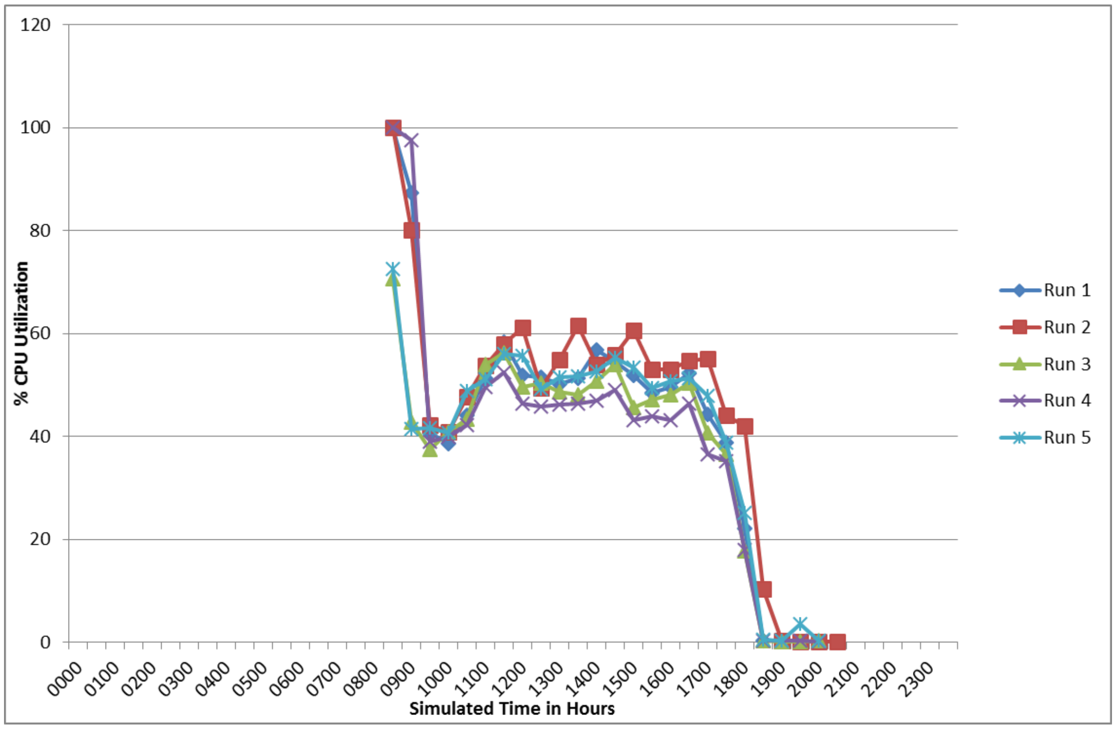


Figure 41. Sale Day test case SAS-only CPU performance for server four

The combined graph of SAS-only Sale Day test case CPU performance is shown in Figure 42. It shows that the system started to execute a normal Basic test case curve when the Sale Day event activated the second server and load increased almost vertically around 0800. Then, a stable state was achieved at about 110% CPU utilization. There was a consistent trough in CPU performance at about 1000 hours that can't be accounted for in the test case, but it was very consistent across all test cases. At about 1800 conformance to the Basic test case curve resumed and all test cases conformed to expected behaviors.

The summary statistics from the SAS-only, Sale Day test case execution are shown in Table 20. RV&V violations remained consistent with those of the Social Network SAS-only

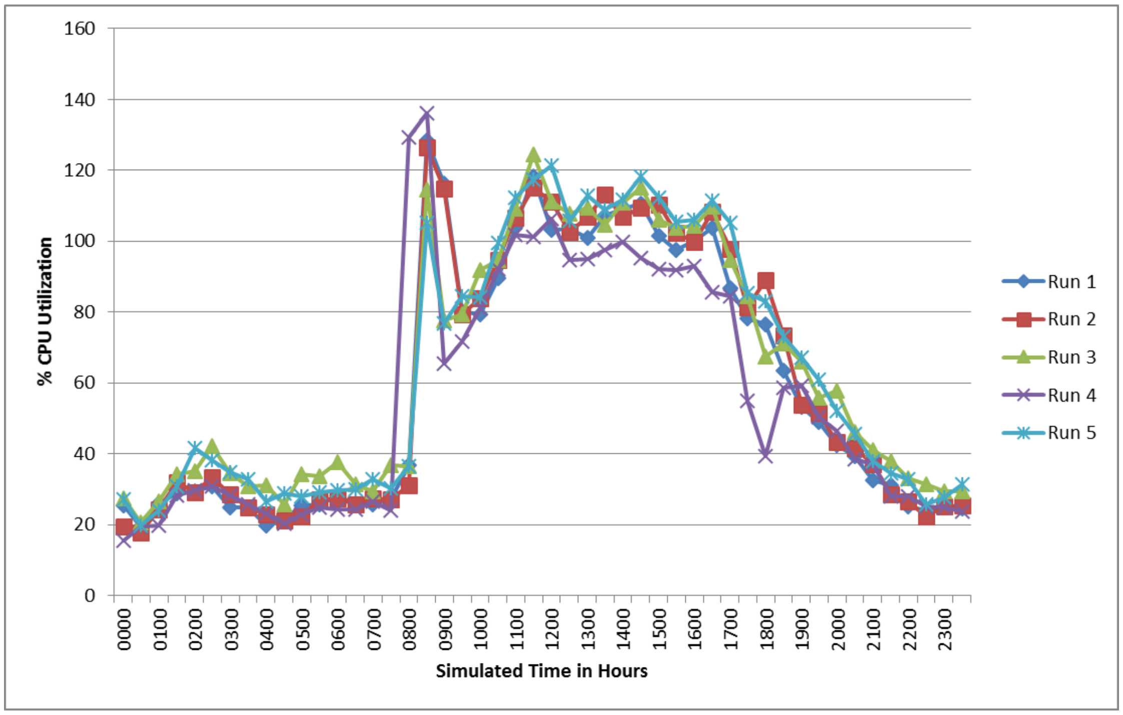


Figure 42. Sale Day test case SAS-only CPU performance combined

and SAS RV&V results, but total transactions were increased from 763 to 1241. This increased transactional performance was due to the increased simulation load, but also because server four was active for over a third of the total test case duration. The server uptime hours peaked for all SAS-only test cases at 36 hours because of the addition of 12 hours allocated to server four.

SAS RV&V results demonstrated the significant contrast in performance by the inclusion of a second-level feedback loop that enforced competition between maximizing customer satisfaction and minimizing costs. These test cases clearly showed that even static SAS RV&V feedback decreased costs with increasing load and maintained RV&V violations at a nominal level throughout the test case runs. In Figure 43 the server three graph showed that server four and server five do augment load above the maximum CPU value of server three shown in the Basic test case runs. CPU volatility was again greater as the overall stress

Table 20. Sale Day SAS-only test case statistics

	Sale Day Sas Test Case Runs					
	1	2	3	4	5	Mean
MaxTransSecCnt	75	66	64	85	72	72
RV&VViolations	14	19	17	12	19	16
MaxCpuUtilization	128	126	125	136	121	127
Server Uptime (Hrs)	36	36.5	36	36	36	36
Transaction Count	1219	1221	1256	1227	1280	1241

on the system increased, and was shown by the staccato spikes and troughs in a graph that still remained consistent with Basic test case performance.

Server four performance, shown in Figure 44, was consistent with the SAS-only server four performance curve where maximal startup utilization rapidly stabilized at a shelf below 60%. The comparison of these two curves showed that when the second server was active for a longer period of time, SAS RV&V server four utilization was actually lower than SAS-only behavior. The SAS RV&V curves for server also terminated earlier in all five test cases as the shutdown SAS RV&V rules were engaged.

Much like the Social Network test case results, server five activations were for only short durations due to the static RV&V violation rules, and server five was active only for the mandatory settling time period. Note that this settling time period was statically established before runtime, and therefore did not have the benefit of measuring performance on the edge of viability zones (Tamura et al., 2012). The results of server five activations can be seen in Figure 45. Each activation was at the maximum CPU utilization load period of the Sale Day test cases, and therefore there is a direct correlation to CPU load and PPO response times. As CPU load reached a maximum value for two server operation, PPO violations caused

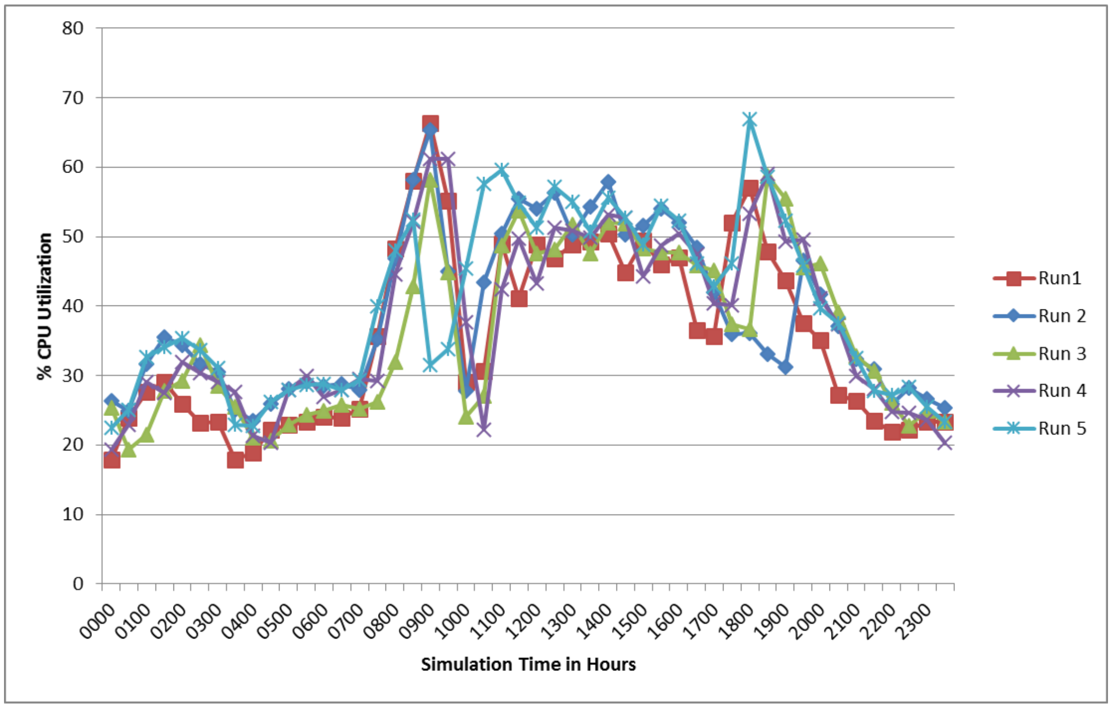


Figure 43. Sale Day test case SAS RV&V CPU performance for server three

server five activation in all five test cases. The total server load did not warrant a third server joining the WPO transactions, but PPO violations from the static SAS RV&V rules triggered server five. These results, along with those of the Social Network test cases, show that SAS RV&V rule tuning would likely have yielded better performance than static rule violation.

The combined CPU utilization of the WPO application is shown in Figure 46, and it showed the impact of the server five activations in the spikes from each test case. These activations were not caused by increased web load, but by Regular or Preferred PPO violations when the WPO application was at maximum sustained utilization. While overall PPO violations remained consistent between SAS-only and SAS RV&V test cases, the combination of RV&V violations and maximum load accounted for server five activations. Because server five was active for only short durations, long running transactions were clearly aborted causing a reduced total transaction volume. This experimentation did not anticipate the impact to total transactions in the SAS RV&V rules as a measure of performance; however, total transactions did seem to provide a marker to predict user performance that PPO violations alone could not predict. The combined server CPU

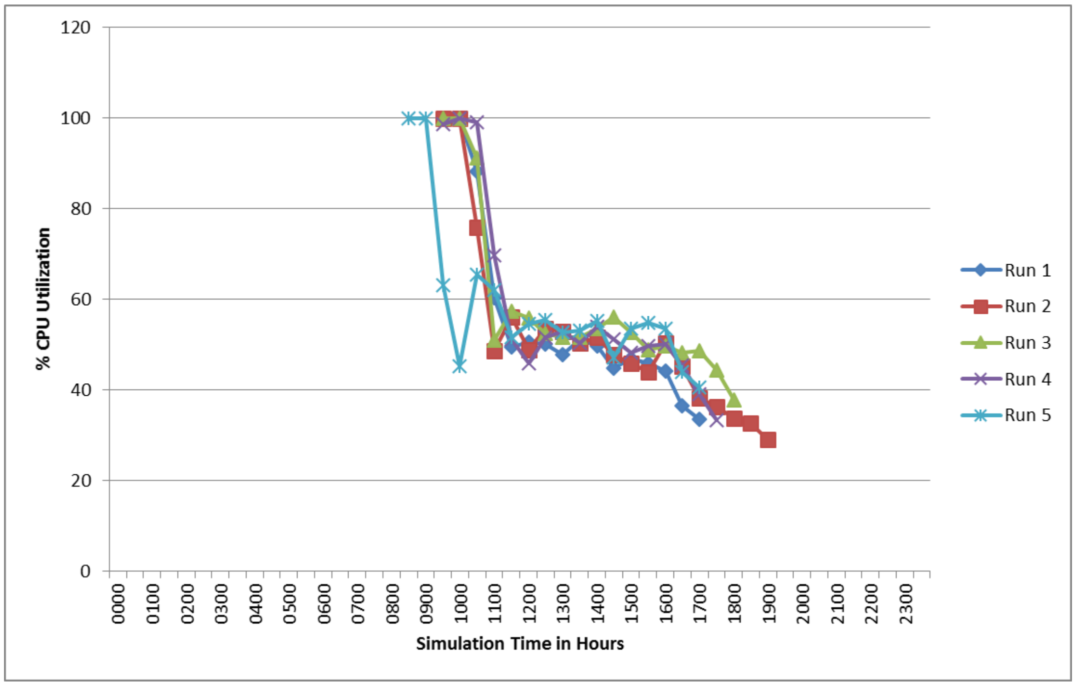


Figure 44. Sale Day test case SAS RV&V CPU performance for server four

utilization graph also demonstrated the advantage of allowing a server to stay active for longer periods of time as opposed to sporadic activations and deactivations. Without the server five spikes this CPU performance curve showed the best server utilization among all test cases.

The total SAS RV&V statistics shown in Table 21 show that Sale Day server uptime savings had a minimum improvement in a single test case of 4% and maximum improvement in a single test case of 10%. The mean of 7% shows that SAS RV&V cost minimization improved with the duration of multi-server operation. RV&V violations of SAS-only performance were measured at 1.3% of total transactions and remained consistent with SAS RV&V violations recorded at 1.5%. The total number of transactions did decrease in the

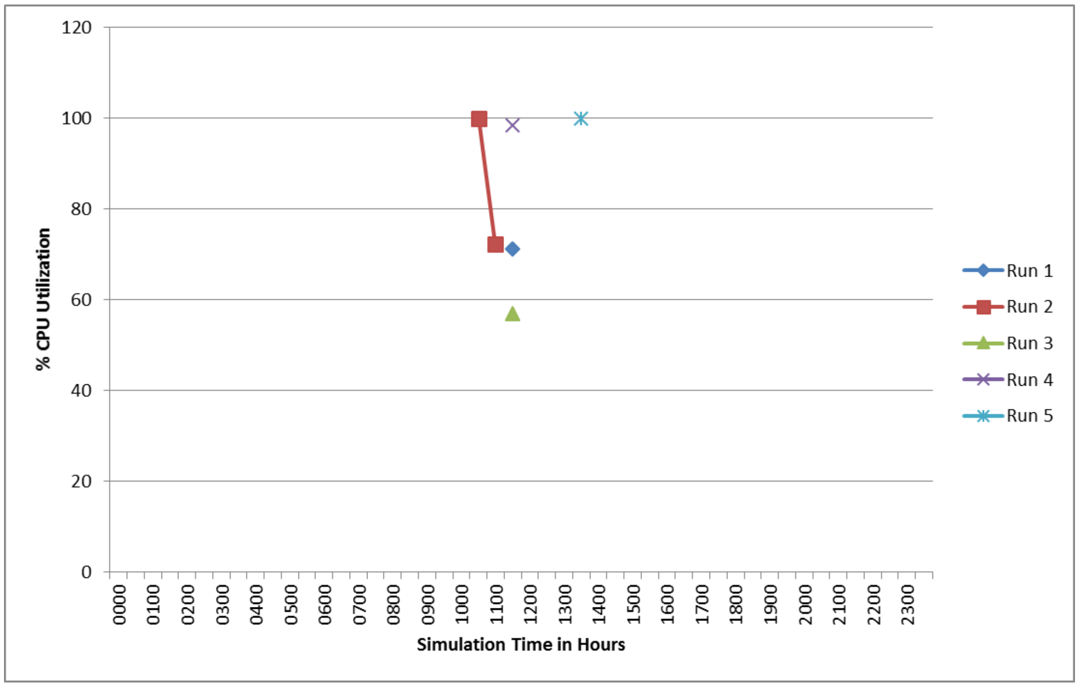


Figure 45. Sale Day test case SAS RV&V CPU performance for server five

same form as the Social Network test cases, largely due to the short duration of server five activations. Overall, the Sale Day SAS RV&V results show that SAS RV&V continues to minimize cost at a greater rate as multi-server operation occurs in a test case. Despite the brute force of SAS RV&V activation and de activation rules, customer satisfaction as a

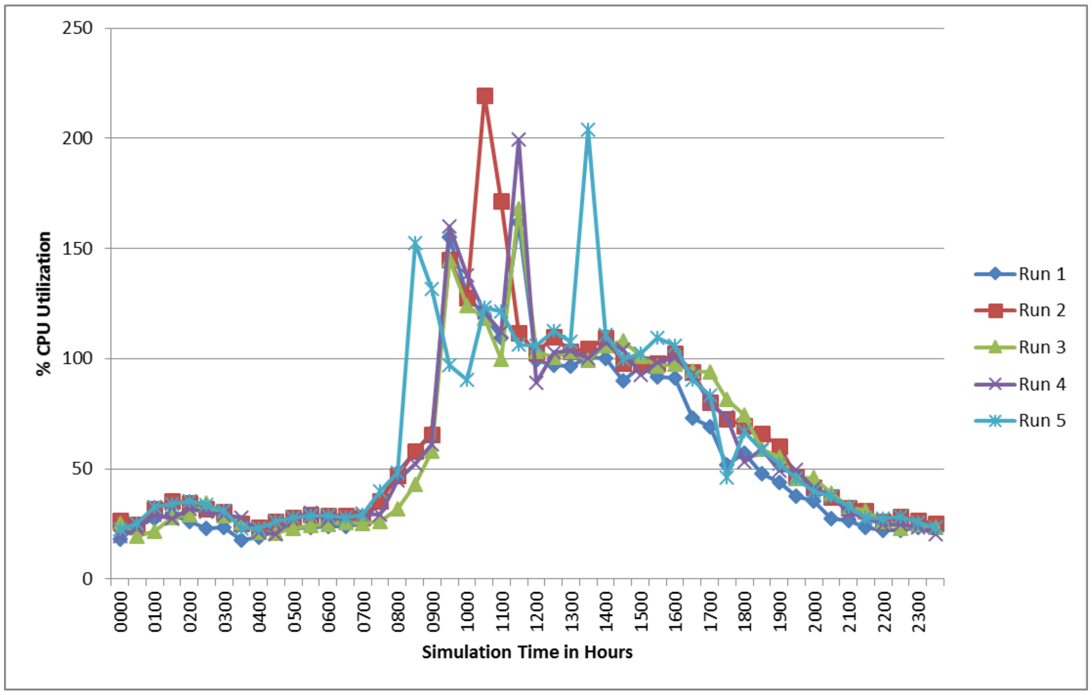


Figure 46. Sale Day test case SAS RV&V CPU performance combined

measure of RV&V violations was maintained and a 7% mean cost saving was achieved. This

Table 21. Sale Day SAS RV&V test case statistics

	Sale Day RV&V Test Case Runs					
	1	2	3	4	5	Mean
MaxTransSecCnt	154	141	180	178	104	151
RV&VViolations	21	21	17	19	15	19
MaxCpuUtilization	162	219	168	199	204	190
Server Uptime (Hrs)	32.5	35	34.5	33	33.5	34
Transaction Count	1212	1141	1148	1120	1206	1165
Savings	10%	4%	4%	8%	7%	7%

savings was achieved because the SAS RV&V rules retarded server four activation until PPO violations required it, and reduced the activation time of server four as web load decreased below single server maximums. This result can be clearly seen by comparing server activation times in Figure 47.

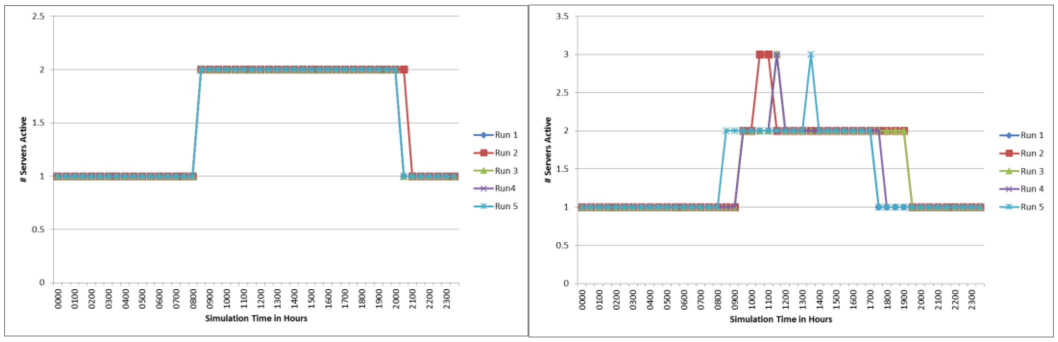


Figure 47. Sale Day test case server uptime comparison

Temporal Logic Transformation Data Analysis

The extraction of the state machine from the extended SmartContext (Villegas et al., 2011) with states and temporal logic transitions was straightforward using the Gamma et al. State pattern (1995); however, experimentation showed that the desired flexibility that temporal logic provides defeated the specificity required by a state machine. The state machine shown in Figure 16 utilized two different temporal logic constructs: next and until. The use of these decorators for the edges required the implementation of additional synthetic states that can be programmatically inserted into the state machine implementation.

Figure 48 shows the actual state machine necessary to implement the temporal logic transitions. For each transition where a temporal logic decorator was implied, a new state was injected into the state machine and a measurement was necessary to transition to the subsequent state. Injected synthetic states are shown in green. The transition from the NORMAL state to the MEDIUM state required that SAS rules one or two fire, or that RV&V

rules one or two fire. State transition rules are shown in red. When these rules were triggered, the state machine immediately transitioned to the NormalMedium synthetic

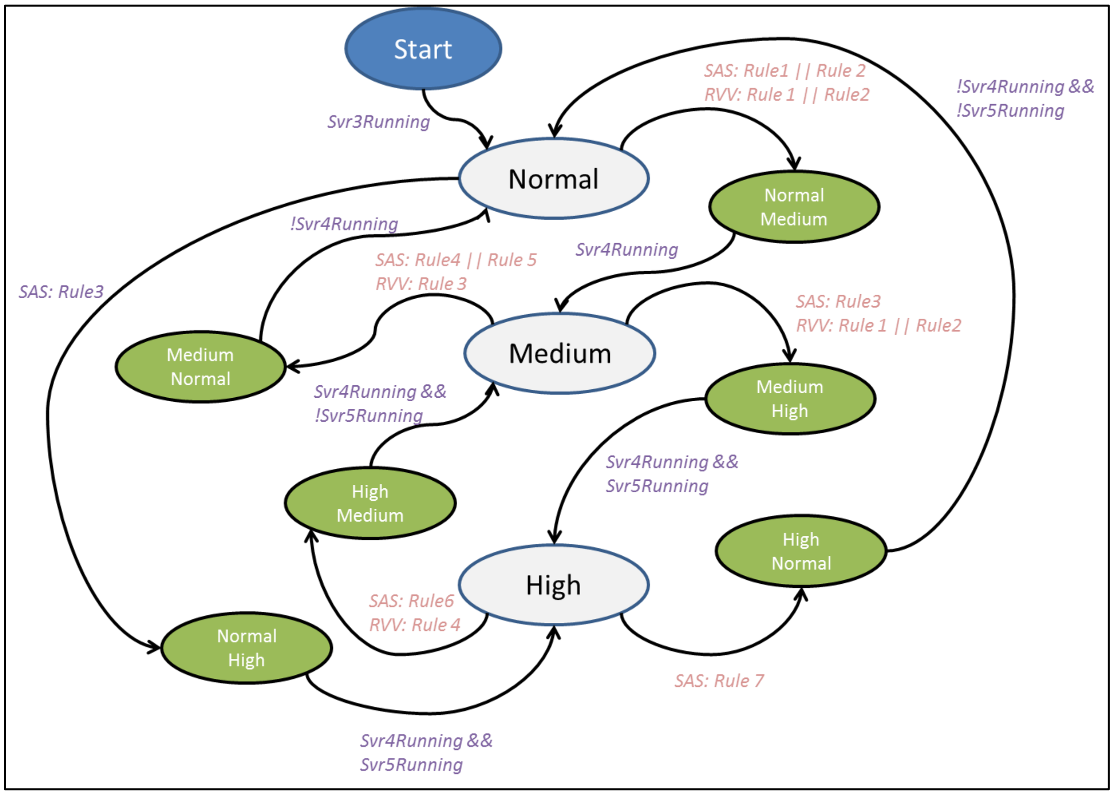


Figure 48. Derived temporal logic state machine

injected state even though the Figure 16 state machine showed a Normal to Medium transition with the Until decorator. The Until decoration implied that all Normal state behavior must be maintained until the input condition for the next state was satisfied. In this case that input condition was that server four was active and available to accept connections.

The combination of the state machine shown in Figure 16, the SAS rules shown in Figure 20, and the RV&V rules of Figures 21 and 22 were integrated into this final state machine. The required information was available in the proposed SmartContext (Villegas et al., 2011) extension, and a simple transformation process produced the final state machine.

This transformation process was similar to that of a non-deterministic finite automata (NFA) to deterministic finite automata (DFA) described in Sipser (2006). In the general case, every state transition that had a temporal logic decorator required the injection of a synthetic state to verify input conditions before the proposed state transition was achieved. Figure 49 clearly demonstrated that the language constructs proposed in the problem statement contained all of the information necessary to construct a primary and secondary feedback loop state machine infused with goals. The NFA transformation method also provided the algorithm by which the extended SmartContext (Villegas et al., 2011) RDF graph was transformed into a DFA using temporal logic.

Findings

SAS systems were inherently designed in such a way that exhaustive or quantitatively verifiable testing of non-functional requirements can be intractable (Dahm, 2010); therefore, new methods of verification and validation must be shown to assure system goals. SAS RV&V was proposed as one method by which SAS systems can be quantitatively verified. Tamura et al. (2012) proposed a conceptual design for a baseline application and test cases that could be used to expand research in the SAS RV&V area. Using this baseline the viability of SAS RV&V methods in a cloud-server environment could be determined. Villegas et al. (2011) also proposed an RDF taxonomy to quantitatively measure performance of web-based applications, and document a measurement framework for monitoring.

This dissertation extended the SmartContext (Villegas et al., 2011) RDF language with states, goals, and temporal logic to document the SAS and SAS RV&V feedback loops.

As well, the Tamura et al. (2012) baseline WPO application was implemented in a set of AWS (“Amazon web services,” 2013) server instances. Two competing SAS RV&V goals were implemented that sought to maximize user satisfaction while minimizing cloud server costs, and three of the proposed test cases from Tamura et al. (2012) were evaluated against the baseline implementation using a static, rule-based feedback approach.

The Basic test case was the first test case evaluated in the WPO environment and five SAS-only runs were executed. The data from these runs was then tabulated and graphed to characterize the best-case operation of a single-server WPO environment. SAS RV&V feedback was then enabled and the same five test cases were executed. Since only a single server was utilized, SAS RV&V server uptime minimization could not improve over the SAS-only execution. The SAS RV&V cloud-server uptime was only 2% worse than SAS-only performance, but RV&V violations were maintained between the two configurations. This result shows that the implementation of SAS RV&V increased cloud server costs only minimally in a worst case scenario, but maintained customer satisfaction. The 2% degradation in cloud server was amplified by the shorted simulation time of two minutes per hour of wall time.

This first experiment demonstrated two shortcomings of the environment. First, static rules that bound variables at startup time caused SAS RV&V rules to fire too soon when PPO violations occurred. The feedback loop environment was not structured to learn an appropriate value for PPO violations from test case to test case; therefore, the system reacted too quickly when two violations happened in consecutive measurements. Secondly, the 2% degradation in cloud server uptime was largely the result of simulation time being set at two minutes per hour. Because this value was set at a comparably small number, the impact of server activations and deactivations spanned multiple measurement periods, amplifying the impact far beyond real-world impacts.

The Social Network test case was the second scenario evaluated against the WPO application using the same testing approach previously defined. In this instance both SAS-only and SAS RV&V performance required multi-server operation. The SAS RV&V second-level feedback loop reduced mean cloud-server costs by 4%, while RV&V violations increased by only 2% over a sample of five test cases. There was a reduction in total transactions caused by the short activation time of server five. This test case demonstrated that SAS RV&V does minimize cloud server costs to a greater degree than the increase in RV&V violations. It also further exposes a weakness in the approach of static rules for RV&V that are not tuned to a specific application in either a manual or learning-mode method. Server five activations should have been suppressed and would have prevented the decrease in total transactions if some form of RV&V tuning were available within the experiment.

The Social Network test cases demonstrated that a goal-based hierarchy must sacrifice some top-level goal performance for subordinate goals. While in SAS-only mode, neither RV&V violations nor server uptime resulted in changes to the server configuration; however, RV&V violations were minimized because servers were under-subscribed in all cases. In SAS RV&V mode an increase in RV&V violations must be permitted to optimize server uptime. While this outcome was not explicitly defined by the feedback loop, it is logical that a hierarchical goal tree would have to surrender some top level goal achievement to subordinate goals. Otherwise, all subordinate goal behavior would be suppressed. In the Social Network test case, a 2% increase in RV&V violations permitted a 4% decrease in WPO server uptime. This tradeoff demonstrated the intent of the goal tree that attempted to balance WPO rule achievement.

The last test case executed against the WPO application was one where an extended Sale Day period required the daylong activation of a second server. Multi-server load had to be maintained for one third of total test case duration. In this instance SAS RV&V interaction continued to reduce cloud-server costs by attaining a 7% mean decrease in server uptime. RV&V violations, which were the primary measure of customer satisfaction, only increased by 0.2% over SAS-only test cases. Sale Day goal satisfaction quantitatively reduced cloud server costs while maintaining customer satisfaction as a measure of regular and preferred RV&V violation in a tiered set of competing rules. The Sale Day test case also demonstrated a byproduct of simple static rule behavior requiring further exploration. As has been previously noted, the variables for each SAS RV&V rule were populated at system startup. Thus, the feedback system was not able to measure or learn from operations within single server or multi-server viability zones. If such a feedback memory were available, the system would have suppressed the activation of server five in all cases, reduced the total number of RV&V violations caused by server transitions, and completed more total transactions. In the general case, RV&V intervention should be suppressed unless the system has moved completely outside its ability to recover under normal, SAS operation. Lastly, RV&V injection always has some negative side effects. This can be shown in the Sale Day test case as the reduction in the total number of completed transactions. While not a feedback loop measured value, the reduced number of total WPO transactions would eventually impact customer satisfaction.

Along with the demonstration that SAS RV&V did quantitatively improve the performance of a SAS application, an extension of states with temporal logic was proposed. This documentation method was proposed to capture SAS and SAS RV&V behavior without having to specify every possible SAS execution path. The inclusion of states, goals and temporal logic into the SmartContext (Villegas et al., 2011) RDF language was proposed as a method to document SAS RV&V measurement and non-functional requirements. Experimentation showed that these extensions to the RDF language, along with a hierarchical goal tree, did provide enough information from which the two feedback loops could be constructed. The use of temporal logic does require a transformation of the state machine that injects synthetic states wherever a temporal logic decorator is used as an edge in the state machine graph.

This dissertation implemented the proposed baseline application from Tamura et al. (2012) in a cloud-server environment, and developed a web load generation approach for three proposed test cases: Basic, Social Network, and Sale Day. An extension to the Villegas et al. (2011) SmartContext RDF language was implemented that allowed for the creation of states with temporal logic transitions, and QoS goals to monitor SAS performance. These same goal constructs allowed for the establishment of SAS RV&V rules. Two goals were implemented in the experimentation with the highest level goal being that of maintaining customer satisfaction. The second goal was a competing goal to minimize server costs. Experimentation showed that this SAS RV&V approach increased cloud server costs over

SAS-only behavior by 2% in the worst case Basic scenario, minimized costs by 4% in the Social Network test case, and minimized cost by 7% in the Sale Day test case. In all test cases customer satisfaction was degraded only minimally showing a quantitative improvement by the use of SAS RV&V methods on SAS-only systems. This research demonstrated that even simple, statically refined SAS RV&V rules implemented in a second-level feedback loop did improve the performance of a SAS cloud-server application.

Chapter 5

Conclusions

This dissertation demonstrated a new approach for verifying that SAS systems maintain their non-functional requirements by implementing a second-level SAS RV&V feedback loop. This method was demonstrated in an Amazon cloud-based server environment where a web purchasing application demonstrated self-adaptive behavior by spinning up new cloud servers based on external context events. The structure of the primary and secondary feedback loops was defined using an extension to the SmartContext (Villegas et al., 2011) monitoring taxonomy that implemented states connected by temporal logic. QoS goals were linked to the state machine via rules that defined generic SAS and SAS RV&V performance boundaries.

This work validated that the implementation of a second-level SAS RV&V feedback loop reduced the cloud-server costs up to a mean of 7% in one of three test cases while only increasing PPO violations by a mean of 0.2% in that same test case. This work achieved the dissertation goal of demonstrating that a hierarchal QoS goal tree in a temporal logic state machine quantitatively improved performance over SAS-only application behavior. This research also provided a sample implementation of the Tamura et al. (2012) baseline application on which future SAS RV&V research can be extended. Three sample test cases were developed and a novel web load generator was provided to simulate load against the benchmark application.

Experimentation demonstrated that the second-level feedback loop increased server costs in the worst case – the Basic SAS-only test case – by 2% because of consecutive PPO violations. In the short-duration Social Network test case, the SAS RV&V feedback loop reduced server costs by a mean of 4% with only a 2% increase in PPO violations. The more sustained external context event inside the Sale Day test case permitted SAS RV&V feedback to engage in one third of the total simulated 24-hour test case duration. When SAS RV&V feedback was enabled for this eight-hour period, a 7% mean reduction in cloud server costs was attained with only a 0.2% increase in PPO violations.

This research also introduced a method by which the myriad of possible state transitions contained in a SAS system can be more generically documented at design time. The addition of temporal logic as part of a feedback process has been previously proposed for SAS systems (Calinescu et al., 2012); however, its definition in a generic modeling language was novel. The addition of states and temporal logic to the SmartContext language (Villegas et al., 2011) provided this needed construct. As part of experimentation it was discovered that the addition of temporal logic transitions required additional synthetic states to be added to the state machine. This transformation followed the NFA to DFA automata construction methods (Sipser, 2006), and resulted in a method to blend SAS and SAS RV&V state transitions with a hierarchy of QoS goals codified in static rules.

Implications

The fundamental implication of this research was that the proposal made by Tamura et al. (2012) that SAS RV&V might be a method by which SAS-only systems can quantitatively build trust was valid. Not only were non-functional requirements maintained by the second-level feedback loop, but non-functional metrics improved consistently with the duration of SAS RV&V engagement. As SAS systems become more complex and used in mainstream applications, this research demonstrated that an RV&V feedback capability had minimal impact on primary system behavior in the worst case, and consistently improved overall non-functional requirements achievement, even with a very simplistic static rule implementation.

Secondly, the Tamura et al. (2012) baseline application proposal has also been implemented in a demonstrable cloud-server environment with accompanying test cases. The baseline proposal did not specify how to generate load for the environment, but a web load simulator was developed to make the WPO application a valuable SAS RV&V research tool. The web load simulator was also novel in that it provided a three-variable method of defining how to generate parallel transactions consistent with the RosenStein (2000) daily transaction model. The method of assigning a load value by hour for the web load simulator and allowing it to determine the appropriate mix of browse-only events versus transactional events was new to the literature.

AI or stochastic approaches have been a focal point for SAS feedback loop behaviors (Arshad et al., 2004; Calinescu et al., 2012). Defining complex AI solutions or populating probabilities for a Markov's Random Walk algorithm are likely too complex for most commercial applications; however, the simple rules that instrumented the state machine transitions in this experimentation should be easily defined for most applications. Secondly, the RDF taxonomy approach is easily ingested by almost every high-level language, and may enable the creation of feedback systems without special tools or skills.

Beyond just demonstrating the value of SAS RV&V, there are several other areas where the literature can be further expanded based on this work. The problem of RV&V intervention side effects was clearly demonstrated in the Sale Day test case when the total transaction count was negatively impacted due to short-duration server five activations and deactivations. The total transaction count was not a value that the Tamura et al. (2012) baseline proposed to measure, but failed transactions would clearly impact customer satisfaction. The fundamental decision of what constitutes an appropriate intervention sequence for RV&V in the general case versus further measurement is a new research question. The value of RV&V is not only intervention, but a documentation method for how the SAS-only system operates. Like SAS RV&V for spacecraft and the avionics industry (Felt et al., 2010; Laurent, 2010), much of the value of SAS RV&V may be measurement feedback for the developer of the next software update, not intervention in current system operation. Each test case independently demonstrated that SAS RV&V must lean toward application stability over intervention whenever possible, even at the edges of a viability zone.

The SmartContext (Villegas et al., 2011) extensions were shown to have substantial value for implementing a system performance specification in the form of a temporal logic state machine. This specification was then transformed into the logic of the feedback subsystem. The work of Calinescu et al. (2012) also used temporal logic within the SAS system, but it did not result in a generally reusable semantics. The temporal logic state machine was shown to be an effective method of defining and measuring adaptive behavior.

The syntax was also readily transformed into high-level language code, and could be used beyond SAS as a method of instrumenting software systems.

Decision trees are not new methods for documenting learning systems (Mitchell, 1997), but the integration of a QoS taxonomy (Fu et al., 2007) as the syntax for a goal tree provided a new semantics for the goal tree approach. With the addition of the Maximize and Minimize directional attributes, like those proposed by Lapouchnian et al. (2005), a complete method of defining a system-level measurement language was provided. The structure of a directional goal being represented by definitive rules draws from Villegas et al. (2011), but its implementation at the system level is unique. Further exploration could be performed to define a generic rule syntax that is also directly transformed into high-level languages.

Recommendations

While this dissertation's purpose was to quantitatively demonstrate that SAS RV&V methods improved non-functional goal achievement over SAS-only systems, it did uncover some areas requiring further investigation. Tamura et al. (2012) noted that additional research needed to be conducted in the area of settling time and measurement at the edges of viability zones. This prediction proved true. The utilization of SAS RV&V rules with values fixed at system startup caused the SAS RV&V feedback system to override SAS performance too early when the system neared the edges of a viability zone. Secondly, settling time was also established at a global level and not localized to a particular viability zone – or state – transition. In both cases there would have been substantive improvement by having the feedback system record performance of the SAS RV&V system in each test case and then modify its rule variables based on previous performance. The baseline system did not define that the feedback loop have persistent memory, but this capability would likely have improved SAS RV&V performance over these experiments.

The total number of test cases proposed by Tamura et al. (2012) was truncated as the first three demonstrated quantitative improvement; however, continued experimentation with the Black Friday test case may have shown further improvement in SAS RV&V performance metrics. The Black Friday test case was developed by this research, but was not executed due to the added tuning and experimentation time required. The Black Friday test case should demonstrate an upper bound of SAS-only activity when all three servers are active. SAS RV&V rules will likely have a more substantial impact in this test case than any of the others because the viability zone is larger and is distributed over most the test case duration. Thus, the highest SAS RV&V performance may be expected by the Black Friday event experiments.

The experimentation also established a simulation time ratio where two minutes equaled one hour of wall time. The Amazon cloud server environment was, however, running in wall time and the impact of server activation and deactivation times were substantially amplified. Server transitions were recorded at longer intervals than they would be in real operation. This amplification of cloud-server effects degraded the cost savings of each test case's performance. In future experimentation the simulation time ratio should be reduced to further negate the activation and deactivation time impacts on cost savings. This experimentation also showed that there was very little variability in test case performance between any of the runs of the same type. Thus, five runs per test case may not be required to establish new performance criteria.

The invocation of SAS RV&V rules in the WPO application was observed to almost always have negative side effects. This behavior was anticipated because the system is being forced into a stable state and the context that caused instability is likely to continue until the settling time clock expires. An example of this behavior was the spinning up of server five during the Sale Day test case. The additional server provided no improvement in PPO performance metrics, but load began to be shifted to server five before the settling time clock expired. The feedback system then determined that the total transaction load did not warrant three active servers, and began the server five shutdown processes stranding several long-running transactions. Unintended side effects of SAS RV&V intervention are also an area where further exploration is required.

The concept of settling time was mentioned by Tamura et al. (2012) in a few differing contexts; however, this experimentation brought the concept into clearer focus. When an RV&V intervention takes place, the simple approach of activating a settling time clock was ineffective. The events that caused the intervention are likely still occurring and the system is ignoring the side effects until a stable state is achieved. Settling time is likely better determined by satisfying a new set of constraints, not by a determination of time. The time-based approach may result in a looping of RV&V intervention, which is always undesirable.

Summary

The concept of self-adaption is a further specification of the concepts put forward by Kephart and Chess (2003) in their autonomic computing challenge. Self-adaption is a systems design approach where a feedback loop operates externally from the primary application and samples internal and external system context to maintain system goals.

Internal context refers to those measures within the application that the feedback loop should monitor, and external context the events happening in the environment outside the application that impact performance. The fundamental feedback approach for a self-adaptive system was defined by Salehie and Tahvildari (2009) and its structure remains largely unchanged. Interest in SAS systems has resulted in substantial research, but there has been a limited commercial acceptance of these new concepts because of a lack of trust in SAS system performance (Tamura et al., 2012).

Space systems depend largely on autonomic software and the same problems of trust have been addressed in that domain by the use of RV&V concepts (Goldberg et al., 2005). Runtime verification methods have been successfully applied for spacecraft and avionics systems where all possible outcomes can't be predicted prior to deployment. In each case temporal logic was employed to simplify complex state machines and permit the RV&V system to verify high-level goals while allowing the primary system to operate without secondary intervention. The ability to rapidly establish trust in these complex software systems has also been requested for future unmanned aerial vehicles, and was listed as one of the U.S. Air Forces primary research challenges over the next 30 years (Dahm, 2010).

Several SAS research roadmaps called out the need for quantitative verification methods for SAS systems as being a critical priority for this technology to achieve substantial commercial acceptance (Calinescu et al., 2012; Cheng et al., 2009; Lemos et al., 2012; Tamura et al., 2012). The Tamura et al. (2012) roadmap suggested an extension to the MAPE-K feedback system (Salehie & Tahvildari, 2009) that added a secondary RV&V feedback loop to operate in concert with the primary SAS feedback loop. Tamura et al.

(2012) also proposed a web purchasing application and associated test cases as a platform on which SAS RV&V experimentation could be conducted. These building blocks were a necessary first step in establishing a research agenda for SAS improvements.

Because SAS applications are inherently complex, new methods of documenting requirements – especially non-functional requirements – were required if runtime methods were to be successfully employed (Sawyer, Bencomo, & Whittle, 2010). Villegas et al. (2011) had developed a web application monitoring language called SmartContext to provide an approach for documenting SLAs at the application level. Tamura et al. (2012) also suggested that SmartContext might be adapted to document SAS RV&V systems. Tamura et al. (2012) suggested that the Requirements@Runtime initiative might also play a role in documenting SAS RV&V capabilities (Sawyer, Bencomo, & Whittle, 2010). The confluence of these requirements efforts provided a vector for this research.

While SmartContext (Villegas et al., 2011) provided a language construct for monitoring SLAs, it did not address the semantics required for defining non-functional requirements. The majority of non-functional characteristics – such as application performance, availability, cost, and reliability – were defined in a QoS taxonomy of Fu et al. (2007). Because SmartContext was based in the RDF specification, the Fu et al. (2007) taxonomy could easily be attached to the semantics as a goal tree.

Previous efforts in SAS RV&V have used AI or stochastic methods to (Arshad et al., 2004; Calinescu et al., 2012) achieve the Planner functions from the Tamura et al. (2012) model; however, these functions were always tightly coupled to the feedback loop or even integrated into the primary application. The research roadmaps called for a more general method of documenting non-functional requirements that could be readily integrated into feedback loops from many different applications. Thus, a language-based approach like SmartContext was desired. The space systems RV&V approach always employed some form of temporal logic as part of their language specification (Artho, Barringer, & Goldberg, 2005), and a temporal logic approach does provide the opportunity for the state machine to be quantifiably verifiable via a Büchi automation; therefore, the addition of a temporal logic approach for defining SAS RV&V states was a viable formal method to be explored.

The problem statement of this dissertation was to show that SAS RV&V could quantifiably improve the adherence to non-functional requirements over SAS-only systems. This was to be accomplished by implementing the Tamura et al. (2012) baseline application and test cases to evaluate the use of an extended SmartContext language containing QoS goals and states connected by temporal logic.

The baseline application defined by Tamura et al. (2012) ran in a cloud-server environment where the addition and subtraction of application servers constituted the SAS behavior. The WPO application was implemented in the Apache Wicket (2014) framework and complied with the baseline requirements of having two classes of customers: Regular and Preferred. The time to complete a purchase was defined as the primary application metric of PPO, and would be the basis for RV&V measurement of internal context. The WPO application had no knowledge of the feedback loop nor of the number of application servers processing transactions.

In order to properly evaluate the WPO application in a SAS-only and then SAS RV&V context, a web load generator was required. To create a realistic profile of internet purchasing site transactions, the Rosenstien (2000) load profile was utilized to develop a load generator where 49 browses of the site were randomly interspersed with one of two differing purchasing scenarios. Each of these 50 web transactions with the WPO application was referred to as a cycle, and the definition of parallel cycles was the metric by which load was generated against the WPO application. The entire WPO application environment was implemented in the Amazon cloud on three small Ubuntu instances. Each instance had an independent version of the WPO application that was externally loaded.

Three of the four Tamura et al. (2012) test cases were implemented using the web load generator and JSON-based load profiles. The Basic test case provided a single-server load profile over a 24-hour period that closely followed the Rosenstein (2000) load profile. The second test case was a Social Network test case that loaded the WPO application with an external context sale event. This event triggered the primary feedback loop to require a second server to join the application environment. Load increased to approximately double that of the Basic test case over a short duration. The last test case was the Sale Day tests case. This experiment also triggered the primary feedback loop to activate a second server over an eight-hour period and approximate a sustained two-server load.

The baseline WPO application, web load simulator, and feedback loops were hosted in the AWS environment in five Ubuntu images. Experimentation was separated into two phases. The first phase tuned the web load generator to attain a stable load curve that did not exhibit substantial HTTP exceptions and was repeatable across each of the test cases. It was determined that the Amazon software load balancer implemented connection pipelining with the Apache Tomcat instances, and therefore defeated the load generation algorithm by combining transactions into a single TCP connection. To resolve this problem, the load balancing requirements were implemented inside the web load generator and all phase one goals were achieved.

The second phase of experimentation was to execute each of the three test cases previously described in SAS-only mode and then with the second-level SAS RV&V feedback loop activated. Each test case was executed five times in the SAS-only mode and then SAS RV&V mode. Mean values were calculated of each required statistic and comparisons were made among the SAS-only and SAS RV&V values to determine if quantitative improvements were achieved.

The Basic test case execution represented the worst-case as SAS RV&V intervention could never reduce cloud server costs when only one server was active in SAS-only mode. The SAS RV&V results did show that PPO violations remained consistent between SAS-only and SAS RV&V execution. Customer satisfaction was maintained or Maximized per the QoS goal tree; however, cloud server costs increased by a 2% mean because static RV&V rules fired with consecutive PPO violations. The Social Network test case was the first multi-server experiment and cloud server savings of a 4% mean were achieved with only a 2% mean increase in PPO violations. Lastly, the Sale Day test case achieved a 7% mean reduction in cloud server costs while increasing PPO violations by no more than 0.2%. The system did, therefore, quantitatively demonstrate that the SAS RV&V method improved the QoS goal of “Maximize performance throughput” compared to SAS-only data.

Several topics for future research were discovered during experimentation. The first of these was that static RV&V whose variables are bound at instantiation are not the best method for triggering RV&V intervention. Secondly, the Black Friday test case proposed by Tamura et al. (2012) was not executed in this research and likely would have shown further cloud server cost savings over the other three test cases. The topic of RV&V intervention versus continued measurement also requires further research. In many cases the side effects from RV&V intervention nearly cancelled out its benefits; therefore, RV&V intervention should be explored as a minimally invasive construct, but actively used to provide feedback for future software improvement. Simulation time in these experiments was set to a ratio of two minutes per hour of wall time. Because this ratio was so short, the impact of AWS cloud server activations and deactivations was amplified and likely reduced the actual cloud server savings that would have been achieved. Lastly, settling time as described by Tamura et al. (2012) can clearly be seen to be better defined by a set of constraints that have been satisfied in the feedback process vice the timeout of a single, global settling time clock. A constraint based approach to settling time requires further research.

This dissertation explored how SAS non-functional requirements can be documented in a language-based approach that is easily implemented into a feedback loop. The research showed that the implementation of a second-level RV&V feedback loop demonstrated quantitative improvement over SAS-only methods, even with simple static rules. States with temporal logic transitions were an adequate documentation method to capture SAS system operations without requiring too much specificity. The use of a QoS goal hierarchy enabled the translation of human readable goals to SAS RV&V rules that the second-level feedback loop could support. This dissertation achieved its goals, and added new content to the SAS literature.

Appendix A

Basic Test Case Results

The following tables provide additional detail from executing the Basic test cases.

Basic SAS-only test case performance for server 3

	Basic Test Case Runs					
	1	2	3	4	5	
Wall Time	100-%idle	100-%idle	100-%idle	100-%idle	100-%idle	Mean
0000	12.5	15.23	14	16.37	13.29	14.278
	12.91	14.34	12.85	15.59	16.59	14.456
0100	12.97	17.1	14.69	16.98	14.93	15.334
	15.85	19.11	16.69	17.06	18.04	17.35
0200	17.75	17.96	21.04	18.39	18.47	18.722
	17.64	26.25	19.18	20.6	19.48	20.63
0300	16.67	16.77	17.68	18.36	16.93	17.282
	13.44	15.13	14.13	17.97	15.14	15.162
0400	13.67	14.7	14.51	15.76	16.6	15.048
	13.8	14.98	14.6	15.26	16.14	14.956
0500	15.34	17.52	15.93	17.08	17.54	16.682
	16.06	19.21	18.27	18.43	23.28	19.05
0600	15.88	15.85	13.78	16.49	17.46	15.892
	14.37	14.82	14.48	16.19	16.05	15.182
0700	16.31	14.69	15.16	16.06	16.89	15.822
	14.16	15.88	14.86	16.37	16.44	15.542
0800	15.76	21.09	20.56	20.27	21.93	19.922
	23.72	27.08	27.53	30.04	32.43	28.16
0900	32.42	35.16	37.27	38.21	40.6	36.732
	38.21	40.89	40.52	42.39	43.81	41.164
1000	41.73	45.58	41.59	46.71	48.38	44.798
	46.71	47.43	47.89	52.85	55.32	50.04
1100	52.4	52.95	54.68	59.97	59.22	55.844
	60.77	63.96	61.06	71.76	70.02	65.514
1200	64.47	58.86	56	64.73	65.2	61.852
	55.86	59.37	56.25	61.13	64.5	59.422
1300	56.22	56.57	52.88	59.51	62.69	57.574
	54.31	52.67	53.78	58.97	61.86	56.318
1400	55.42	56.89	55.49	60.26	61.43	57.898
	60.69	60.36	61.34	65.85	66.31	62.91
1500	60.54	59.55	56.43	60.34	60.2	59.412
	54.8	52.17	55.25	57.76	57.97	55.59
1600	58.8	57.32	54	60.49	62.31	58.584
	58.6	55.55	57.05	61.69	65.32	59.642
1700	54.07	48.38	49.01	49.92	52.22	50.72
	41.17	46.07	40.41	46.39	46.69	44.146
1800	39.45	40.26	36.74	41.23	44.67	40.47
	36.4	36.54	36.97	44.97	40.86	39.148
1900	37.99	31.86	28.72	33.72	34.57	33.372
	27.8	28.06	27.15	32.04	30.69	29.148
2000	24.98	26.79	23.61	27.38	27.32	26.016
	22.47	21.74	22.59	25.84	28.54	24.236
2100	20.91	20.54	17.18	24.35	22.97	21.19
	16.46	17.69	16.73	19.33	19.84	18.01
2200	18.23	14.59	13.23	18.31	16.39	16.15
	12.85	13.55	11.11	13.4	13.75	12.932
2300	12.59	12.79	12.29	13.39	14.16	13.044
	11.93	13.59	12.53	15.75	12.66	13.292

Basic SAS RV&V test case performance for server three

	Basic Rvv Test Case Runs - Server 3					
	1	2	3	4	5	
Wall Time	100-%idle	100-%idle	100-%idle	100-%idle	100-%idle	Mean
0000	17.21	15.48	18.73	15.98	16.93	16.866
	18.06	16.44	18.26	17.29	20.29	18.068
0100	18.15	19.37	21.55	23.29	20.45	20.562
	20.66	20.22	20.31	24.78	27.24	22.642
0200	25.05	18.04	24.83	23.09	22.79	22.76
	16.42	15.83	21.84	18.9	18.15	18.228
0300	16.61	15.75	18.14	18.05	17.93	17.296
	16.92	15.96	17.58	18.08	17.92	17.292
0400	18.1	14.5	17.85	20.66	22.67	18.756
	23.86	16.75	20.35	19.63	20.92	20.302
0500	21.28	18.1	20.93	21.79	21.41	20.702
	17.01	13.83	21.59	18.87	18.72	18.004
0600	16.38	17.09	18.27	18.1	17.67	17.502
	16.43	14.38	19.92	21.1	21.48	18.662
0700	19.84	15.34	19.93	18.73	20.82	18.932
	22	21.92	21.74	22.54	29.26	23.492
0800	33	31.56	28.91	31.69	37.29	32.49
	40.62	39.97	41.21	42.4	47.4	42.32
0900	41.95	41.79	49.83	56.06	56.53	49.232
	50.45	47.33	54.75	53.09	54.68	52.06
1000	52.74	50.42	56.31	57.2	62.97	55.928
	59.18	58.49	63.35	65.26	72.52	63.76
1100	66.08	64.67	73.03	76.3	75.58	71.132
	57.73	56.18	75.55	77.14	67.62	66.844
1200	58.86	56.8	70.02	68.36	66.27	64.062
	61.57	54.11	68.24	67.84	63.69	63.09
1300	56.18	54.97	65.91	65.38	66.26	61.74
	59.28	61.02	69.94	69.17	71.86	66.254
1400	65.03	62.26	71.12	74.76	74.29	69.492
	59.87	53.14	73.56	67.82	64.19	63.716
1500	59.32	56.36	65.1	64.81	66.42	62.402
	60.6	59.28	70.08	68.86	68.28	65.42
1600	59.62	57.94	70.73	66.52	65.81	64.124
	42.02	43.8	67.61	62.24	53.29	53.792
1700	48.36	43.82	57.57	51.3	51.39	50.488
	42.13	37.34	51.57	47.46	47.74	45.248
1800	39.37	38.46	46.18	45.71	41.07	42.158
	31.02	30.97	44.67	41.64	34.9	36.64
1900	29.33	27.06	36.02	37.54	36.86	33.362
	28.99	24.88	32.94	34.72	27.7	29.846
2000	26.19	24.43	31.6	29	28.71	27.986
	21.23	18.36	28.83	29.82	22.27	24.102
2100	21.64	20.02	21.36	25.44	19.63	21.618
	17.43	13.48	20.33	21.45	17.17	17.972
2200	18.95	13.16	19.44	19.12	15.13	17.16
	13.69	14.44	14.65	15.52	15.11	14.682
2300	14.31	13.1	15.28	16.13	12.72	14.308
	10.29	*13.1*	16.89	15.02	*12.72*	13.604

Basic SAS RV&V test case performance for server four

	Basic Rvv Test Case Runs - Server 4					
	1	2	3	4	5	
Wall Time	100-%idle	100-%idle	100-%idle	100-%idle	100-%idle	Mean
0000	0	0	0	0	0	0
	0	0	0	0	0	0
0100	0	0	0	0	0	0
	0	0	0	0	0	0
0200	0	0	0	0	0	0
	0	0	0	0	0	0
0300	0	0	0	0	0	0
	0	0	0	0	0	0
0400	0	0	0	0	0	0
	0	0	0	0	0	0
0500	0	0	0	0	0	0
	0	0	0	0	0	0
0600	0	0	0	0	0	0
	0	0	0	0	0	0
0700	0	0	0	0	0	0
	0	0	0	0	0	0
0800	0	0	0	0	0	0
	0	0	0	0	0	0
0900	0	0	0	0	0	0
	0	0	0	0	0	0
1000	0	0	0	0	0	0
	0	0	0	0	0	0
1100	0	0	0	0	0	0
	0	0	0	0	83.47	16.694
1200	0	0	0	0	0	0
	0	0	0	0	0	0
1300	0	0	0	0	0	0
	0	0	0	0	0	0
1400	0	0	0	0	0	0
	0	0	0	0	0	0
1500	0	0	0	0	0	0
	0	0	0	0	0	0
1600	73.55	0	0	0	0	14.71
	0	0	0	0	0	0
1700	0	0	0	0	0	0
	0	0	0	0	0	0
1800	0	0	0	0	0	0
	0	0	0	0	0	0
1900	0	0	0	0	0	0
	0	0	0	0	0	0
2000	0	0	0	0	0	0
	0	0	0	0	0	0
2100	0	0	0	0	0	0
	0	0	0	0	0	0
2200	0	0	0	0	0	0
	0	0	0	0	0	0
2300	0	0	0	0	0	0
	0	0	0	0	0	0

Basic SAS RV&V test case performance combined

	Basic Rvv Test Case Runs - Combined					
	1	2	3	4	5	
Wall Time	100-%idle	100-%idle	100-%idle	100-%idle	100-%idle	Mean
0000	17.21	15.48	18.73	15.98	16.93	16.866
	18.06	16.44	18.26	17.29	20.29	18.068
0100	18.15	19.37	21.55	23.29	20.45	20.562
	20.66	20.22	20.31	24.78	27.24	22.642
0200	25.05	18.04	24.83	23.09	22.79	22.76
	16.42	15.83	21.84	18.9	18.15	18.228
0300	16.61	15.75	18.14	18.05	17.93	17.296
	16.92	15.96	17.58	18.08	17.92	17.292
0400	18.1	14.5	17.85	20.66	22.67	18.756
	23.86	16.75	20.35	19.63	20.92	20.302
0500	21.28	18.1	20.93	21.79	21.41	20.702
	17.01	13.83	21.59	18.87	18.72	18.004
0600	16.38	17.09	18.27	18.1	17.67	17.502
	16.43	14.38	19.92	21.1	21.48	18.662
0700	19.84	15.34	19.93	18.73	20.82	18.932
	22	21.92	21.74	22.54	29.26	23.492
0800	33	31.56	28.91	31.69	37.29	32.49
	40.62	39.97	41.21	42.4	47.4	42.32
0900	41.95	41.79	49.83	56.06	56.53	49.232
	50.45	47.33	54.75	53.09	54.68	52.06
1000	52.74	50.42	56.31	57.2	62.97	55.928
	59.18	58.49	63.35	65.26	72.52	63.76
1100	66.08	64.67	73.03	76.3	75.58	71.132
	57.73	56.18	75.55	77.14	151.09	83.538
1200	58.86	56.8	70.02	68.36	66.27	64.062
	61.57	54.11	68.24	67.84	63.69	63.09
1300	56.18	54.97	65.91	65.38	66.26	61.74
	59.28	61.02	69.94	69.17	71.86	66.254
1400	65.03	62.26	71.12	74.76	74.29	69.492
	59.87	53.14	73.56	67.82	64.19	63.716
1500	59.32	56.36	65.1	64.81	66.42	62.402
	60.6	59.28	70.08	68.86	68.28	65.42
1600	133.17	57.94	70.73	66.52	65.81	78.834
	42.02	43.8	67.61	62.24	53.29	53.792
1700	48.36	43.82	57.57	51.3	51.39	50.488
	42.13	37.34	51.57	47.46	47.74	45.248
1800	39.37	38.46	46.18	45.71	41.07	42.158
	31.02	30.97	44.67	41.64	34.9	36.64
1900	29.33	27.06	36.02	37.54	36.86	33.362
	28.99	24.88	32.94	34.72	27.7	29.846
2000	26.19	24.43	31.6	29	28.71	27.986
	21.23	18.36	28.83	29.82	22.27	24.102
2100	21.64	20.02	21.36	25.44	19.63	21.618
	17.43	13.48	20.33	21.45	17.17	17.972
2200	18.95	13.16	19.44	19.12	15.13	17.16
	13.69	14.44	14.65	15.52	15.11	14.682
2300	14.31	13.1	15.28	16.13	12.72	14.308
	10.29	13.1	16.89	15.02	12.72	13.604

Appendix B

Social Network Test Case Results

Social Network SAS-only test case performance for server three

	Social Network Sas Test Case Runs - Server 3					
	1	2	3	4	5	
Wall Time	100-%idle	100-%idle	100-%idle	100-%idle	100-%idle	Mean
0000	12.19	13.35	17.09	10.13	14.73	13.498
	12.82	19.51	14.7	17.38	17.03	16.288
0100	16.23	17.17	17.4	16.59	15.84	16.646
	17.71	19.97	17.23	16.78	18.92	18.122
0200	15.46	21.33	20.11	18.25	20.63	19.156
	20.71	25.35	21.92	21.78	21.86	22.324
0300	15.65	22.63	17.37	20.62	17.07	18.668
	15.21	18.32	15.72	18.99	16.78	17.004
0400	10.56	16.31	10.65	15.49	10.96	12.794
	6.63	9.82	8.3	9.06	8.91	8.544
0500	7.24	12.6	12.7	10.03	12.28	10.97
	11.15	18.19	16.11	13.82	17.4	15.334
0600	13.78	18.03	18.73	18.16	15.81	16.902
	12.15	17.18	15.62	14.67	20.3	15.984
0700	12.06	18.15	16.08	17.06	17.29	16.128
	11.58	17.86	16.87	14.97	15.91	15.438
0800	10.43	17.38	9.99	17.24	10.97	13.202
	6.47	10.31	8.56	10.07	8.79	8.84
0900	7.93	11.03	10.79	8.98	10.67	9.88
	8.1	12.09	8.76	9.83	8.59	9.474
1000	11.66	16.06	21.31	10.56	21.67	16.252
	28.28	39.49	39.35	28.36	38.12	34.72
1100	45.5	49.74	50.54	41.14	49.84	47.352
	52.37	63.7	62.57	61.66	59.5	59.96
1200	48.09	63.18	50.83	56.69	48.76	53.51
	43.62	48.25	52.7	52.36	49.34	49.254
1300	42.22	61.12	53.31	55.2	58.66	54.102
	48.79	59.01	57.22	50.23	57.89	54.628
1400	51.63	66.34	59.96	54.99	61.82	58.948
	52.92	67.72	63.54	60.74	64.16	61.816
1500	54.3	68.11	59.99	64.87	59.08	61.27
	51.37	66.24	54.57	55.52	59.94	57.528
1600	51.23	63.52	58	58.16	56.46	57.474
	54.15	68.11	58.62	59.27	61.2	60.27
1700	50.68	64.3	49.74	57.01	52.68	54.882
	37.32	48.56	44.5	46.39	42.05	43.764
1800	40.34	52.04	40.07	43.13	42.9	43.696
	34.64	44.48	38.47	38.27	40.95	39.362
1900	32.8	39.6	34.11	38.58	33.22	35.662
	29.1	32.72	27.83	28.82	32.59	30.212
2000	24.04	32.43	25.75	28.81	27.35	27.676
	23.92	30.03	25.63	26.88	25.51	26.394
2100	26.65	25.14	21.85	21.91	21.33	23.376
	17.46	21.32	19.19	18.56	19.64	19.234
2200	16.28	18.41	15.84	18.52	17.81	17.372
	13.02	16.69	12.72	13.28	13.44	13.83
2300	11.08	17.14	12.84	13.91	15.18	14.03
	14.05	14.52	13.55	14.31	20.9	15.466

Social Network SAS-only test case performance for server four

	Social Network Sas Test Case Runs - Server 4					
	1	2	3	4	5	
Wall Time	100-%idle	100-%idle	100-%idle	100-%idle	100-%idle	Mean
0000						0
						0
0100						0
						0
0200						0
						0
0300						0
						0
0400						0
						0
0500						0
						0
0600						0
						0
0700						0
						0
0800						0
						0
0900						0
	37.99	38.47	31.71	38.42	30.48	35.414
1000	30.81	28.33	27.19	26.76	27.17	28.052
	44.9	38.43	35.4	42.91	36.27	39.582
1100	52.52	52.56	49.44	53.33	50.67	51.704
	62.83	61.12	57.28	58.86	58.82	59.782
1200	53.58	52.09	50.77	45.77	52.72	50.986
	53.95	51.34	49.12	50.77	47.57	50.55
1300	33.83	31.42	37.86	30.11	41.93	35.03
	0.62	0.52	1.08	0.5	0.87	0.718
1400	0.18	0.18	0.23	0.22	0.22	0.206
	0.17	0.32	0.23	0.15	0.25	0.224
1500	0.23	0.22	0.23	0.25	0.32	0.25
						0
1600						0
						0
1700						0
						0
1800						0
						0
1900						0
						0
2000						0
						0
2100						0
						0
2200						0
						0
2300						0
						0

Social Network SAS-only test case performance for combined

	Social Network Sas Test Case Runs - Combined					
	1	2	3	4	5	
Wall Time	100-%idle	100-%idle	100-%idle	100-%idle	100-%idle	Mean
0000	12.19	13.35	17.09	10.13	14.73	13.498
	12.82	19.51	14.7	17.38	17.03	16.288
0100	16.23	17.17	17.4	16.59	15.84	16.646
	17.71	19.97	17.23	16.78	18.92	18.122
0200	15.46	21.33	20.11	18.25	20.63	19.156
	20.71	25.35	21.92	21.78	21.86	22.324
0300	15.65	22.63	17.37	20.62	17.07	18.668
	15.21	18.32	15.72	18.99	16.78	17.004
0400	10.56	16.31	10.65	15.49	10.96	12.794
	6.63	9.82	8.3	9.06	8.91	8.544
0500	7.24	12.6	12.7	10.03	12.28	10.97
	11.15	18.19	16.11	13.82	17.4	15.334
0600	13.78	18.03	18.73	18.16	15.81	16.902
	12.15	17.18	15.62	14.67	20.3	15.984
0700	12.06	18.15	16.08	17.06	17.29	16.128
	11.58	17.86	16.87	14.97	15.91	15.438
0800	10.43	17.38	9.99	17.24	10.97	13.202
	6.47	10.31	8.56	10.07	8.79	8.84
0900	7.93	11.03	10.79	8.98	10.67	9.88
	46.09	50.56	40.47	48.25	39.07	44.888
1000	42.47	44.39	48.5	37.32	48.84	44.304
	73.18	77.92	74.75	71.27	74.39	74.302
1100	98.02	102.3	99.98	94.47	100.51	99.056
	115.2	124.82	119.85	120.52	118.32	119.742
1200	101.67	115.27	101.6	102.46	101.48	104.496
	97.57	99.59	101.82	103.13	96.91	99.804
1300	76.05	92.54	91.17	85.31	100.59	89.132
	49.41	59.53	58.3	50.73	58.76	55.346
1400	51.81	66.52	60.19	55.21	62.04	59.154
	53.09	68.04	63.77	60.89	64.41	62.04
1500	54.53	68.33	60.22	65.12	59.4	61.52
	51.37	66.24	54.57	55.52	59.94	57.528
1600	51.23	63.52	58	58.16	56.46	57.474
	54.15	68.11	58.62	59.27	61.2	60.27
1700	50.68	64.3	49.74	57.01	52.68	54.882
	37.32	48.56	44.5	46.39	42.05	43.764
1800	40.34	52.04	40.07	43.13	42.9	43.696
	34.64	44.48	38.47	38.27	40.95	39.362
1900	32.8	39.6	34.11	38.58	33.22	35.662
	29.1	32.72	27.83	28.82	32.59	30.212
2000	24.04	32.43	25.75	28.81	27.35	27.676
	23.92	30.03	25.63	26.88	25.51	26.394
2100	26.65	25.14	21.85	21.91	21.33	23.376
	17.46	21.32	19.19	18.56	19.64	19.234
2200	16.28	18.41	15.84	18.52	17.81	17.372
	13.02	16.69	12.72	13.28	13.44	13.83
2300	11.08	17.14	12.84	13.91	15.18	14.03
	14.05	14.52	13.55	14.31	20.9	15.466

Social Network SAS RV&V test case performance for server three

	Social Network Rvv Test Case Runs - Server 3					
	1	2	3	4	5	
Wall Time	100-%idle	100-%idle	100-%idle	100-%idle	100-%idle	Mean
0000	14.51	17.75	17.31	17.7	14.05	16.264
	16.47	16.49	16.46	19.15	19.36	17.586
0100	16.88	20.92	15.12	21.86	16.32	18.22
	17.97	26.47	18.68	22.73	17.96	20.762
0200	19.13	24.32	19.91	24.38	19.57	21.462
	18.99	21.99	20.71	20.87	20.75	20.662
0300	15.87	18.47	19	19.67	17.19	18.04
	15.11	17.01	17.28	15.87	15.31	16.116
0400	8.17	11.98	10.68	10.24	11.37	10.488
	9.45	10.87	12.78	14.27	9.22	11.318
0500	13.2	15.32	11.55	17.23	11.71	13.802
	16.81	18.72	16.54	18.34	17.25	17.532
0600	14.55	21.94	14.34	20.52	14.45	17.16
	14.88	22.1	16.96	17.12	18.58	17.928
0700	15.09	17.89	16.88	20.2	16.86	17.384
	14.38	16.91	14.82	13.14	15.26	14.902
0800	8.87	9.63	11.99	11.32	12.8	10.922
	8.41	10.48	8.39	12.57	8.85	9.74
0900	9.52	13.16	9.78	15.78	10.2	11.688
	16.55	22.06	12.49	37.18	13.94	20.444
1000	48.63	58.26	37.14	70.7	42.05	51.356
	60.62	57.1	64.63	84.67	55.94	64.592
1100	49.63	53.06	55.37	65.4	49.16	54.524
	39.32	39.37	44.48	37.69	40.5	40.272
1200	40.66	37.81	28.9	44.53	47.47	39.874
	48.5	48.22	45.12	44.76	45.84	46.488
1300	36.83	37.54	40.44	39.38	38.23	38.484
	30.92	41.14	30.33	42.51	34	35.78
1400	31.07	44.27	34.48	42.57	37.23	37.924
	46.09	40.13	35.92	55.9	44.77	44.562
1500	55.17	55.01	30.99	68.39	58	53.512
	57.42	67.48	54.15	69.51	59.54	61.62
1600	57.1	68.14	56.85	72.75	58.14	62.596
	57.64	70.85	56.12	63.8	58.42	61.366
1700	44.45	51.43	50.12	54.07	50.45	50.104
	41.56	48.79	43.27	50.39	46.55	46.112
1800	39	47.05	38.56	49.71	42.06	43.276
	36.03	44.13	38.46	42.55	41.45	40.524
1900	29.89	34.82	32.98	35.15	30.81	32.73
	26.97	34.59	27.07	34.67	29.95	30.65
2000	22.74	28.84	25.15	29.66	25.44	26.366
	27.17	26.73	23.45	26.37	25.2	25.784
2100	18.62	24.01	19.29	24.37	19.47	21.152
	17.4	22.91	18.47	21.52	20.37	20.134
2200	14.29	15.17	15.09	15.97	15.24	15.152
	12.98	15.12	12.5	16.63	12.72	13.99
2300	12.45	15.72	13.14	17.9	19.17	15.676
	13.12	14.58	12.81	15.68	13.36	13.91

Social Network SAS RV&V test case performance for server four

	Social Network Rvv Test Case Runs - Server 4					
	1	2	3	4	5	
Wall Time	100-%idle	100-%idle	100-%idle	100-%idle	100-%idle	Mean
0000						0
						0
0100						0
						0
0200						0
						0
0300						0
						0
0400						0
						0
0500						0
						0
0600						0
						0
0700						0
						0
0800						0
						0
0900						0
						0
1000						0
		99.95		99.92		39.974
1100	99.93	100	99.93	100	99.93	99.958
	95.42	100	100	84.35	98.2	95.594
1200	65.57	81.82	86.31	65.26	69.27	73.646
	45.28	54.4	62.05	46.25	55.17	52.63
1300	50.11	39.28	49.73	38.39	46.82	44.866
	29.99	37.32	35.96	34.29	32.57	34.026
1400	30.72	38.7	38.52	37.64	30.62	35.24
	32.21	41.06	37.33		29.84	28.088
1500			38.76			7.752
						0
1600						0
						0
1700						0
						0
1800						0
						0
1900						0
						0
2000						0
						0
2100						0
						0
2200						0
						0
2300						0
						0

Social Network SAS RV&V test case performance for server five

	Social Network Rvv Test Case Runs - Server 5					
	1	2	3	4	5	
Wall Time	100-%idle	100-%idle	100-%idle	100-%idle	100-%idle	Mean
0000						0
						0
0100						0
						0
0200						0
						0
0300						0
						0
0400						0
						0
0500						0
						0
0600						0
						0
0700						0
						0
0800						0
						0
0900						0
						0
1000						0
						0
1100						0
						0
1200						0
	65.63	98.57		94.19	93.23	70.324
1300		74.59	56.51			26.22
						0
1400						0
						0
1500						0
						0
1600						0
						0
1700						0
						0
1800						0
						0
1900						0
						0
2000						0
						0
2100						0
						0
2200						0
						0
2300						0
						0

Social Network SAS RV&V test case performance combined

	Social Network Rvv Test Case Runs - Combined					
	1	2	3	4	5	
Wall Time	100-%idle	100-%idle	100-%idle	100-%idle	100-%idle	Mean
0000	14.51	17.75	17.31	17.7	14.05	16.264
	16.47	16.49	16.46	19.15	19.36	17.586
0100	16.88	20.92	15.12	21.86	16.32	18.22
	17.97	26.47	18.68	22.73	17.96	20.762
0200	19.13	24.32	19.91	24.38	19.57	21.462
	18.99	21.99	20.71	20.87	20.75	20.662
0300	15.87	18.47	19	19.67	17.19	18.04
	15.11	17.01	17.28	15.87	15.31	16.116
0400	8.17	11.98	10.68	10.24	11.37	10.488
	9.45	10.87	12.78	14.27	9.22	11.318
0500	13.2	15.32	11.55	17.23	11.71	13.802
	16.81	18.72	16.54	18.34	17.25	17.532
0600	14.55	21.94	14.34	20.52	14.45	17.16
	14.88	22.1	16.96	17.12	18.58	17.928
0700	15.09	17.89	16.88	20.2	16.86	17.384
	14.38	16.91	14.82	13.14	15.26	14.902
0800	8.87	9.63	11.99	11.32	12.8	10.922
	8.41	10.48	8.39	12.57	8.85	9.74
0900	9.52	13.16	9.78	15.78	10.2	11.688
	16.55	22.06	12.49	37.18	13.94	20.444
1000	48.63	58.26	37.14	70.7	42.05	51.356
	60.62	157.05	64.63	184.59	55.94	104.566
1100	149.56	153.06	155.3	165.4	149.09	154.482
	134.74	139.37	144.48	122.04	138.7	135.866
1200	106.23	119.63	115.21	109.79	116.74	113.52
	159.41	102.62	107.17	91.01	101.01	112.244
1300	86.94	76.82	90.17	77.77	85.05	83.35
	60.91	78.46	66.29	76.8	66.57	69.806
1400	61.79	82.97	73	80.21	67.85	73.164
	78.3	81.19	73.25	55.9	74.61	72.65
1500	55.17	55.01	69.75	68.39	58	61.264
	57.42	67.48	54.15	69.51	59.54	61.62
1600	57.1	68.14	56.85	72.75	58.14	62.596
	57.64	70.85	56.12	63.8	58.42	61.366
1700	44.45	51.43	50.12	54.07	50.45	50.104
	41.56	48.79	43.27	50.39	46.55	46.112
1800	39	47.05	38.56	49.71	42.06	43.276
	36.03	44.13	38.46	42.55	41.45	40.524
1900	29.89	34.82	32.98	35.15	30.81	32.73
	26.97	34.59	27.07	34.67	29.95	30.65
2000	22.74	28.84	25.15	29.66	25.44	26.366
	27.17	26.73	23.45	26.37	25.2	25.784
2100	18.62	24.01	19.29	24.37	19.47	21.152
	17.4	22.91	18.47	21.52	20.37	20.134
2200	14.29	15.17	15.09	15.97	15.24	15.152
	12.98	15.12	12.5	16.63	12.72	13.99
2300	12.45	15.72	13.14	17.9	19.17	15.676
	13.12	14.58	12.81	15.68	13.36	13.91

Appendix C

Sale Day Test Case Results

Sale Day SAS-only test case performance for server three

	Sale Day Sas Test Case Runs - Server 3					
	1	2	3	4	5	
Wall Time	100-%idle	100-%idle	100-%idle	100-%idle	100-%idle	Mean
0000	25.34	19.59	27.44	15.55	27.01	22.986
	19.35	17.87	20.64	19.58	19.79	19.446
0100	26.14	24.3	26.54	19.8	23.99	24.154
	28.95	31.84	34.33	28.19	30.74	30.81
0200	29.28	29.02	35.16	29.67	41.5	32.926
	30.83	33.29	42.1	30.86	38.3	35.076
0300	24.79	28.43	34.58	28.27	34.81	30.176
	24.83	24.89	30.73	25.65	32.84	27.788
0400	19.79	22.83	30.98	23.15	26.47	24.644
	21.5	21.08	25.76	20.44	28.81	23.518
0500	25.34	22.45	34.31	22.82	28.04	26.592
	25.76	27.28	33.81	24.75	29.1	28.14
0600	27.98	27.51	37.56	24.33	29.78	29.432
	25.87	25.76	31.36	24.41	30	27.48
0700	25.59	27.42	29.55	26.49	32.88	28.386
	27.27	27.07	36.88	23.9	30.14	29.052
0800	36.79	31.2	36.55	29.36	36.65	34.11
	28.47	26.42	43.89	38.5	32.68	33.992
0900	29.02	34.88	34.73	26.48	35.18	32.058
	39.4	36.94	42.02	31.58	42.7	38.528
1000	40.71	42.93	50.82	38.94	43.65	43.41
	45.59	46.82	51.28	42.66	50.78	47.426
1100	52.78	52.66	55.15	49.36	61.1	54.21
	59.84	57.16	68.28	54.8	61.44	60.304
1200	51.26	49.96	61.71	60.61	65.51	57.81
	51.91	53.21	57.35	48.33	56.35	53.43
1300	50.6	51.97	60.76	48.6	61.5	54.686
	55.92	51.65	56.65	50.55	57.34	54.422
1400	52.41	52.85	60.02	50.85	59.12	55.05
	56.14	53.54	61.29	52.06	62.84	57.174
1500	49.4	49.61	60.36	48.37	58.78	53.304
	48.92	49.44	56.65	48.72	56.11	51.968
1600	50.6	46.76	56.27	46.58	55.37	51.116
	51.51	53.44	59.39	49.15	60.15	54.728
1700	42.56	42.6	54.1	49.37	57.35	49.196
	39.29	37.28	47.87	36.86	46.67	41.594
1800	54.42	46.93	49.64	39.1	57.92	49.602
	63.17	62.91	70.78	58.3	72.56	65.544
1900	53.03	53.64	65.83	58.84	66.71	59.61
	48.86	51.31	55.68	50.29	57.35	52.698
2000	42.36	43.24	57.52	46.43	51.83	48.276
	39.48	41.42	46.02	38.62	45.59	42.226
2100	32.48	36.96	40.91	37.16	37.85	37.072
	31.23	28.6	37.79	28.06	34.58	32.052
2200	25.08	26.53	33.16	28.08	32.95	29.16
	24.1	22.35	31.39	25.19	25.85	25.776
2300	25.05	25.03	29.42	24.8	27.79	26.418
	24.7	25.36	29.41	23.75	31.27	26.898

Sale Day SAS-only test case performance for server four

	Sale Day Sas Test Case Runs - Server 4					
	1	2	3	4	5	
Wall Time	100-%idle	100-%idle	100-%idle	100-%idle	100-%idle	Mean
0000						0
						0
0100						0
						0
0200						0
						0
0300						0
						0
0400						0
						0
0500						0
						0
0600						0
						0
0700						0
						0
0800						0
	99.92	99.95	70.64	99.95	72.47	88.586
0900	87.27	80.05	42.82	97.54	41.52	69.84
	40.72	42.3	37.47	39.02	41.68	40.238
1000	38.62	40.89	40.97	40.03	40.66	40.234
	44.12	47.81	43.32	42.22	48.78	45.25
1100	51.34	53.82	53.89	49.58	51.22	51.97
	58.32	57.98	56.22	52.46	56.03	56.202
1200	51.98	61.2	49.58	46.48	55.74	52.996
	51.52	49.34	50.43	45.82	49.31	49.284
1300	50.23	54.95	48.72	46.24	51.49	50.326
	51.24	61.52	48.01	46.31	51.68	51.752
1400	56.84	54.02	50.72	47.01	52.63	52.244
	54.55	55.8	53.94	49.06	55.33	53.736
1500	51.98	60.66	45.57	43.19	53.46	50.972
	48.52	53.05	47.06	43.89	49.51	48.406
1600	49.53	52.94	48.16	43.14	50.81	48.916
	52.32	54.79	50.46	46.36	51.33	51.052
1700	44.27	55.09	40.67	36.54	47.95	44.904
	38.9	44.18	36.49	35.24	38.77	38.716
1800	22.15	42	17.78	18.02	25.08	25.006
	0.35	10.39	0.28	0.25	0.6	2.374
1900	0.32	0.28	0.12	0.35	0.35	0.284
	0.23	0.18	0.12	0.33	3.6	0.892
2000	0.17	0.18	0.37	0.18	0.28	0.236
		0.17				0.034
2100						0
						0
2200						0
						0
2300						0
						0

Sale Day SAS-only test case performance combined

	Sale Day Sas Test Case Runs - Combined					
	1	2	3	4	5	
Wall Time	100-%idle	100-%idle	100-%idle	100-%idle	100-%idle	Mean
0000	25.34	19.59	27.44	15.55	27.01	22.986
	19.35	17.87	20.64	19.58	19.79	19.446
0100	26.14	24.3	26.54	19.8	23.99	24.154
	28.95	31.84	34.33	28.19	30.74	30.81
0200	29.28	29.02	35.16	29.67	41.5	32.926
	30.83	33.29	42.1	30.86	38.3	35.076
0300	24.79	28.43	34.58	28.27	34.81	30.176
	24.83	24.89	30.73	25.65	32.84	27.788
0400	19.79	22.83	30.98	23.15	26.47	24.644
	21.5	21.08	25.76	20.44	28.81	23.518
0500	25.34	22.45	34.31	22.82	28.04	26.592
	25.76	27.28	33.81	24.75	29.1	28.14
0600	27.98	27.51	37.56	24.33	29.78	29.432
	25.87	25.76	31.36	24.41	30	27.48
0700	25.59	27.42	29.55	26.49	32.88	28.386
	27.27	27.07	36.88	23.9	30.14	29.052
0800	36.79	31.2	36.55	129.31	36.65	54.1
	128.39	126.37	114.53	136.04	105.15	122.096
0900	116.29	114.93	77.55	65.5	76.7	90.194
	80.12	79.24	79.49	71.61	84.38	78.968
1000	79.33	83.82	91.79	81.16	84.31	84.082
	89.71	94.63	94.6	92.24	99.56	94.148
1100	104.12	106.48	109.04	101.82	112.32	106.756
	118.16	115.14	124.5	101.28	117.47	115.31
1200	103.24	111.16	111.29	106.43	121.25	110.674
	103.43	102.55	107.78	94.57	105.66	102.798
1300	100.83	106.92	109.48	94.91	112.99	105.026
	107.16	113.17	104.66	97.56	109.02	106.314
1400	109.25	106.87	110.74	99.91	111.75	107.704
	110.69	109.34	115.23	95.25	118.17	109.736
1500	101.38	110.27	105.93	92.26	112.24	104.416
	97.44	102.49	103.71	91.86	105.62	100.224
1600	100.13	99.7	104.43	92.94	106.18	100.676
	103.83	108.23	109.85	85.69	111.48	103.816
1700	86.83	97.69	94.77	84.61	105.3	93.84
	78.19	81.46	84.36	54.88	85.44	76.866
1800	76.57	88.93	67.42	39.35	83	71.054
	63.52	73.3	71.06	58.65	73.16	67.938
1900	53.35	53.92	65.95	59.17	67.06	59.89
	49.09	51.49	55.8	50.47	60.95	53.56
2000	42.53	43.42	57.89	46.43	52.11	48.476
	39.48	41.59	46.02	38.62	45.59	42.26
2100	32.48	36.96	40.91	37.16	37.85	37.072
	31.23	28.6	37.79	28.06	34.58	32.052
2200	25.08	26.53	33.16	28.08	32.95	29.16
	24.1	22.35	31.39	25.19	25.85	25.776
2300	25.05	25.03	29.42	24.8	27.79	26.418
	24.7	25.36	29.41	23.75	31.27	26.898

Sale Day SAS RV&V test case performance for server three

	Sale Day Rvv Test Case Runs - Server 3					
	1	2	3	4	5	
Wall Time	100-%idle	100-%idle	100-%idle	100-%idle	100-%idle	Mean
0000	17.85	26.33	25.38	19.27	22.41	17.766
	23.92	24.89	19.34	22.98	25.1	18.226
0100	27.59	31.64	21.51	29.09	32.65	21.966
	29.07	35.55	27.76	27.48	34.12	23.972
0200	25.99	34.34	29.23	31.99	35.4	24.31
	23.15	31.61	34.42	30.36	33.69	23.908
0300	23.27	30.47	28.53	29.01	31.03	22.256
	17.83	25.11	25.54	27.64	22.9	19.224
0400	18.92	23.48	21.1	21.29	22.81	16.958
	22.19	25.92	20.63	20.39	26.15	17.826
0500	22.9	28.04	22.94	27.86	27.96	20.348
	23.35	29.01	24.38	29.88	28.59	21.324
0600	23.99	28.57	24.9	26.87	28.72	20.866
	23.97	28.84	25.74	27.96	28	21.302
0700	25.27	27.93	25.15	29.55	29.37	21.58
	35.68	35.24	26.14	29.2	39.91	25.252
0800	48.31	46.9	31.95	44.57	48.02	34.346
	58.08	58.16	42.78	52.19	52.49	42.242
0900	66.36	65.34	58.12	61.25	31.46	50.214
	55.19	45.05	44.86	61.22	33.86	41.264
1000	29.1	27.81	24	37.68	45.38	23.718
	30.63	43.46	27.07	22.13	57.61	24.658
1100	49.06	50.44	48.77	42.45	59.59	38.144
	41.16	55.47	53.78	49.69	54.86	40.02
1200	48.86	54	47.5	43.25	51.31	38.722
	46.86	56.27	48.13	51.24	57.18	40.5
1300	48.84	50.28	51.77	50.85	55.04	40.348
	49.33	54.36	47.53	49.87	50.68	40.218
1400	50.48	57.86	52.03	53.1	55.57	42.694
	44.88	50.28	51.82	52.73	52.69	39.942
1500	49.39	51.52	48.22	44.24	48.92	38.674
	46.04	54.09	47.77	48.67	54.4	39.314
1600	46.96	52.08	47.75	50.38	52.24	39.434
	36.52	48.47	45.88	47.4	46.2	35.654
1700	35.7	41.94	45.14	40.47	42.89	32.65
	51.94	35.99	37.39	40.11	46.13	33.086
1800	57.04	36.06	36.69	53.31	66.86	36.62
	47.86	33.1	58.77	59.05	58.61	39.756
1900	43.7	31.23	55.5	49.23	52.32	35.932
	37.59	46.52	45.52	49.51	45.79	35.828
2000	35.16	41.66	46.13	41.1	39.66	32.81
	27.18	37.09	39.05	37.44	37.55	28.152
2100	26.37	32.41	32.77	29.97	32.51	24.304
	23.51	30.98	30.72	28.04	27.83	22.65
2200	21.93	25.9	26.02	24.74	27.24	19.718
	22.26	28.17	22.83	24.63	28.29	19.578
2300	23.3	26.69	24.86	23.56	25.51	19.682
	23.3	25.34	23.3	20.4	23.31	18.468

Sale Day SAS RV&V test case performance for server four

	Sale Day Rvv Test Case Runs - Server 4					
	1	2	3	4	5	
Wall Time	100-%idle	100-%idle	100-%idle	100-%idle	100-%idle	Mean
0000						0
						0
0100						0
						0
0200						0
						0
0300						0
						0
0400						0
						0
0500						0
						0
0600						0
						0
0700						0
						0
0800						0
					99.93	19.986
0900					100	20
	99.93	99.93	99.93	98.62	63.2	92.322
1000	100	100	100	100	45.18	89.036
	88.37	75.97	91.23	99.04	65.44	84.01
1100	60.38	48.57	51.07	69.66	62.04	58.344
	49.47	56.09	57.3	51.15	51.63	53.128
1200	50.59	48.79	55.88	45.86	54.7	51.164
	50.26	53.54	52.6	51.71	55.38	52.698
1300	47.9	52.93	51.58	52.98	52.73	51.624
	51.26	50.38	51.61	50.32	53.04	51.322
1400	49.77	51.69	53.67	53.94	55.23	52.86
	44.88	47.82	56.15	51.31	47.17	49.466
1500	47.07	45.98	52.63	48.17	53.53	49.476
	45.81	43.91	48.97	49.73	54.94	48.672
1600	44.27	50.3	49.82	50.22	53.56	49.634
	36.63	45.32	48.3	45.53	43.98	43.952
1700	33.51	38.2	48.62	39.04	40.58	39.99
		36.42	44.32	33.39		22.826
1800		33.67	37.87			14.308
		32.68				6.536
1900		29.03				5.806
						0
2000						0
						0
2100						0
						0
2200						0
						0
2300						0
						0

Sale Day SAS RV&V test case performance for server five

	Sale Day Rvv Test Case Runs - Server 5					
	1	2	3	4	5	
Wall Time	100-%idle	100-%idle	100-%idle	100-%idle	100-%idle	Mean
0000						0
						0
0100						0
						0
0200						0
						0
0300						0
						0
0400						0
						0
0500						0
						0
0600						0
						0
0700						0
						0
0800						0
						0
0900						0
						0
1000						0
		99.9				19.98
1100		72.3				14.46
	71.14		57.01	98.58		45.346
1200						0
						0
1300						0
					99.97	19.994
1400						0
						0
1500						0
						0
1600						0
						0
1700						0
						0
1800						0
						0
1900						0
						0
2000						0
						0
2100						0
						0
2200						0
						0
2300						0
						0

Sale Day SAS RV&V test case performance combined

	Sale Day Rvv Test Case Runs - Combined					
	1	2	3	4	5	
Wall Time	100-%idle	100-%idle	100-%idle	100-%idle	100-%idle	Mean
0000	17.85	26.33	25.38	19.27	22.41	22.248
	23.92	24.89	19.34	22.98	25.1	23.246
0100	27.59	31.64	21.51	29.09	32.65	28.496
	29.07	35.55	27.76	27.48	34.12	30.796
0200	25.99	34.34	29.23	31.99	35.4	31.39
	23.15	31.61	34.42	30.36	33.69	30.646
0300	23.27	30.47	28.53	29.01	31.03	28.462
	17.83	25.11	25.54	27.64	22.9	23.804
0400	18.92	23.48	21.1	21.29	22.81	21.52
	22.19	25.92	20.63	20.39	26.15	23.056
0500	22.9	28.04	22.94	27.86	27.96	25.94
	23.35	29.01	24.38	29.88	28.59	27.042
0600	23.99	28.57	24.9	26.87	28.72	26.61
	23.97	28.84	25.74	27.96	28	26.902
0700	25.27	27.93	25.15	29.55	29.37	27.454
	35.68	35.24	26.14	29.2	39.91	33.234
0800	48.31	46.9	31.95	44.57	48.02	43.95
	58.08	58.16	42.78	52.19	152.42	72.726
0900	66.36	65.34	58.12	61.25	131.46	76.506
	155.12	144.98	144.79	159.84	97.06	140.358
1000	129.1	127.81	124	137.68	90.56	121.83
	119	219.33	118.3	121.17	123.05	140.17
1100	109.44	171.31	99.84	112.11	121.63	122.866
	161.77	111.56	168.09	199.42	106.49	149.466
1200	99.45	102.79	103.38	89.11	106.01	100.148
	97.12	109.81	100.73	102.95	112.56	104.634
1300	96.74	103.21	103.35	103.83	107.77	102.98
	100.59	104.74	99.14	100.19	203.69	121.67
1400	100.25	109.55	105.7	107.04	110.8	106.668
	89.76	98.1	107.97	104.04	99.86	99.946
1500	96.46	97.5	100.85	92.41	102.45	97.934
	91.85	98	96.74	98.4	109.34	98.866
1600	91.23	102.38	97.57	100.6	105.8	99.516
	73.15	93.79	94.18	92.93	90.18	88.846
1700	69.21	80.14	93.76	79.51	83.47	81.218
	51.94	72.41	81.71	73.5	46.13	65.138
1800	57.04	69.73	74.56	53.31	66.86	64.3
	47.86	65.78	58.77	59.05	58.61	58.014
1900	43.7	60.26	55.5	49.23	52.32	52.202
	37.59	46.52	45.52	49.51	45.79	44.986
2000	35.16	41.66	46.13	41.1	39.66	40.742
	27.18	37.09	39.05	37.44	37.55	35.662
2100	26.37	32.41	32.77	29.97	32.51	30.806
	23.51	30.98	30.72	28.04	27.83	28.216
2200	21.93	25.9	26.02	24.74	27.24	25.166
	22.26	28.17	22.83	24.63	28.29	25.236
2300	23.3	26.69	24.86	23.56	25.51	24.784
	23.3	25.34	23.3	20.4	23.31	23.13

References

Amazon elastic load balancing. (2013). Retrieved from http://aws.amazon.com/elasticloadbalancing/

Amazon web services. (2013). Retrieved from http://http//aws.amazon.com

Apache tomcat. (2013). Retrieved from http://tomcat.apache.org

Apache wicket. (2014). Retrieved from https://wicket.apache.org

Arshad, N., Heimbigner, D., & Wolf, A. L. (2004). A planning based approach to failure recovery in distributed systems. *Proceedings of the 1st ACM SIGSOFT Workshop on Self-Managed Systems - WOSS '04*, 8–12. http://doi.org/10.1145/1075405.1075407

Artho, C., Barringer, H., & Goldberg, A. (2005). Combining test case generation and runtime verification. *Theoretical Computer …*, *336*, 209–234. http://doi.org/10.1016/j.tcs.2004.11.007

Baier, C., & Katoen, J.-P. (2008). *Principles of model checking*. The MIT Press. Retrieved from http://dl.acm.org/citation.cfm?id=1373322

Banks, J., Carson, J. S., Nelson, B. L., & Nicol, D. M. (2001). *Discrete Event System Simulation* (3rd ed.). Upper Saddle River, NJ: Prentice Hall.

Barringer, H., Havelund, K., Rydeheard, D., & Groce, A. (2009). Rule systems for runtime verification: A short tutorial. *Runtime Verification*, 1–24. Retrieved from http://www.springerlink.com/index/V88W047845RL4416.pdf

Behrmann, G., David, A., & Larsen, K. (2006). A tutorial on Uppaal 4.0. *… /darts/papers/texts/new-Tutorial. …*. Retrieved from http://www.csi.uottawa.ca/~bochmann/ELG7187C/CourseNotes/PerformanceModeling/Timed-Automata/UPPAAL - new-tutorial.pdf

Calinescu, R., Ghezzi, C., Kwiatkowska, M., & Mirandola, R. (2012). Self-adaptive software needs quantitative verification at runtime. *Communications of the …*, *55*(9), 69. http://doi.org/10.1145/2330667.2330686

Calinescu, R., & Kwiatkowska, M. (2009). CADS*: Computer-aided development of self-* systems. *Fundamental Approaches to Software …*, 1–4. Retrieved from http://www.springerlink.com/index/y0863g33hv748262.pdf

Cheng, B., Lemos, R. De, & Giese, H. (2009). Software engineering for self-adaptive systems: a research roadmap. *Software Engineering for …*, 1–26. Retrieved from http://www.springerlink.com/index/H380742725036312.pdf

Chung, H., & Park, J. (2009). Consumer motivation and site transfer behavior: weblog analysis for online service. *2009 IEEE/INFORMS International Conference on Service Operations, Logistics and Informatics*, 78–84. http://doi.org/10.1109/SOLI.2009.5203908

Creeger, M. (2009). Cloud computing: an overview. *Queue*, *7*(5), 3. http://doi.org/10.1145/1551644.1554608

Dahm, W. (2010). *Report on Technology Horizons A Vision for Air Force Science & Technology During 2010-2030* (Vol. 1). Retrieved from http://www.af.mil/shared/media/document/AFD-100727-053.pdf

Dashorst, M., & Hillenius, E. (2009). *Wicket in action*. Greenwich: Manning Publications.

de la Iglesia, D. G., & Weyns, D. (2013). Guaranteeing robustness in a mobile learning application using formally verified MAPE loops. *2013 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 83–92. http://doi.org/10.1109/SEAMS.2013.6595495

Feldt, R., Torkar, R., Ahmad, E., & Raza, B. (2010). Challenges with software verification and validation activities in the space industry. *2010 Third International Conference on Software Testing, Verification and Validation*, 225–234. http://doi.org/10.1109/ICST.2010.37

Fu, X., Zou, P., Jiang, Y., & Shang, Z. (2007). QoS consistency as basis of reputation measurement of web service. In *The First International Symposium on Data, Privacy, and E-Commerce (ISDPE 2007)* (pp. 391–396). IEEE. http://doi.org/10.1109/ISDPE.2007.23

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns*. Reading, MA: Addison-Wesley Publishing Company.

Goldberg, A., Havelund, K., & McGann, C. (2005). Runtime verification for autonomous spacecraft software. *IEEE Aerospace Conference Proceedings*, *2005*, 507–516. http://doi.org/10.1109/AERO.2005.1559341

Gupta, M., Mittal, H., Singla, P., & Bagchi, A. (2014). Characterizing Comparison Shopping Behavior : A Case Study. http://doi.org/10.1109/ICDEW.2014.6818314

Heimdahl, M. P. E., & Leveson, N. G. (1996). Completeness and consistency in hierarchical state-based requirements. *IEEE Transactions on Software Engineering*, *22*(6), 363–377. http://doi.org/10.1109/32.508311

Herman, I. (2014). Web Ontology Language (OWL). Retrieved from http://www.w3.org/2004/OWL/

Jureta, I. J., Borgida, A., Ernst, N. A., & Mylopoulos, J. (2010). Techne: towards a new generation of requirements modeling languages with goals, preferences, and inconsistency handling. *2010 18th IEEE International Requirements Engineering Conference*, 115–124. http://doi.org/10.1109/RE.2010.24

Kephart, J., & Chess, D. (2003). The vision of autonomic computing. *Computer*, (January), 41–50. Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1160055

Lamsweerde, A. Van. (2000). Requirements Engineering in the Year 00 : A Research Perspective. In *Proceedings of the International Conference on Software Engineering* (pp. 5–19). Limerick, Ireland: ICSE. http://doi.org/10.1109/ICSE.2000.870463

Lapouchnian, A., Yu, Y., Liaskos, S., & Mylopoulos, J. (2005). Requirements-driven design of autonomic application software. In *Proceedings of the Workshop on Design and Evolution of Autonomic Application Software* (pp. 1–7).

Laurent, O. (2010). Using formal methods and testability concepts in the avionics systems validation and verification (v&v) process. *… , Verification and Validation (ICST), 2010 Third …*, 1–10. http://doi.org/10.1109/ICST.2010.38

Lemos, R. De, Giese, H., Muller, H. a., & Shaw, M. (2011a). Software Engineering for Self-Adaptive Systems: A Second Research Roadmap (Draft Version of May 20, 2011), (October 2010). Retrieved from http://didattica.uniroma2.it/assets/uploads/corsi/144538/dagstuhl-2ndSelfAdaptRoadmap.pdf

Lemos, R. De, Giese, H., Muller, H. A., & Shaw, M. (2011b). Software engineering for self-adaptive systems: a second research roadmap. *InProceedings of Software Engineering for Self-Adaptive Systems*. Retrieved from http://vesta.informatik.rwth-aachen.de/opus/volltexte/2011/3156/

Merz, S. (2001). Model checking: A tutorial overview. *Modeling and Verification of Parallel Processes*. Retrieved from http://www.springerlink.com/index/111t8re3ww3l5nbt.pdf

Mitchell, T. (1997). *Machine learning*. Boston, MA: McGraw-Hill.

Qureshi, N. A., Jureta, I. J., & Perini, A. (2011). Requirements engineering for self-adaptive systems : core ontology and problem statement. In *CAiSE'11: Proceedings of the 23rd international conference on Advanced information systems engineering* (pp. 33–47). Berlin Heidelberg: Springer-Verlag.

Qureshi, N. A., Jureta, I. J., & Perini, A. (2012). Towards a requirements modeling language for self-adaptive systems. *REFSQ'12 Proceedings of the 18th International Conference on Requirements Engineering: Foundation for Software Quality*, 263–279. Retrieved from http://www.springerlink.com/index/0Q2KU352L70732T4.pdf

Rosenstein, M. (2000). What is actually taking place on web sites : e-commerce lessons from web server logs. *EC '00: Proceedings of the 2nd ACM Conference on Electronic Commerce*, 38–43.

Russell, S., & Norvig, P. (2010). *Artificial intelligence: a modern approach* (3rd ed.). Upper Saddle River, NJ: Pearson.

Salehie, M., & Tahvildari, L. (2009). Self-adaptive software: landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems*, *4*(2), 1–42. http://doi.org/10.1145/1516533.1516538

Sawyer, P., Bencomo, N., & Whittle, J. (2010). Requirements-aware systems. *International …*, 95–103. http://doi.org/10.1109/RE.2010.21

Sipser, M. (2006). *Introduction to the theory of computation* (2nd Editio). Boston, MA: Thomson Course Technology.

Tamura, G., Villegas, N., Muller, H., Sousa, J. P., Becker, B., Karsai, G., … Wong, K. (2012). Towards practical runtime verification and validation of self-adaptive software systems. *… -Adaptive Systems …*, 108–132. Retrieved from http://hal.archives-ouvertes.fr/hal-00709943/

Ubuntu. (2013). Retrieved from http://www.ubuntu.com

Unified modeling language. (2013). Retrieved from http://www.uml.org

Villegas, N. M., Muller, H. a., & Tamura, G. (2011). Optimizing run-time SOA governance through context-driven SLAs and dynamic monitoring. *2011 International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems*, 1–10. http://doi.org/10.1109/MESOCA.2011.6049036

Welsh, K., Sawyer, P., & Bencomo, N. (2011). Towards Requirements Aware Systems : Run-time Resolution of Design-time Assumptions. In *26th IEEE/ACM International Conference on Automated Software Engineering* (pp. 560–563).

Weyns, D., Iftikhar, M. U., de la Iglesia, D. G., & Ahmad, T. (2012). A survey of formal methods in self-adaptive systems. In *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering - C3S2E '12* (pp. 67–79). New York, New York, USA: ACM Press. http://doi.org/10.1145/2347583.2347592