Automated State-Space Exploration for Configuration Management of Service-Oriented Applications

Mike Smit, Eleni Stroulia
Department of Computing Science
University of Alberta
Edmonton, Canada
Email: msmit,stroulia@cs.ualberta.ca

***Abstract*—Configuration management is a complex task, even for experienced system administrators, which makes self-managing systems a desirable solution. Self-management implies the need for a model based on which configuration changes may be decided. In previous work, we described a method for constructing a state-transition model of application behavior, by observing the application in simulation. This method relied on an expert to manage the (simulated) application in order to collect the necessary observations for constructing the model. However, that method was agnostic about (a) the size of the system space space as implied by the granularity of the observations, and (b) the sufficiency of the actual observations collected for understanding the application in a variety of configurations and environments. In this paper, we replace the (expensive) expert domain knowledge with automatic approaches to ensuring coverage of the application, and demonstrate the superiority of this approach. We present empirical data regarding state space and granularity to explore the use of state models for understanding applications.**

***Keywords*-self-configuring software, autonomic computing, simulation, service-oriented architectures, adaptation**

I. Introduction

The demand for dynamic, distributed, component-driven software has led to increased adoption of service-oriented software systems. In such systems, additional copies of service instances may be deployed at run time; new services may be removed and new services may be introduced in an existing composition; the provisioned network bandwidth can be changed. This is why deploy-time and even run-time configuration decisions are required to manage the system in the face of varying business requirements, service providers and consumers, and available resources [1]. The frequency of changes and decisions required for just-in-time resource provisioning practically precludes manual intervention; instead, a self-managed system is desired [2], [3], [4].

Previous work ([5]) described a methodology for autonomic decision-making for configuration management. The target application was simulated using our services-aware simulation framework, then "exercised" by an expert user who changed the configuration in real time to obtain performance traces in a variety of scenarios. A state-transition model of the application's behavior was created from these traces. An autonomic manager made configuration decisions based on the future behavior predicted by the model. This approach was validated in both simulation and a real-world cloud computing environment, and was able to produce results superior to fixed configurations at lower cost, and was comparable to an expert system administrator attempting to change configurations at run-time. The state-transition model methodology is described in Section II, and the results are included here as a baseline to which we can compare this refined approach.

Though effective, this approach raised several unanswered questions. First, an expert user was required to "exercise" the simulation. In this paper, we replace expert knowledge with an automatic algorithm that tests the simulation in a variety of realistic simulations. Starting from a single simulation, each time a new state is encountered, a set of new simulations is launched to test mutations of that state - increasing load, adding servers, and so on. This approach does not require (expensive) expert domain knowledge, discovered more states and transitions, had fewer "missing" states at run-time, and produced better results in a new set of experiments. Statistics about the automatic generation of state-transition models are also examined.

Second, the challenge of defining a tractable search space with sufficient granularity to differentiate between states without impairing the decision-making ability of the algorithm was not addressed. Here, we explore various abstraction methods for producing states from raw performance metrics, and how each impacts the state-transition model and the autonomic manager that uses it. We conclude that though intuitively a highly-granular model is desirable, in actuality this makes it more likely that the states encountered at run-time will be considered new and different.

The state model and its use in the autonomic manager is described in Section II. We then describe our automated systematic exploration of the application's possible states in Section III. We then describe a series of experiments and their results, comparing manual versus automatic model construction at varying levels of abstraction, Section IV. The relation of our work to the state of the art is reviewed in Section V. Section VI summarizes our contributions.

II. System Behavior Modeling and Adaptation

Our approach to autonomic adaptation [5] involves three main steps. The first step relies on our simulation framework [6], [7] for producing a model of the system under examination that reflects the system services (each with a performance profile that realistically reflects the performance of its corresponding service) and the connections and types of invocations among them. This model is used to generate system behavior metrics, through a sequence of simulations systematically exploring varying configurations, loads and SLAs. In this paper, the example subject system is a text-processing service called TAPoRware (referred to as the *simulated system*). The motivating reconfiguration scenarios envision the adaptation of the number of servers on which the system services are deployed, as well as changes to the number of deployed instances of its services.

The second step creates the system's behavioral state-transition model from the data collected by the simulation step. To that end, the metrics recorded in each monitoring step are discretized to produce a more coarse-grained representation of the recorded data, which is clustered into classes, each of which represents a behavioral *state*. Two states are related with a *transition* between them if an instance of the source state is followed in the simulation by an instance of the destination state. These transitions essentially represent changes in demand (when the system load metrics increase) or changes in the system configuration (when the system configuration changes).

The final step of our method involves the autonomic manager, which, at run time, monitors the simulated system by taking periodic snapshots of its load and configuration. This data enables it to identify in which of the behavioral states it is, thus tracking the progress of the system through the state-transition model. When the system is in a state known to violate (or to potentially lead to violations of) its SLA, a search for a configuration change transition (or a series thereof) that will lead to a satisfactory state is initiated. The required changes are executed (*i.e.*, the subject system is reconfigured) and monitoring resumes. Our evaluation was conducted in both simulation and in a real-world cloud computing environment, and showed promising results.

States, transitions, and the autonomic manager are described in more detail in the following subsections.

A. States

Every observation recorded during the simulation of the system model is called a snapshot. For each snapshot, a *snapshot descriptor* is calculated and recorded. A snapshot descriptor has three parts. The first part defines the state's compliance with the SLA: *Satisfactory*, *Unsatisfactory*, or *Boundary* (the *SUB metric*). Boundary states are states which meet the SLA, but which have transitions to unsatisfactory states[1]. The second element is a set of configuration-related parameters. The third captures the current environment (number of incoming requests, *etc.*). Finally, the fourth is a set of performance metrics:

$$S = \langle (\mathrm{S} \mid \mathrm{U} \mid \mathrm{B}), \langle \mathrm{Configuration} \rangle, \langle \mathrm{Environment} \rangle, \langle \mathrm{Metrics} \rangle \rangle$$

The internal structure of the snapshot descriptor, namely the actual metrics included in the configuration-parameter and performance-metrics set, depends on the subject system. The challenge is to identify a tuple that includes measurable information sufficient to characterize the performance of the system without including extraneous information. The snapshot descriptor used for TaporSim, our test-bed system, is shown in Table I.

```
1) SUB metric
2) Configuration
   a) Number of servers.
   b) Number of processors.
   c) Limit on concurrent requests
3) Environment
   a) Requests / 5 seconds (sliding window)
   b) Request size / 5 seconds (sliding
      window)
4) Performance
   a) Total queued requests
   b) CPU utilization (over all processors)
   c) Average response time (rolling)
   d) Response time standard deviation
   e) Largest queue at single servers
   f) Highest CPU utilization at single server
   g) Smallest queue at single server
   h) Lowest CPU utilization at single server
   i) Throughput (responses / minute)
   j) Memory footprint
```

Table I
The elements of a snapshot descriptor for our testbed application.

The collected snapshot descriptors are then clustered into equivalence classes, called *states*. The snapshot descriptors within each state are "similar enough", in that there is no significant difference in their environment, performance, or configuration metrics[2]. Each state is annotated with a cost, the cost of the cheapest configuration of all its snapshot descriptors. The function used to determine the configuration costs could be the cost of hardware, the cost of maintenance, the "green-ness", *etc.*

Whether two values of a performance metric are "similar enough" depends on the level of precision at which metrics are examined. At the highest level of precision, any variation in any metric is considered a change. If the metrics involved are measured using real numbers (and not restricted to a

[1]Note that this value is not static: the same simulation data with a different SLA will produce different states as the SUB metric may change.

[2]The SUB metric is ignored in the clustering process.

range of integers), the state space is theoretically infinite. To restrict the size of the state space while still producing meaningful results, we discretize ranges of values into "windows" where all values within a window are considered equal to each other. For enumerated performance metrics, each enumerated value defines a "window"; for example, the SUB metric has 3 possible values and has window size 3. For numeric metrics, the size of the window is chosen to give the desired number of windows and therefore the desired state space, while remaining appropriate for that metric. We tested three different discretization strategies.

1) A fixed window size based on the theoretical range of values. For example, average response time can range from 0 seconds to 30 seconds (the timeout point). The window size is then $\frac{\text{30 seconds}}{\text{desired \# of windows}}$.
2) A fixed window size based on the actual or expected range of values. Based on the collected simulation data, the actual range of observed values is divided by the desired number of windows, to produce the window size. For example, the average response time might actually range from .2 to 5.5 seconds, so $\frac{5.5-.2\text{ seconds}}{\text{desired \# of windows}}$.
3) A window size chosen to ensure an equal number of values in each window. If $k = \frac{\text{\# of values}}{\text{desired \# of windows}}$, then the first k values make up the first window, the next k values make up the second, and so on.

When constructing a state-transition model from a set of observations recorded during a completed simulation, any of the methods can be used. When, as in this paper, the total number of values and their range is not known *a priori*, only the first discretization method is directly useable.

Once the number of windows for each metric w_m is decided, if d is the number of metrics included in the descriptor, then we can define the theoretical state space size as the product of the window size of each descriptor. For example, consider a snapshot descriptor that includes only the SUB metric ($w_1 = 3$), a CPU utilization metric, and a memory usage metric. CPU utilization ranges from 0-100 (percent); if we use a window size of 5, there are 20 windows. In our example, memory usage ranges from 0-4096 (MB); if we use a window size of 512, there are 8 windows. The state space for the example is therefore $3 \times 20 \times 8 = 480$.

B. Transitions

The model includes two primary types of transitions between states. First, *need-based transitions* (n-transitions) occur when the environment changes, and the software is asked to provide out-of-the-ordinary service (this new level may be the new "ordinary"). In our ongoing example of performance, this takes the form of a load change: increased requests, increased request size, network congestion, and so forth. Thus, a need-based transition implies that there is a pair of states, s_{src} and s_{dest}, such that (a) s_{src} is followed

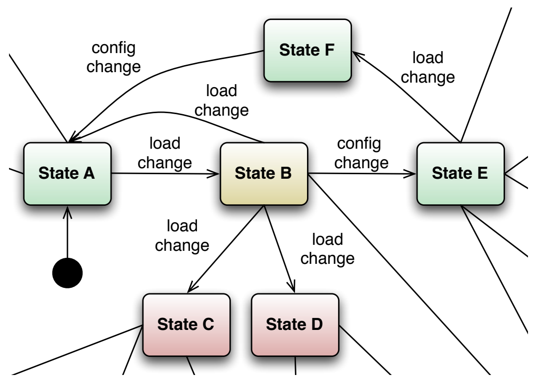

Figure 1. A hypothetical fragment of a state diagram; states can be satisfactory, unsatisfactory, or boundary; transitions occur when the load changes or the configuration changes.

by s_{dest} in the simulation and (b) the load in s_{dest} has significantly different load metrics associated with it from the load metrics associated with s_{src} (namely the former load metrics fall in different windows from the latter load metrics). In addition to the metrics that change between s_{src} and s_{dest}, a $n - transition$ is characterized by its cost differential, namely $cost(s_{src}) - cost(s_{dest})$.

The second type of state transitions, *configuration-based transitions* (c-transitions), occur when the configuration changes (*i.e.*, s_{dest} has a different configuration from s_{src}). This transition may be caused by a system administrator or the autonomic manager. These transitions are also annotated by their cost differential. However, they also imply a secondary cost, namely the cost of actually making the change. For example, adding a second server involves a deployment cost in addition to the ongoing maintenance and energy costs implied by the additional server.

As each state may have any number of these transitions, the number of occurrences of a given transition t in the simulation traces is calculated. Then, the estimated probability of a given transition occurring is given by its occurrence count relative to the other transitions.

The process also identifies an additional type of transitions, *performance transitions*. While a configuration and the incoming load on the software will define the system's performance, it generally takes some time after a load or configuration change for the system to stabilize, during which the configuration and load will not change, but the performance will (*i.e.*, s_{dest} has the same configuration and load as s_{src} but different performance metrics). The process recognizes these transitions, but they do not alter the conceptual model: if there is a chain of performance transitions following a configuration or need transition, this "chain" can be followed to the eventual end state.

Figure 1 depicts a hypothetical fragment of a state diagram. The system begins in State A, a satisfactory state. There are three transitions out of this state. The transition to

State B is a load-change transition (need-based). An implicit loopback transition represents the possibility that the system remains in this state. If the load changes, the system is in a boundary state, with direct transitions to unsatisfactory states, namely States C and D (with probability above a given threshold). At this point the load could change again back to where it started, and move the system to State A. Or, it could continue to increase and move the system into one of these two unsatisfactory states. The final option is for a manager to intervene and change the configuration, moving the system to the satisfactory State E. Note here that if the load decreases back to its original levels, the system moves to State F, where a configuration change will move us back to the cheaper State A.

C. Autonomic Management

Our autonomic-management method makes decisions by monitoring the application, periodically recording a snapshot of its metrics, and classifying the snapshot as an instance of a corresponding state in the state-transition model of the system behavior. The decision on whether changes are needed rests on the SUB metric of the identified state.

If the current system state is classified as an unsatisfactory state, the state space is searched, starting from the current state and following only c-transitions until a satisfactory state is found. The search algorithm used for that purpose is modified from A*. First, the state space is further abstracted to provide a substantially smaller *abstracted state* space. An iterative-deepening depth first search (IDDFS) [8] algorithm is used to search at this level. IDDFS finds states within a few levels of the start state quickly with space-complexity similar to DFS. It can be aborted and will return the best result found so far (*e.g.*, a state with improved performance but a boundary state) if it does not find a satisfactory state in the time allotted. The abstracted state returned, as well as any on the path to it, are expanded to their component states, and another IDDFS is run on this restricted set of states. The transitions needed to get to the goal state are saved. Any c-transitions identified by the search algorithm are executed. A timestamp of the last change is recorded.

If the current state is classified as a boundary state, a search is initiated just as for the unsatisfactory state; however, the configuration changes implied by the transition sequence are only executed if the identified target state is superior to the current boundary state (an S-state or a B-state with improved performance and/or lower cost).

If the current state is classified as a satisfactory state, a search for a path of c-transitions that leads to another satisfactory state is initiated. If a path is identified, it is executed if (a) the time elapsed since the last change is sufficient (we chose 5 minutes), and (b) the new state is cheaper than the current state. This is intended to remove over-provisioned resources while avoiding "churn" of repeated adds/removes for performance levels at or near SLA levels.

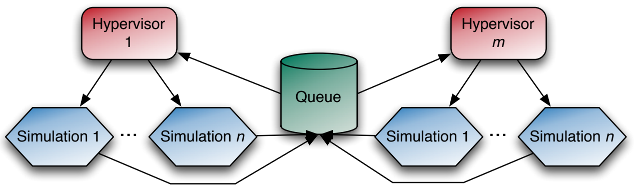

Figure 2. The high-level architecture of the systematic exploration.

If the SLA changes, each of the recorded snapshot descriptors is revisited to recalculate its associated SUB metric. If after this process the system is found to be in an unsatisfactory state, the adaptation algorithm above is invoked in order to transition to a satisfactory state. Recall that every unsatisfactory state must have at least one c-transition to a satisfactory state; if new unsatisfactory states are added as a result of the SLA change, this requirement might be violated and additional simulations may be required to update the state-transition model.

III. The Model-Construction Process

Let us now discuss the implementation of the model-construction process. The simulation records the system behavior at 5-simulated-second intervals (frequent, but not so frequent that it impairs simulation performance). As each new snapshot descriptor is recorded, its abstracted representation is calculated. If a state already exists for this abstract representation, the original descriptor is clustered in it, and the state-population counter is incremented. Otherwise, a new state entry is added to the model.

If two subsequent snapshot descriptors are not equivalent, the model-construction process analyzes what has changed: the configuration, the environment, or both. For configuration changes, the deployment cost is calculated and a c-transition is created. For load changes, a n-transition is created. When both types of metrics have changed, first a n-transition is created (where only load-related elements in the snapshot descriptor change), creating a new intermediate state if an equivalent state does not already exist. Next, a c-transition is also created (again creating a placeholder if need be). Two transitions are considered equivalent if they are both the same type (n or c) and they transition between the same two states; a counter is incremented when duplicates are encountered. The probability of each transition out of a given state is determined after state-transition detection.

The result of the model-construction process is a knowledge base containing the system states, their associated snapshot descriptors, and the transitions between them. In our previous work, the model construction relied on observations recorded while an expert monitored the performance and managed the configuration of a system. In this paper, observations are recorded while the performance and configuration of the system are changed automatically.

A. *Expert-driven Model Construction*

Using the interactive version of the simulation, an expert varied the request arrival rate of a stochastic but realistic request generator, changing the number of servers manually to adequately service the requests. The goal was to execute as many different scenarios as possible. The result was 18 hours of simulation traces with metrics captured every 5 seconds. This is a relatively small training set and was not intended to be comprehensive.

From this data set, 1,465 states were identified, with 2,698 total transitions. 1,304 of the states were unsatisfactory. The number of abstracted states used in the search process was 89, with 472 transitions. Given the values used in discretization, the theoretical state space is approximately 57,024,000. The majority of the theoretical state space is also theoretically impossible to encounter in reality.

B. *Systematic Exploration of the State Space*

To systematically explore the configuration/performance state space of a service-oriented system, we consider its behavior in all realistically possible states. There are four actions that influence the state of an application: adding a server, removing a server, increasing the rate at which requests arrive, and decreasing said rate. The simulated system must be observed to see what happens in each state when each of these actions occurs.

To achieve this, we start a simulation, and monitor its behavior at 5-second intervals. The state of the application at each interval is identified. If this state has already been seen, execution continues. If this is a new state, the steps taken to reach this state are stored: the application's topology, the configuration file, all requests sent, and any actions taken previously (and the time or state at which these actions occur). A new experiment is created for each action, where an experiment is a tuple e = [id, topology, configuration, requests, actions]. The actions element includes all actions taken up to this point in the simulation, as well as the new action we wish to test. A new experiment is not created if the action in this state would breach a threshold: a minimum / maximum number of servers, or a minimum / maximum request arrival rate. The new experiments are placed on a FIFO queue (implemented in a database), awaiting execution. The originating simulation continues execution.

A simulation hypervisor is responsible for pulling experiments from this queue and executing them. The system architecture (Figure 2) allows for any number of hypervisors, each running any number of simulations (the limiting factor is system resources). The hypervisor is responsible for creating the environment described by the experiment, including the same topology, instructing the simulation to simulate the exact requests used in the originating simulation (then switching to stochastically generated requests), and informing the simulation of the times and states where actions would be performed. It also monitors and stores the output of the simulation, and waits for the simulation to exit before launching the next experiment.

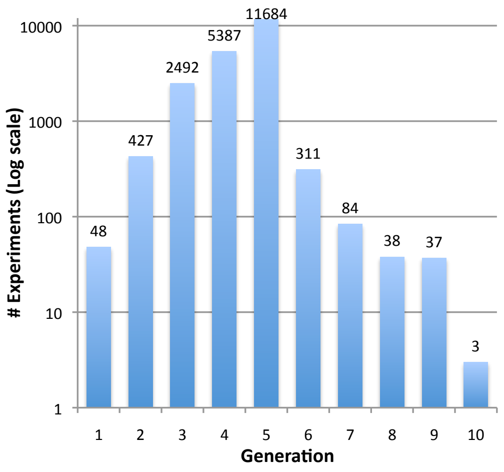

Figure 3. Number of simulations in each generation (log scale).

As each simulation runs, new states are encountered and the queue grows. Each simulation will run until one of three possible normal termination points is reached: (a) the simulated application enters an implausible or failed state (e.g. the simulated server is so overloaded that it fails); (b) all actions have been taken but no new states have been seen for 20 minutes of simulated time[3]; and (c) the time at which the target state was reached in the originating simulation is exceeded by one hour, but the target state is not reached. In the third case, the experiment will be re-queued and attempted again. For each new state encountered, 2-4 new simulations are added to the queue.

The initial simulation is started and ended manually, and is called "experiment 0". This experiment forms *generation* 0. Each experiment it enqueues belongs to generation 1, and will enqueue experiments belonging to the 2nd generation, and so on. In our experiments, the mutations continued out to the 10th generation, after which no new experiments were enqueued (this is not a limit we imposed, it is simply when no new states were encountered).

Figure 3 shows the number of experiments belonging to each generation. Figure 4 shows the number of experiments enqueued by each generation (except the 0th generation). The number of new experiments enqueued at each generation approaches 0 very quickly. Other than the start point, the only manual intervention occurred during the fifth generation when 2,914 experiments were manually pruned from the queue, as they were guaranteed to enter implausible scenarios immediately.

A total of 17,122 experiments were queued and executed over a 105 hour period on commodity hardware (Core 2 Duo

[3]Based on our observations, when no states are seen after 20 minutes, the simulation has stagnated

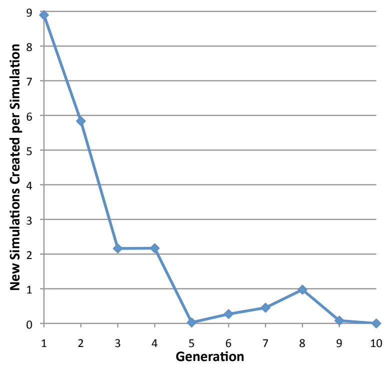

Figure 4. Average number of simulations enqueued per simulation for each generation.

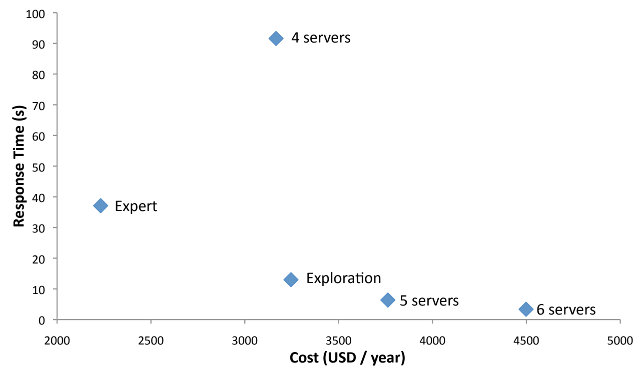

Figure 6. The performance / cost trade-offs for the test configurations.

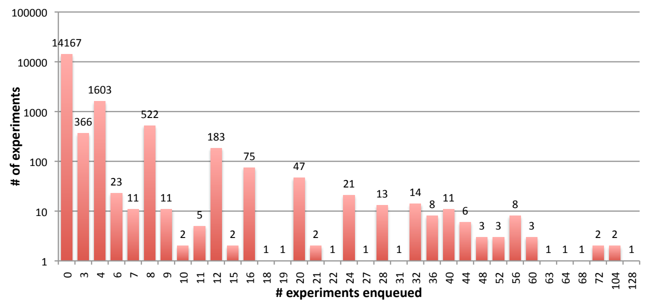

Figure 5. The number of experiments (y, log scale) that enqueued the given number of experiments (x).

processor, 4GB RAM). The majority of these experiments (82.7%) did not find any new states; 94.2% found only one new state (Figure 5). The total number of states identified was 5,123; 3,369 of them were unsatisfactory; there were 34,139 transitions in total. There were 720 abstracted states with 14,798 transitions.

IV. Experiments

The first question we are interested in examining is the relative quality of the models constructed (a) by the expert's monitoring of the simulated system and (b) by the automated state-space exploration manager, for the purpose of supporting autonomic run-time configuration management. To that end, we designed an experiment in which the simulated application is monitored and managed by adding or removing servers in response to changing request loads. The service level objective was a very ambitious response time of 10 seconds, with a maximum queue length of 20 requests. (This is reasonable for CPU-intensive analysis of texts up to 7 MB in size).

For the input requests, an individual not familiar with this project generated 3 hours of request traffic using our modifiable stochastic load generator (variable data set). The requests (size, type, and time) were recorded and used as input to the simulated application in 5 different conditions: a fixed number of servers (4, 5, and 6, respectively), an autonomic manager deploying 1-6 servers based on the **expert** decision model, and the autonomic manager based on the automatic **exploration** decision model.

For each of the above five conditions, we recorded average response time as the performance measure. We also calculated configuration cost, based on total active CPU time (loosely, a cloud-computing scenario[4]). A server contributed to the cost if it was configured to process requests, whether it actually received requests or not. A delay was imposed to emulate start-up time for a new server, and another delay was imposed before server shutdown was initiated to give the autonomic manager time to reverse its decision.

Figure 6 shows the results when comparing the expert and exploration conditions. The automatic exploration approach provided superior performance, servicing requests on average 3 times faster, with a cost increase of only 45%. This is when using the same discretization method; the expert data set does not perform as well using this discretization method as it does when using the other two. Another comparison metric is the number of unknown states encountered during the simulated tests. The expert data set encountered 730 new states (49.8% of the original total), 723 of them unsatisfactory states where remedial actions could have been taken. The exploration data set encountered 188 new states (3.6% of the original total, 185 unsatisfactory). (Most of the new states were beyond the threshold of what was considered plausible when generating the state-transition model; the thresholds could be adjusted).

The second set of comparisons examines high versus low granularity. If the window size is too large, it is not possible to understand what is actually happening in the application at each point in time, since one window may

[4]The estimated costs depend on the price model assumed; in this work, we assume a hardware-leasing model. Given alternative cost-calculation models, the estimated costs would be different. Our work is independent of any particular cost models.

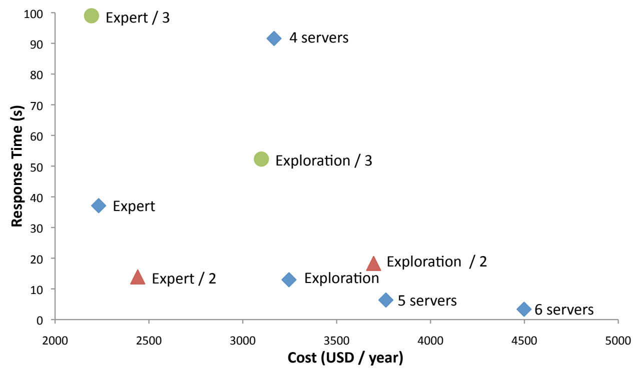

Figure 7. The performance / cost trade-offs for the test configurations.

	WS 1	WS 2	Dec.	WS 3	Dec.
Total	57,000,000	150,000	99%	14,000	91%
Expert	1465 (.003%)	312 (.21%)	79%	231 (1.7%)	26%
Auto	5123 (.009%)	3009 (2.0%)	41%	1306 (9.3%)	56%

Table II
THE DECREASE OF STATE SPACE AND THE COVERAGE WITH VARIED GRANULARITY.

contain significantly different snapshot descriptors. If the window size is too small, the state space will be larger and as the number of transitions increases, the computation time involved in analyzing the possible future alternatives will also increase. It is also not desirable to be too precise - this differentiates between states that are essentially the same.

To test for appropriate levels of granularity, we use the same simulation trace data, but changed the window sizes, both doubling and tripling the window size (except for discrete values). The changes to state space and granularity are shown in Table II. We show the total theoretical number of states for the three window sizes used, and the coverage of this theoretical space. Note that many states theoretically possible will not occur in reality. The exploration-generated model has better coverage and lost relatively fewer states as the window size grew; from this, we can conclude that the automatically driven traces had a greater variety of states.

The impact on the autonomic modification of the system is shown in Figure 7 (overlaid on the data from Figure 6). The decrease in granularity actually improved the expert-driven modifications at first, though tripling the window size produced practically unusable results. The exploration model saw increased costs and worse response time for both window sizes. As the number of states decreases, the probability of encountering a state seen previously increases, improving recall at a cost to precision. This is more beneficial to the model that is sparse to begin with; the more complete model sees a smaller improvement to recall in exchange for the same cost to precision.

V. Related Work

The autonomic adaptation of software systems is studied extensively. Approaches to run-time adaptation vary in their approach, for example from explicit models of the system's performance [9], [10], [11] like our approach, to linear programming formulations [12], [13], and to hybrid approaches [14]. Our work distinguishes itself from other approaches in several ways. A popular web services autonomic adaptation is to substitute one service for another equivalent service. Our work is service provider focused, ensuring availability and compliant performance for the services a single provider offers, ruling out substitution as a viable solution. Much work in autonomic adaptation is at the server level, monitoring load averages and memory. In contrast, our work is generally applicable to many applications, but the simulation creation and the knowledge base once created are done at the application level, monitoring application-specific metrics and capable of making changes to the application rather than to system infrastructure. We create our knowledge base entirely in simulation. We deliberately chose a model that can be visualized and used to explain why the autonomic manager made a certain decision, in contrast to black-box approaches. Finally, systematic exploration of the states of an application has not been extensively studied. The two approaches in the literature most similar to our methodology are described in more detail below.

Calinescu and Kwiatkowska [9] use a Markov model of a software application to generate an autonomic manager for that application, using the PRISM probabilistic model checker as an engine. The Markov model is created during a formal-verification process of the application (or later following the same process). Decisions are chosen by the engine based on the maximizing of a utility function. They describe empirical results for Markov chains with a small number of states, for two scenarios: energy saving by putting hard drives to sleep and cluster availability in a data center. It is not clear how accurately the deterministic model reflects reality, or how well this method translates to a real system at run time, or how this low granularity impacts their results.

Bahati *et al.* [15] use a state-transition model to encode past decisions and their results. A state consists of metrics (based on the metrics described in the policies being enforced) and the various transitions already tried when in this state. The dividing line between states is the threshold of any metric. A state-transition graph is built over time as the deployed system is manually transitioned from state-to-state. The autonomic manager identifies the state in which the system is and the transitions that could potentially move the system to an acceptable state. In comparison, our methodology builds a behavioral model offline (skipping the expensive learning stage) and offers variable granularity for each metric which allows more fine-tuned control, rather than a simple binary compliant/non-compliant discretization.

Finally, their approach does not address the problem of over-provisioned systems; our method provides a more complete solution to the problem of self-configuration.

VI. Conclusion

In this paper we extended our simulation-based method for autonomic reconfiguration of service-oriented systems. The base solution developed a state-transition model of the behavior of the system using a simulated system that is manipulated by an expert user. This model essentially captures how the system's performance is impacted by the changes in the request load that the environment imposes to the system and changes to its configuration. At run time, the system's actual behavior is tracked against this model and when the autonomic manager finds the system in a state that may lead to possible futures that violate service level agreements, its configuration is changed to move the system to a safer future state.

The primary contribution of this paper was replacing the (expensive) expert user with an automated system that recursively simulated various mutations of each state of a simulated application, creating a more complete knowledge base. This automatic creation of a decision model offers a low-cost solution for autonomic self-configuration, which in this case was able to achieve average results closer to a defined service level objective.

We further examined the process of creating the state-transition model, offering empirical results on the growth of the state-transition model over the various generations of the automatic creation. We explored trade-offs in granularity when discretizing raw performance metrics into states that meaningfully differentiate between actual application states, finding that it is possible to be too vague and to be too precise, and establishing empirically the appropriate balance for this application. Though the quality of the exploration-generated model deteriorated as granularity decreased, the low cost of CPU time versus expensive expert knowledge gives automatic exploration the advantage.

The major challenge in our method is that it relies on anticipating situations ahead of time, which implies the need for having already seen simulations of these situations. Though we have tools to automatically explore the configuration space, we intend to analytically study the coverage enabled by our method and to work on providing confidence metrics.

VII. Acknowledgments

The authors acknowledge the generous support of iCORE, NSERC, and IBM. We heartily thank Hyeon Rok Lee at the University of Alberta for his help in implementing the state detection and state searching modules.

References

[1] E. D. Nitto, C. Ghezzi, A. Metzger, M. Papazoglou, and K. Pohl, "A journey to highly dynamic, self-adaptive service-based applications," *Automated Software Engineering*, 2008.

[2] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[3] R. Murch, *Autonomic Computing*. IBM Press, 2004.

[4] M. Parashar, *Autonomic Computing: Concepts, Infrastructure, and Applications*. Bristol, USA: Taylor & Francis, Inc., 2007.

[5] M. Smit and E. Stroulia, "Autonomic configuration adaptation based on simulation-generated state-transition models," in *To Appear*, 2011.

[6] M. Smit, A. Nisbet, E. Stroulia, A. Edgar, G. Iszlai, and M. Litoiu, "Capacity planning for service-oriented architectures," in *CASCON '08*. New York, NY, USA: ACM, 2008, pp. 144–156.

[7] M. Smit, A. Nisbet, E. Stroulia, G. Iszlai, and A. Edgar, "Toward a simulation-generated knowledge base of service performance," *Proceedings of the 4th International Workshop on Middleware for Service Oriented Computing*, Nov 2009.

[8] R. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," *Artificial intelligence*, vol. 27, no. 1, pp. 97–109, 1985.

[9] R. Calinescu and M. Kwiatkowska, "Using quantitative analysis to implement autonomic IT systems," in *ICSE '09*. IEEE Computer Society, 2009, pp. 100–110.

[10] K. Verma, P. Doshi, K. Gomadam, J. Miller, and A. Sheth, "Optimal adaptation in web processes with coordination constraints," in *Web Services, 2006. ICWS '06. International Conference on*, 2006, pp. 257 –264.

[11] M. Litoiu, M. Woodside, and T. Zheng, "Hierarchical model-based autonomic control of software systems," *SIGSOFT Softw. Eng. Notes*, vol. 30, pp. 1–7, May 2005.

[12] V. Cardellini, E. Casalicchio, V. Grassi, F. Lo Presti, and R. Mirandola, "Qos-driven runtime adaptation of service oriented architectures," in *7th European software engineering conference and Foundations of software engineering*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 131–140.

[13] D. Ardagna and B. Pernici, "Adaptive service composition in flexible processes," *Software Engineering, IEEE Transactions on*, vol. 33, no. 6, pp. 369 –384, 2007.

[14] M. Litoiu, M. Mihaescu, D. Ionescu, and B. Solomon, "Scalable adaptive web services," in *Proceedings of the 2nd international workshop on Systems development in SOA environments*, ser. SDSOA '08. New York, NY, USA: ACM, 2008, pp. 47–52.

[15] R. M. Bahati, M. A. Bauer, and E. M. Vieira, "Adaptation strategies in policy-driven autonomic management," in *ICAS '07: Proceedings of the Third International Conference on Autonomic and Autonomous Systems*. Washington, DC, USA: IEEE Computer Society, 2007, p. 16.