In Proceedings of the Workshop on Self-Organizing Architecture, Cambridge, UK, September 14, 2009.

Improving Architecture-Based Self-Adaptation Using Preemption

Rahul Raheja, Shang-Wen Cheng, David Garlan, Bradley Schmerl
Carnegie Mellon University
Pittsburgh, PA - USA
rahul.raheja2009@gmail.com, {zensoul, garlan, schmerl}@cs.cmu.edu

Abstract— **One common approach to self-adaptive systems is to incorporate a control layer that monitors a system, supervisorily detects problems, and applies adaptation strategies to fix problems or improve system behavior. While such approaches have been found to be quite effective, they are typically limited to carrying out a single adaptation at a time, delaying other adaptations until the current one finishes. This in turn leads to a problem in which a time-critical adaptation may have to wait for an existing long-running adaptation to complete, thereby missing a window of opportunity for that adaptation. In this paper we improve on existing practice through an approach in which adaptations can be preempted to allow for other time-critical adaptations to be scheduled. Scheduling is based on an algorithm that maximizes time-related utility for a set of concurrently executing adaptations.**

self-adaptation; preemption; utility; concurrency

I. INTRODUCTION

Today's complex systems require considerable administrative overhead; this has led to a demand that they self-adapt at run-time to variable resource availability, loads and faults. Until recently, mechanisms for self-adaptation were largely in the form of programming language features, embedded in the code, hence prohibiting reusability and modifiability. Today there is an increasing trend toward *autonomic computing*, in which external control modules are used to provide adaptive capabilities that monitor and adapt the system at run time. Rainbow [1] is a framework for external control that provides capabilities for self-adaptation and provides mechanisms to balance adaptations amongst multiple stakeholder objectives [2]. It forms a closed-loop control system in which the adaptation mechanism probes, evaluates, decides, executes, and then probes again. In this respect, Rainbow is similar to other autonomic systems [3,4,5].

While such approaches have been demonstrated to be useful, they are largely limited to carrying out a single adaptation at a time; violations that occur while an adaptation is taking place will not trigger new adaptations until the current one has finished executing. Unfortunately, ignoring or delaying adaptations potentially leads to less-than-optimal adaptation, since a high-priority adaptation may have to wait for a less critical adaptation to finish. For example, it may not be desirable to delay addressing security issues until a currently executing performance adaptation completes. By not addressing a security problem early, the system may be compromised, so that later the security adaptation will not be effective, or will be more disruptive to the system.

Ideally, whenever new adaptation conditions arise, the adaptive mechanism should be able to reconsider all objectives and determine the best course of adaptation without waiting for an existing adaptation to complete. In this paper we propose an improvement upon existing approaches that avoids delaying adaptation decisions, where the adaptation mechanism promptly considers all adaptations, including the currently executing adaptation, when new conditions arise. To make this possible, we extend Rainbow's adaptation mechanisms to support preemption of an executing adaptation strategy, reasoning amongst multiple strategies, starting new ones, and resuming preempted strategies. To facilitate this, we propose an adaptation time utility dimension for self-adaptive mechanisms to reason about and to prioritize amongst multiple objectives and their corresponding adaptations at runtime. The specific contributions of this paper are:

- Applying the concept of time-utility curve (TUC) to prioritize amongst multiple adaptations using adaptation times[1] (see III.b)
- The application of existing concepts of multi-task scheduling under the constraint of maximizing overall utility to improve current adaptation decision-making algorithm (see III.c and III.d)
- The use of an approximation of the rely-guarantee technique to ensure consistency amongst multiple interleaving adaptation strategies (see III.f)
- The use of an architecture model of a system as a reference to provide architectural locks to ensure non-interference (see III.f)

The remainder of this paper is organized as follows: In Section 2, we give a brief overview of the existing Rainbow adaptation mechanism components and outline a motivating scenario. In Section 3 we present our design methodology and implementation. In Section 4 we give some experimental results. In Section 5 we give related work, and sections 6 and 7 give discussion points and future work followed by conclusions in Sections 8.

II. MOTIVATION

The Rainbow framework uses software architectures and a reusable infrastructure to support self-adaptation of software systems. Figure 1 illustrates its adaptation control loop. *Probes* are used to extract information from the target system

[1] The time from when a constraint violation is triggered until the corresponding adaptation strategy is executed

that updates the model via a set of gauges. The *architecture evaluator* checks constraints in the model and triggers adaptation if any violation is found. The *adaptation manager*, on receiving the adaptation trigger, chooses the best strategy to execute, and passes it to the *strategy executor*, which executes the strategy on the target system via *effectors*. The strategy to be executed is chosen on the basis of stakeholder utility preferences and the current state of the system as reflected in the architecture model. The underlying decision making model is based on decision theory and utility theory [6]. Each strategy, which is written using the Stitch adaptation language, is a multi-step pattern of adaptations in which each step evaluates a set of condition-action pairs and executes an action, called a tactic, on the target system. A tactic defines an action, packaged as a sequence of primitive commands, called operators. It also specifies conditions of applicability, expected effects, and cost-benefit attributes to define its expected impact on the quality dimensions.

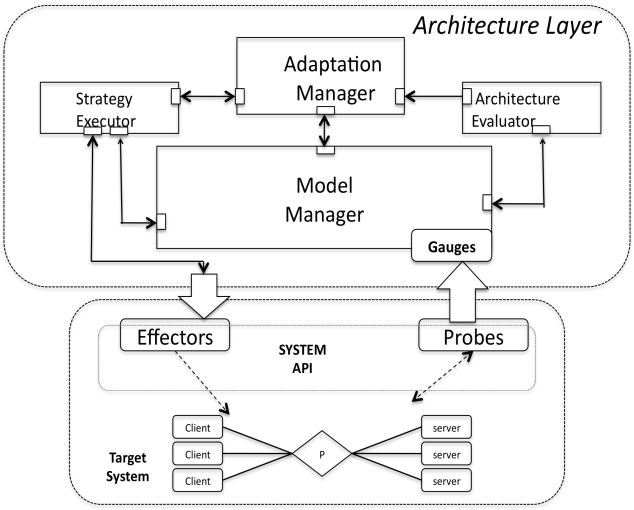

Figure 1. Rainbow adaptation control loop components

To illustrate, consider a web-based client-server system that conforms to an N-tier style (as illustrated in Figure 1 target system). It uses a load balancer to balance requests across a pool of replicated servers, the size of which is dynamically adjusted to balance server utilization, response time, cost, etc. Assume Rainbow can monitor the system for information such as server load, port activity on servers, etc. Assume also that we can modify the system to add more servers to the pool or to change the quality of the content and to shut down ports. Suppose Rainbow detects a violation of some servers' response time and considers adding servers to the server group. While it is in the middle of carrying out this adaptation, probes report a port-scan activity on one of the servers and this triggers a security-related adaptation. In the current implementation of Rainbow, the new trigger will be ignored since Rainbow is in the middle of adding servers to the server group. If the port activity were in fact an intrusion and not just a scan, and the port was not closed (because the trigger was ignored), the server will be compromised, reducing overall system utility.

Rainbow's mechanisms were deliberately built to delay adaptations in such circumstances. Rainbow considered that a new adaptation trigger in the middle of an ongoing adaptation could potentially be caused by an intermediate action of the current adaptation on the target system, and that finishing the current adaptation could potentially eliminate the need for a new one. If the condition persists, a repair will eventually be triggered. Although this might be reasonable in some cases, in others, like the one mentioned above, this would reduce overall system utility. We aim to extend Rainbow's mechanisms to be able to manage both scenarios.

III. APPROACH

We apply the concepts of preemption and concurrency in adaptation mechanisms to address the above situation. When an adaptation condition is triggered during an ongoing adaptation, Rainbow will consider the new condition along with the condition that caused the currently executing adaptation strategy. If addressing the new condition first yields higher utility, Rainbow will preempt the running strategy and schedule a new one, provided there are no conflicts between the two strategies. While promising, the introduction of preemption and concurrency gives rise to a number of issues. In the following sections, we present the design of incorporating preemption into Rainbow.

A. Incorporating changes in Rainbow

While introducing preemption mechanisms and dealing with its related issues, we want to extend Rainbow to provide a generic framework with which different Rainbow users can customize the prioritization knowledge, scheduling and conflict detection mechanisms according to their own requirements. To incorporate preemption, we add an *RTEvaluator* (Real Time Evaluator: Figure 2) sub-component into the Adaptation Manager component. This component is responsible for ordering a set of strategies that could be executed according to customizable criteria and algorithms.

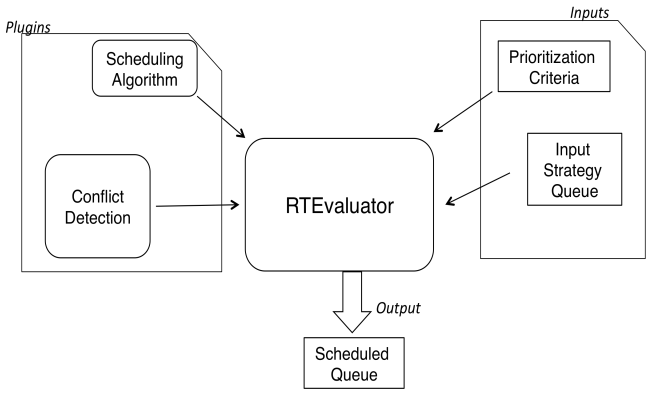

Figure 2. Detailed design of the RTEvaluator sub-component

The RTEvaluator expects two inputs and two plugins. The inputs are a queue of strategies that need to be ordered for scheduling and the prioritization criteria. The plugins are the scheduling algorithm and the conflict detection mechanism. The evaluator, on receiving a set of strategies, uses the prioritization criteria and the scheduling algorithm, in consultation with the conflict detection module (described later), and outputs an execution order that satisfies the conflict rules and the prioritization criteria. The interactions of the RTEvaluator with other sub-components are shown in Figure 3.

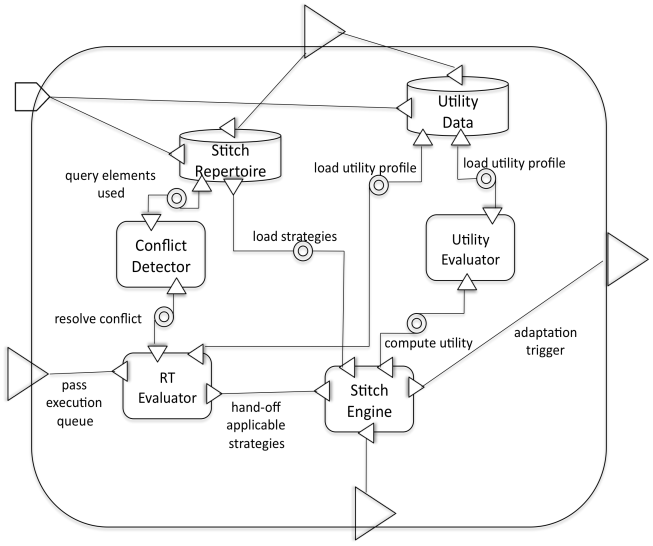

Figure 3. Interactions of the RTEvalutor sub-component with other sub-components in the Adaptation Manager

B. Prioritizing Adaptations with TUC

When an adaptation trigger occurs during an ongoing strategy execution, the Adaptation Manager must decide whether to finish executing the current strategy before servicing the newly triggered adaptation, or preempt it and service the new condition first. Accordingly, we need to be able to reason about which strategy or adaptation condition is more important given the current state of the system and user priorities. One way to prioritize strategy selection is for the architect of the system to specify the priorities of adaptation conditions at design time. In such a case, whenever a higher priority adaptation condition occurs, other lower priority adaptations will be aborted or preempted. This would imply that if a lower priority adaptation condition occurs, then it will always wait for the current adaptation to finish, and, if a higher priority adaptation occurs, then it will always be serviced first.

As mentioned earlier, adaptations in self-adaptive systems are opportunities for improvement. They are aimed to increase overall system utility, which quantifies the happiness of the system stakeholders with the system. If strategies take a long time to execute over the target system, their intended utility improvement will diminish or disappear. In particular, given the static priority scheduling proposed above, a lower-priority condition strategy would *always* wait until higher-priority strategies finish executing, hence potentially losing its window of opportunity.

To avoid the above situation, we must introduce some criteria into the Adaptation Manager that will allow it to make a decision at run-time about how to prioritize strategies. To do this, we associate each strategy with a time utility curve (TUC) [11]. A TUC specifies the expected utility [0,1] of completing a strategy as a function of its completion time (its use is explained further in section III.d). Using TUCs, we can express both hard real-time adaptations and soft real-time adaptations. It also gives one the ability to specify arbitrary utility values rather than step/linear functions (Figure 4). We can then use this information to schedule strategies according to some scheduling criteria.

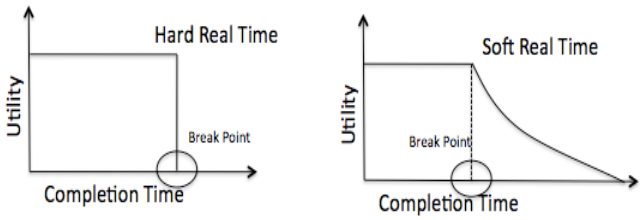

Figure 4. Time Utility Curves for Hard/Soft Real time adaptations

C. Improvements to Strategy Selection Algorithm

The scheduling algorithm is encapsulated as a plug-in to the RTEvaluator, which different users can customize. Our algorithm aims to maximize end system utility. When the *evaluator* triggers an adaptation, the set of applicable strategies are handed to the *RTEvaluator* by the *Stitch Engine*[2] which schedules as follows:

a. Let Q be the ordered queue of strategies pending execution; if non-empty, the head entry $Q[0]$ is the currently executing strategy, so preempt it
b. Let P be the unordered set of Q
c. Let A be the set of applicable strategies for the new condition
d. Create a new empty set C of candidate strategy orderings
e. For each strategy a in A
    a. Pick the ordering of $a \cup P$ that has the highest Predicted System Utility (PSU) (explained in the next section)
    b. Add this ordering and PSU to C
f. Pick the ordering with the highest PSU from C to form Q', the new ordered queue of strategies pending execution
g. Execute (or resume) the strategy at $Q'[0]$

When the Stitch Engine detects constraint violations for the first time and selects a set of applicable strategies, before handing over the applicable set to the RTEvaluator, it saves the set of constraint violations that caused the adaptation

[2] Adaptation Manager sub-component that receives adaptation trigger from the evaluator

trigger. When a strategy is selected, the constraint set is associated with this strategy. In the next cycle, if the Stitch Engine finds constraint violations that already have some strategy(s) associated with them, it ends the evaluation cycle and waits for that strategy(s) to finish. When the strategy finishes execution, it instructs the Stitch Engine to remove its associated constraints so they can be considered in the next cycle if required. The rationale is not to service a set of constraint violations again if a strategy has already been chosen to improve or undo them.

D. Predicted System Utility

The Predicted System Utility (PSU) of a set of strategies gives the final utility of the system assuming the strategies are executed in a particular order. Suppose the current system state is represented by S[], which is a vector of all utility dimensions of the system. Say,

$$S[] = [\text{fidelity}, \text{cost}, \text{responseTime}, \text{security}] = [p, q, r, t]$$

We have a set of strategies A_1, A_2 … A_N, that will be executed in the order [A_1, A_2 … A_N]. Each strategy is associated with an aggregate Attribute vector[3] $agg[]_{Ai}$, and a time utility curve TUC_{Ai}

$$agg[]_{Ai} = [\Delta p, \Delta q, \Delta r, \Delta t]$$

Running A_1 first leaves the system in state $S[]_1$

$$S[]_1 = S[]_0 + TUC_{A1}(\text{exec time of } A_1)*agg[]_{A1}$$

$TUC_{A1}(\text{exec time of } A_1)$ will give the percentage of expected change that actually happened depending on the execution time of A. Similarly running A_2, A_3 … A_N gives

$$S[]_2 = S[]_1 + TUC_{A2}(\text{exec time of } A_1+A_2)*agg[]_{A2}$$

$$S[]_N = S[]_{N-1} + TUC_{AN}(\text{exec time of } A_1+A_{2\ldots}A_N)*agg[]_{AN}$$

The state of the system at the end of executing A_1, A_2 … A_N will be

$$S[]_N = [p_N , q_N , r_N , t_N]$$

We calculate utility using utility profiles and weights for the quality dimensions

$$\text{Weight}_{p*}\text{UtilityCurve}_p(p_N) + \text{Weight}_{q*}\text{UtilityCurve}_q(q_N) + \text{Weight}_{r*}\text{UtilityCurve}_r(r_N) + \text{Weight}_{t*}\text{UtilityCurve}_t(t_N) = \text{PredictedUtility}$$

This utility is the predicted system utility for the specified execution order. The strategy execution times are calculated by profiling under serial mode (i.e., with no preemption). Rainbow is run under different configurations and the mean, standard deviation and error are calculated using multiple runs for each tactic. The execution time of the strategy is then estimated probabilistically from its tactic tree, which is then fed into the calculation of the predicted system utility. Also, to calculate the execution time of a preempted strategy, the remaining execution time is calculated from the tactic where it was preempted and not from the beginning.

[3] Aggregate attributed vector of a strategy is the expected change to utility dimensions it expects to provide after it finishes executing

E. Granularity of Concurrency

The execution of a running strategy must be interrupted to serve a higher priority adaptation. The granularity at which a strategy can be preempted, must strike a balance between achieving maximum possible interleaving and ensuring that preemption leaves neither the target system, nor the model on which decisions are made, in an inconsistent state. In Rainbow, there are three potential levels at which this basic execution unit can be set.

First, we can set the basic execution unit at the strategy level, which is the most coarse-grained. This is equivalent to a policy in which a strategy cannot be preempted. Thus, if a new adaptation condition arises, it will always wait for the current strategy to finish. In this case, as mentioned before, the new strategy might lose its window of opportunity. The second possible granularity unit would be at the tactic level; a third at the operator level. If we choose an operator as the granularity unit, we will need to have preconditions added before every operator is executed, since there is currently no applicability condition check for operators. Also, specification of timing at every operator level would be unwieldy, as this would require a 2-way communication between the Rainbow adaptation and target layer involving overheads. If we choose tactic as the unit, it will not provide as fine grain granularity as an operator, and hence not as high end system utility. But Rainbow's underlying assumption is that a tactic leaves the system in a consistent state, at least at the architectural level. Furthermore, tactics already have condition guards. For these reasons, for our work, we chose a tactic over operator to be the atomic unit of execution that cannot be pre-empted.

F. Conflict Detection and Resolution using Architectural Locks

Consider a strategy A that is pre-empted to execute another strategy B. Suppose that strategy A was pre-empted after executing 2 tactics, and is yet to execute 2 more. While strategy B executes it makes some changes to parts of the target system. Now when strategy A resumes and starts executing the remaining tactics, it assumes its target components to be in a state they were in when the first two tactics finished. If B undid those changes, then finishing A will leave the target system in a potentially undesirable or inconsistent state.

Following the above example, we need to ensure that actions of interleaving strategies do not conflict with each other. There are a couple of ways to ensure this. First, we can leave the onus on the strategy writers to ensure that their strategies are written in ways that do not conflict with any other strategies written for this system. But there could be many strategies for a system, potentially written by different people; expecting a strategy writer to know the behavior of all other strategies would be unreasonable. Even if one assumes that the strategy writer knows about all other strategies, it could still be unreasonable for him to frame the strategies in a way that doesn't conflict with others, because pairwise conflicts would again mean considering too many cases.

Second, we can provide some guarantees to a strategy that other strategies will not touch the parts of the system that it acts on using an approximation of rely and guarantee reasoning [17]. In this, each strategy *guarantees* that it will make changes to only a subset of the system and that it *relies* on the fact that no other strategy will makes change to this subset while it is executing. This would be like a strategy requesting a set of components and connectors that it would make changes to, effectively putting a lock over them, and no other strategy would make changes to that set. We propose to use the architecture model of the target system and its environment as a reference to provide virtual locks to strategies. The environment of a system consists of the components and connectors that are currently unused and can be added to the system. For example, unused servers that can be added to a system under high load are a part of the system's environment. Each strategy, when queried at runtime, will return a set of components and connectors corresponding to the architectural model of the system and its environment that it would act upon if executed at that moment. We can use this information to ensure that a set of strategies do not interfere with each other.

We encapsulate the conflict detection part in the Conflict Detector subcomponent in the Adaptation Manager by providing four interfaces to it. First is an interface with which a strategy that is about to be executed will register its components, and second is to deregister (mark as free by removing a virtual lock). When a new strategy is to be run, it seeks permission from the Conflict Detector using the third interface. Using the fourth interface that we provide, if one gives a set of strategies to the Detector, it will return a schedule, or a subset that will be a non-conflicting schedule. We leave it to the convenience of the Rainbow users to decide which best fits their requirements in terms of the interface they want to use, as well as the algorithms in the Conflict Detector. For our implementation, if a strategy about to run uses a component(s) being used by any pre-empted strategy, it is not allowed to execute, and a new order is found that is non-conflicting and gives the next highest utility.

IV. RESULTS

A. Target System

We use a typical news website infrastructure, Znn.com, which is typically a three-tiered architecture, to demonstrate the preemption scenario. In this model, a set of application servers serve content from the backend databases to clients via the presentation logic. Similar setup was used to demonstrate Rainbow's results [18] during its stages of development (refer Figure 1 target system). The servers can be adjusted to balance utilization for response time. The set of clients makes requests for content that includes text, images, videos (static content) and templates (dynamic content).

Common objectives for a news provider are to deliver server content within a reasonable response time, while keeping the budget under control. To avoid dropping requests in high load time, the fidelity of the content is adjusted, which decreases response time, hence serving more requests in a time frame. The providers also want to ensure that all their servers are secure and no malicious activity risks the server group. In short, we short-list four quality objectives for Znn.com – cost, performance, fidelity of content and security. The number of servers in the backend group directly impacts cost analysis and hence we take the server count into consideration. For content quality, we define three levels based on type of content served – high (text, video and images), medium (text and images) and low (only text). Performance analysis comes from considering response time, bandwidth and server load. For security analysis, we associate a security Confidence Level with each server, which indicates the trust the system has regarding its security (highest value being 1.0). In case of any suspicious activity (e.g., on a port) this confidence level would be lowered causing adaptation to be triggered.

The server activate and deactivate operators are defined to add or remove server(s) from the server group, while the setFidelity operator changes fidelity of the content to be served at the specified level and the closePort operator closes the specified port. From these operators, we specify 2 pairs of tactic. One pair discharges (or enlists) a server(s) and the other shuts down the port on a ServerT type element.

B. Competing Scenario

The scenario that we chose to demonstrate is one in which cost adaptation competes with security adaptation. The Rainbow framework detects the cost of the system is too high. It triggers an adaptation that leads to a strategy being selected for execution – ReduceOverallCost. This will aim to reduce overall cost by discharging a few servers. While this adaptation is in progress, a possible security breach is reported on one of the servers, and is picked up by the Evaluator. This causes the secureServer strategy to be selected. At this point of time, Rainbow will have to make a decision.

If Rainbow is running in the serial mode (which is how the current mechanisms exists), it will delay the new adaptation. It will continue to execute the cost adaptation, and when finished it will execute the security adaptation. By this time, the server could have been potentially compromised. If the cost strategy was preempted, and security adaptation scheduled, this could have prevented the server from being further compromised. This deadline time is reflected in the time utility curves of both the strategies that are used to schedule them as explained previously (Section 3.2, 3.3).

C. System Utility

We execute the scenario mentioned above in both modes, multi and serial, for multiple runs and record the instantaneous utility at regular intervals. We use these values to plot the utility graphs. Figure 5 displays the instantaneous system utility over the entire execution trace for both modes; Figure 6 plots the accrued utility[4] for both runs. From the

[4] Accrued utility at any instance is the sum of instantaneous utilities of all previous instants in the execution trace

instantaneous utility graph, we see that in serial mode, because of the security strategy missing its deadline, the stable system utility drops down below to 0.854. When the server gets compromised, its response time and its service time go high, causing the system utility to drop to as low as almost 0.65. Seeing such a state of the system, adaptation layer selects new strategies in an effort for improvement. This effort ends up being a transient effort to compromise between cost and response time, where trying to improve one violates the other. This carries on until a compromise is made and no other strategies are available for further improvement. At this point the instantaneous utility stabilizes to being around 0.854. For the multi mode, the utility is not harmed since the cost adaptation was pre-empted and the security adaptation was carried out first. This stabilizes around 0.9725. The % difference in stable instantaneous utility is almost 13.9%.

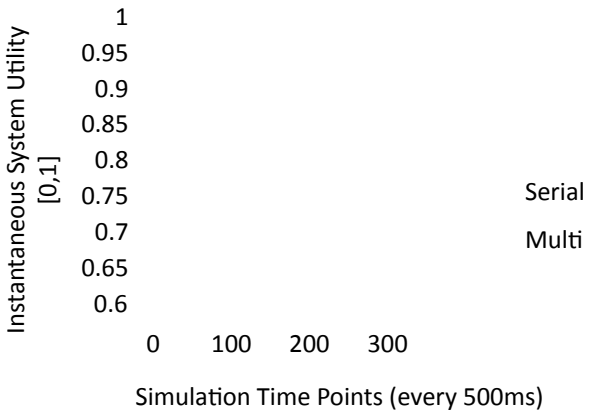

Figure 5. Instantaneous utility

From the accrued utility graph, we get a measure of how well the system has been performing since it was brought online, being monitored by Rainbow.

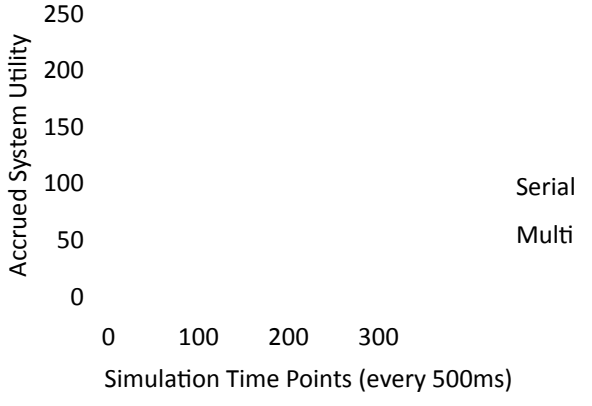

Figure 6. Accrued utility

Results show that in serial mode, the accrued utility at the end of the execution trace is almost 194 and for the multi mode its approximately 234. This is net 20% increase by adding preemption mechanisms to Rainbow, clearly indicating the need for such mechanisms in the self-adaptation domain.

V. Related Work

The technique of using rely-guarantee to ensure consistency is a studied concept. It has been used to ensure non-conflicting interactions for implementing linked lists fine-grain interactions [7]. It has been used for constructing a reasoning style system for the aspect-oriented programming paradigm [8]. Modifications for ease of use have been proposed as local rely and guarantee [9], and it has been combined with other concepts such as separation logic to give stronger consistency guarantees [10]. Although we have used a weak approximation of this concept, a stronger notion can be embedded into the Conflict Detector sub-component in the rainbow's Adaptation Manager.

The concept of time utility curves [11] has been explored with respect to time-critical resource management [12]. They have been used in scheduling algorithms such as GUS [13] [14]. GUS was one of the earliest proposals for scheduling under real-time and mutual exclusion constraints in the operating system domain. This algorithm, similar to Predicted System Utility as proposed in this paper, aimed to maximize accrued system utility. It produced sub-schedules that were mutually exclusive and gave maximum system accrued utility. In a similar fashion, Rainbow's RT Evaluator and Conflict Detector would interact to produce schedules and sub-schedules to ensure maximum predicted utility and adherence to rely-guarantee.

The pre-emption aspect of this work could be viewed as a form of conflict resolution in strategies. While prior work [19] provided integrated support for resolving conflicts in adaptation strategies, it did not consider the case of making an adaptation decision when an existing action is already in progress. Our work explicitly tackles this problem, which combines both scheduling and resource conflicts.

VI. Discussion

When an adaptation requirement is triggered during an ongoing strategy execution, the Adaptation Manager must decide whether to finish executing the current strategy before servicing the newly triggered adaptation, or preempt it and service the new condition first. The algorithm we proposed in Section 3.3 preempts the currently executing strategy and then decides the new execution order. One possible alternative would be to preempt the currently executing strategy only if the RTEvaluator chooses an order where the currently executing strategy is not first. The problem with this approach arises when the executing strategy is in the middle of a long tactic execution and cannot be preempted immediately. This loss of precious time for the new strategy would lead to a reduction in overall utility. On the contrary, had we indicated the request to preempt and then started

scheduling, the strategy could have possibly been preempted (or about to be finishing the long tactic) by the time when the execution order was finalized and the new one could have been scheduled almost instantly; hence our decision to indicate preemption immediately.

In using external control modules for self-adaptation, the time when problems occur in the target system and the time when they are actually reported/detected in the Rainbow layer will be different (see Figure 7). There will always be some time lag. Our current implementation of preemption mechanisms does not include this time lag. Some approximations can be used to negate this lag, for example, adding the execution cycle time to the constraint detection time, etc. But as of now there is no clear solution of how to negate or minimize this lag.

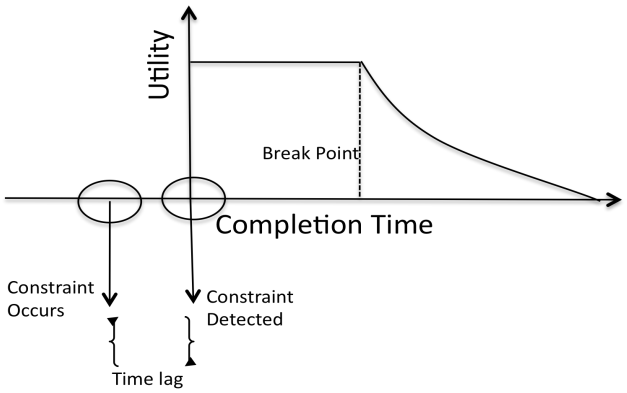

Figure 7. Potential time lag in actual constraint violation and detection

In Rainbow, there are potentially four places where we could associate TUC. Firstly, with each architectural property in the model; it could be possible that one architectural property is associated with multiple constraints, and each constraint associated with multiple strategies. So, it would be of lesser relevance associating TUC with architectural properties. This also rules out association of TUC with constraint violations. Thirdly, we can associate it with each quality dimension; the stakeholders of the system specify the dimensions that are most important to the business, but asking them to specify adaptation responsiveness for their objectives may be unreasonable. Lastly, with each strategy we can associate a different TUC. This is reflected in our implementation, the rationale being that domain experts writing strategies are in the best position to state the responsiveness expected.

The scheduling algorithm suggested in sections III.c and III.d, which is used as a plugin to the RTEvaluator, involves factorial times computation of the number of applicable strategies. In large-scale deployments where potentially hundreds of strategies would be involved, the algorithmic computation would become an overhead. Using well explored techniques of caching (of intermediate results) and parallel processing would reduce the overhead, but in cases where this is not permissible, Rainbow users can provide a different algorithm in the plugin. For demonstration under our setup, the computational time was negligible.

VII. Future Work

From here onwards, we would like to see our work to be the background of a potentially fully concurrent adaptive system. Rather than having just an interleaving of strategies, we have different threads of execution for each strategy leading to true concurrency. This would depend not only on better and improved conflict detection mechanisms, but the effectors interface in the translation layer, supporting multiple instances.

Also, we would like to see more work being done towards the run time updatability test. Software systems consist of multiple nodes and many of these systems are distributed transactional systems. Transactions are sequences of steps that need to be carried out as one; if one of them fails, the entire transaction fails. In a distributed context, a node could depend on the services provided by other nodes to provide its own functionality to process a transaction step. Any changes to a node that is servicing or about to service as a part of a transaction could potentially lead to a transaction fail, and performing a rollback could be very costly. As identified by Kramer and Magee: the system must be in a consistent state before and after runtime changes [15]. Rainbow uses strategies to implement adaptation, where each strategy is a sequence of tactics. It assumes that each tactic leaves the system in a consistent state. The question that Rainbow does not address is "Can the tactic be executed now?" This means Rainbow needs to address the topic of whether a target node is in a quiescent state or undergoing update. Future work needs to be done to ensure that the target nodes are in an updatable state. We propose that one of the ways to ensure run-time updatability is to use the notion of tranquility [16]. We can use the probes in the Rainbow translation infrastructure to gather this information and make tranquility as a property in the architecture model. The strategies can check whether the target node is in a tranquil state or not as a condition of applicability in the strategy, or enforce the node to be tranquil via a tactic.

VIII. Conclusion

Adding preemption mechanisms in self–adaptation and allowing multiple adaptations to be considered for scheduling gives time critical adaptations the opportunity to be scheduled promptly and hence increase overall system utility. We also proposed a framework with which different users can customize different scheduling methodologies, conflict detection mechanisms and prioritization criteria depending on their requirements, thus making this approach flexible and promising.

Acknowledgment

This work is supported in part by the Office of Naval Research (ONR), United States Navy, N000140811223 as

part of the HCSB project under OSD, the National Science Foundation (NSF) under grant CNS-0615305, and by the US Army Research Office (ARO) under grant number DAAD19-02-1-0389 ("Perpetually Available and Secure Information Systems") to Carnegie Mellon University's CyLab.

REFERENCES

[1] D. Garlan, S. Wen-Cheng, A. Cheng-Huang, B. Schmerl and P. Steenkiste, "Rainbow: Architecture-Based Self Adaptation with Reusable Infrastructure", in *IEEE Computer,* vol. 37(10), October 2004, pages 46- 54.

[2] S. Wen-Cheng, D. Garlan and B. Schmerl, "Architecture-based Self-adaptation in the Presence of Multiple Objectives*",* in *ICSE 2006 Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, Shanghai, China, 21-22 May 2006, pages 2 - 8.

[3] A. Sztajnberg and O. Loques, "Describing and deploying self-adaptive applications", in *Proc. 1st Latin American Autonomic Computing Symposium,* July 14–20, 2006. 2.3.3.

[4] T. Vasconcelos Batista, A. Joolia, and G. Coulson, "Managing dynamic reconfiguration in component-based systems", in *EWSA,* volume 3527 of LNCS, pages 1–17. Springer, June 13–14, 2005. 2.3.3.

[5] E. M. Dashofy, A. Van der Hoek, and R. N. Taylor, "Towards Architecture-based Self-healing systems", in *Proceedings of the First Workshop on Self-healing Systems*, 2002, pages 21-26.

[6] J. A. Bather, Decision Theory: An Introduction to Dynamic Programming and Sequential Decisions, John Wiley and Sons, July 13, 2000. 1.3.1, 2.2.

[7] V. Vafeiadis, M. Herlihy, T. Hoare, M. Shapiro, "Proving Correctness of Highly-Concurrent Linearisable Objects", *Proceedings of the eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* New York, 2006, pages: 129 – 136.

[8] R. Khatchadourian, J. Dovland ,N. Soundarajan, "Enforcing Behavioral Constraints in Evolving Aspect-Oriented Programs", in *Proceedings of the 7th Workshop on Foundations of Aspect-oriented Languages,* Brussels, Belgium, 2008, pages 19-28.

[9] X. Feng, "Local Rely-Guarantee Reasoning", in *Proc. 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'09),* Savannah, Georgia, USA, pages 315-327, January, 2009.

[10] V. Vafeiadis, M. Parkinson, "A Marriage of Rely/Guarantee and Separation Logic", in *18th International Conference on Concurrency Theory (CONCUR)*, vol 4703 of Lecture Notes in Computer Science. Springer, Lisbon, Portugal September 2007, cited on pages 12 and 45.

[11] E. D. Jensen, C. D Locke and H. Tokuda, "A time-driven scheduling model for real-time systems", in *IEEE RTSS,* pages 112-122, December 1985.

[12] P. Li, B. Ravindran, E. Douglas Jensen, "Adaptive Time-Critical Resource management Using Time-Utility Functions: Past, Present and Future", in *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International,* Hong Kong, vol 2, 28-30 Sept. 2004, page 12 – 13.

[13] P. Li, H. Sang Wu, B. Ravindran and E. Douglas Jensen, "A Utility Accrual Scheduling Algorithm for Real-Time Activities with Mutual Exclusion Resource Constraints", *IEEE Transactions on Computers,* vol. 55, No. 4, April 2006, pages 454- 469.

[14] K. Chen and P. Muhlethaler, "A Task Scheduling algorithm for tasks described by time value function", *Real-Time Systems*, vol 10, pages 293-312 (1996).

[15] J. Kramer, J. Magee, "The evolving philosophers problem: dynamic change management", *Software Engineering, IEEE Transactions,* vol 16, Issue 11, Nov 1990 pages: 1293 – 1306.

[16] Y. Vandewoude, Y. Berbes, P. Ebraert, T. D'Hondt, "An Alternative to Quiescence: Tranquility", *Software Maintenance, 2006. ICSM '06. 22nd IEEE International Conference*, Philadelphia, USA, pages 73-82.

[17] C.B. Jones, "Tentative steps toward a development method for interfering programs", *In Transactions on Programming Languages and Systems 1983*, vol. 5 number 4, pages 569-619

[18] S. Wen-Cheng, Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation, Submitted in partial fulfillments for degree of doctor of philosophy.

[19] A.-C.Huang and P.Steenkiste, "Bulding Self-adaptation services using service-specific knowledge", in *Proceedings of IEEE High Performance Distributed Computing (HPDC)*, Research Triangle Park, NC, USA, July 2005, pages 34-43.