A New Architecture for Trustworthy Autonomic Systems

Thaddeus O. Eze, Richard J. Anthony, Chris Walshaw and Alan Soper
Autonomic Computing Research Group
School of Computing & Mathematical Sciences (CMS)
University of Greenwich, London, United Kingdom
{T.O.Eze, R.J.Anthony, C.Walshaw and A.J.Soper}@gre.ac.uk

***Abstract* — This paper presents work towards a new architecture for trustworthy autonomic systems (different from the traditional autonomic computing architecture) that includes mechanisms and instrumentation to explicitly support run-time self-validation and trustworthiness. The state of practice does not lend itself robustly enough to support trustworthiness and system dependability. For example, despite validating system's decisions within a logical boundary set for the system, there's the possibility of overall erratic behaviour or inconsistency in the system. So a more thorough and holistic approach, with a higher level of check, is required to convincingly address the dependability and trustworthy concerns. Validation alone does not always guarantee trustworthiness as each individual decision could be correct (validated) but overall system may not be consistent or dependable. A new approach is required in which, validation and trustworthiness are designed in and integral at the architectural level, and not treated as add-ons as they cannot be reliably retro-fitted to systems. In this paper we analyse current state of practice in autonomic architecture and propose a different architectural approach for trustworthy autonomic systems. To demonstrate the feasibility and practicability of our approach, a case example scenario is examined. The example is a deployment of the architecture to an envisioned Autonomic Marketing System that has many dimensions of freedom and which is sensitive to a number of contextual volatility.**

Keywords - trustworthy architecture; trustability; validation; autonomic marketing; autonomic system; dependability

I. INTRODUCTION

The autonomic architecture as originally presented in the autonomic computing blueprint [1] has been widely accepted and deployed across an ever-widening spectrum of autonomic system (AS) design and implementations. Research results in the autonomic research community are based, predominantly, on the architecture's basic MAPE (monitor-analyse-plan-execute) control loop, e.g., [13][14][15][16]. Although several implementation variations of this control loop have been promoted, alternative approaches (e.g., [17]) have also been proposed. In [17], Shuaib *et al.* presented an 'alternative' autonomic architecture based on Intelligent Machine Design (IMD), which draws from the human autonomic nervous system. However, research [11] shows that most approaches are MAPE [2] based. Despite progress made, the traditional autonomic architecture and its variations is not sophisticated enough to produce trustworthy ASs. A new approach with inbuilt mechanisms and instrumentation to support trustworthiness is required.

At the core of system trustworthiness is validation and this has to satisfy run-time requirements. In large systems with very wide behavioural space and many dimensions of freedom, it is close to impossible to comprehensively predict possible outcomes at design time. So it becomes highly complex to make sure or determine whether the autonomic manager's (AM's) decision(s) are in the overall interest and good of the system. There is a vital need, then, to dynamically validate the run-time decisions of the AM to avoid the system 'shooting itself on the foot' through control brevity. The traditional autonomic architecture does not explicitly and integrally support run-time self-validation; a common practice is to treat validation and reliability as add-ons. Identifying such challenges, the traditional architecture has been extended (e.g., in [3]) to accommodate validation. Diniz *et al.* [3] extended the MAPE control loop to include a new function called *test*. By this it defines a new control loop comprising Monitor, Analyse, Decision, Test and Execute –MADTE activities. The main point here is that a *self-test* activity is integrated into the autonomic architecture to provide a run-time validation of AM decision-making processes. But the question is can validation alone guarantee trustworthiness.

The peculiarity of context dynamism in autonomic computing places unique and complex challenges on trustworthy ASs that validation alone cannot sufficiently address. Take for instance; if a manager (AM) erratically changes its mind, it ends up introducing noise to the system rather than smoothening the system. In that instance, a typical validation check will pass each correct decision (following a particular logic or rule) but this could lead to oscillation in the system resulting in instability and inconsistent output. A typical example could be an AM that follows a set of rules to decide when to move a server to or from a pool of servers. As long as the conditions of the rules are met, the AM will move servers around not minding the frequency of changes in the conditions. An erratic change of mind (high rate of moving servers around) will cause undesirable oscillations that ultimately detriment the system. What is required is a kind of intuition that enables the manager to carry out a change only when it is safe and efficient to do so – within a particular safety margin. A higher level of self-monitoring to achieve, e.g., stability over longer time frames, is absent in the MAPE-oriented architectures. This is why ASs need a different approach. The ultimate goal is not just to achieve self-management but to achieve consistency and reliability of results through self-management. These are the core values of the proposed architecture.

We look at the current state of practice in the work towards AS trustworthy architecture in Section II. We propose an AS trustworthy architecture in Section III and present a case example in Section IV. Section V concludes the paper.

II. CURRENT STATE OF PRACTICE TOWARDS TRUSTWORTHY ARCHITECTURE

In this section, we look at the current state of practice and efforts directed towards AS trustworthiness. We analyse few proposed trustworthy architectures and some isolated bits of work that could contribute to trustworthy autonomic computing. Trustworthiness requires a holistic approach, i.e., a long-term focus as against the near-term needs that merely address methods for building trust into existing systems. This means that trustworthiness needs to be designed into systems as integral properties.

A trustworthy autonomic grid computing architecture is presented in [4]. This is to be enabled through a proposed fifth self-* functionality, *self-regulation*. Self-regulating capability is able to derive policies from high-level policies and requirements at run-time to regulate self-managing behaviours. One concern here is that proposing a fifth autonomic functionality to regulate the other functionalities as a solution to AS trustworthiness assumes that trustworthiness can be achieved when all four functionalities perform 'optimally'. The four self-* functionalities alone do not ensure trustworthiness in ASs. For example, the self-* functionalities do not address *validation* which is a key factor in AS trustworthiness. Amongst effort focused on validation include [3][5][6]. As explained earlier, Diniz *et al.* [3] has extended the MAPE-based autonomic architecture to incorporate a self-test activity to guarantee run-time validation of AM decisions. This is a huge step towards AS trustworthiness. The approach in [5][6] is another extension of the MAPE-based structure to include self-testing as an integral and implicit part of the AS. The same model for AS management using autonomic managers (AMs) is replicated for the self-testing. In the self-test structure, test managers (TMs) (which extend the concept of AMs to testing activities) implement closed control loops on AMs (such as AMs implement on managed resources) to validate change requests generated by AMs. Although not a 'trustworthy' solution in itself, King *et al.* [5] introduces an important concept (nested control looping) useful for the proposed trustworthy architecture as explained in Section III.

Another idea is that trustworthiness is achieved when a system is able to provide accounts of its behaviour to the extent that the user can understand and trust. But these accounts must, amongst other things, satisfy three requirements: provide a representation of the policy guiding the accounting, some mechanism for validation and accounting for system's behaviour in response to user demands [7]. The system's actions are transparent to the user and also allows the user (if required) the privilege of authorising or not authorising a particular process. This is a positive step (at least it provides the user a level of confidence and trust) but also important is a mechanism that ensures that any 'authorised' process does not lead to oscillation and/or instability in the system resulting in misleading or unreliable results. One powerful way of addressing this challenge is by implementing a *dead-zone* (DZ) logic presented in [8]. A DZ, which is a simple mechanism to prevent unnecessary, inefficient and ineffective control brevity when the system is sufficiently close to its target value, is implemented in [8] using Tolerance-Range-Check (TRC) object. The TRC object encapsulates DZ logic and a three-way decision fork that flags which action (left, null or right) to take depending on the rules specified. The size of the DZ can be dynamically adjusted to suit changes in environmental volatility. A key use of dead-zones is to reduce oscillation and ensure stability despite high extent of adaptability. A mechanism to automatically monitor the stability of an autonomic component, in terms of the rate the component changes its decision (for example when close to a threshold tipping point), was presented in [12]. The *DecisionChangeInterval* property is implemented in the AGILE policy language [12] on decision making objects such as rules and utility functions. This allows the system to monitor itself and take action if it detects instability at a higher level than the actual decision making activity.

A. *Trustworthy architecture life-cycles representing current practice*

We argue that trustworthiness cannot be reliably retrofitted into systems but must be designed into system architectures. We track autonomic architecture (leading to trustworthiness) pictorially in a number of progressive stages addressing it in an increasing level of detail and sophistication. Figure 1 provides a key to the symbols used.

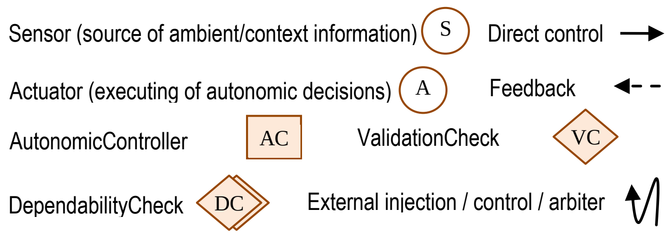
Sensor (source of ambient/context information)
S
Direct control
Actuator (executing of autonomic decisions)
A
Feedback
AutonomicController
AC
ValidationCheck
VC
DependabilityCheck
DC
External injection / control / arbiter

Figure 1. Pictographic key used for the architecture

Figure 2 illustrates the progression, in sophistication, of autonomic architectures and how close they have come to achieving trustworthiness. Although this may not be exhaustive as several variations and hybrids of the combinations may exist, it represents a series of discrete progressions in current approaches. Two distinct levels of sophistication are found: 1. The traditional autonomic architecture (a and b) basically concerned with direct self-management of controlled/monitored system following some basic *sense-manage-actuate* logic defined in AC. For the prevailing context, AC is just a container of autonomic logic which, could be based on MAPE or any other control logic. To add a degree of trust and safeguard, an external interface for user control input is introduced in (b). This chronicles such approaches that provide a console for external administrative interactions (e.g., real-time monitoring, tweaking, feedback, knowledgebase source, trust input, etc.) with the autonomic process. 2. On the horizon (c and d) are efforts towards addressing run-time validation. Systems are able to check the conformity of management decisions and where this check fails; VC sends feedback to AC with notification of failure (e.g., policy violation) and new decision is generated. An additional layer of sophistication is introduced (d) with external touch-point for higher level of manageability control. This can be in the form of an outer control loop monitoring over a longer time frame an inner (shorter time frame) control loop (e.g., as presented in [5]).

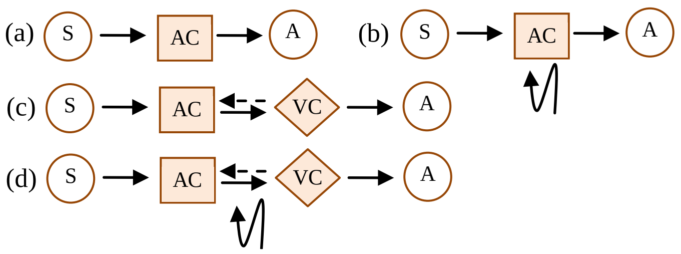

Figure 2. Pictorial representation of trustworthy autonomic architecture life-cycles.

At the level of current sophistication (state-of-the-art), there are techniques to provide run-time validation check (for behavioural and structural conformity), additional console for higher level (external) control, etc. Emerging and needed capabilities include techniques for managing oscillatory behaviour in ASs. These are mainly implemented in isolation. What is required is a holistic framework that collates all these capabilities into a single autonomic *unit*. Policy autonomics is one of the most used autonomic solutions. Autonomic managers (AMs) follow rules to decide on actions. As long as policies are validated against set rules the AM adapts its behaviour accordingly. This may mean changing between states. And when the change becomes rapid (despite meeting validation requirements) it is capable of introducing oscillation, vibration and erratic behaviour (all in form of noise) into the system. This is more noticeable in highly sensitive systems. So a trustworthy autonomic architecture (TAA) needs to provide a way of addressing these issues.

III. TRUSTWORTHY AUTONOMIC ARCHITECTURE

In this section we introduce our proposed TAA. We start with a general view of the architecture and then move on to explain its components. Figure 3 explains a trustworthy autonomic architecture that embodies self-validation and dependability. The architecture builds on the traditional autonomic computing solution (denoted as the *AutonomicController* component). Other components include *ValidationCheck* (which is integrated with the decision-making object to validate all *AutonomicController* decisions) and *DependabilityCheck* component which, guarantees stability and reliability after validation.

The *AutonomicController* component (based on e.g., MAPE logic, Intelligent Machine Design framework, etc.) monitors the managed sub-system for context information and takes decision for action based on this information. The decided action is validated against the system's goal (described as policies) by the *ValidationCheck* component before execution. If validation fails (e.g., policy violation), it reports back to the *AutonomicController* otherwise the *DependabilityCheck* is called to ensure that outcome does not lead to, e.g., instability in the system.

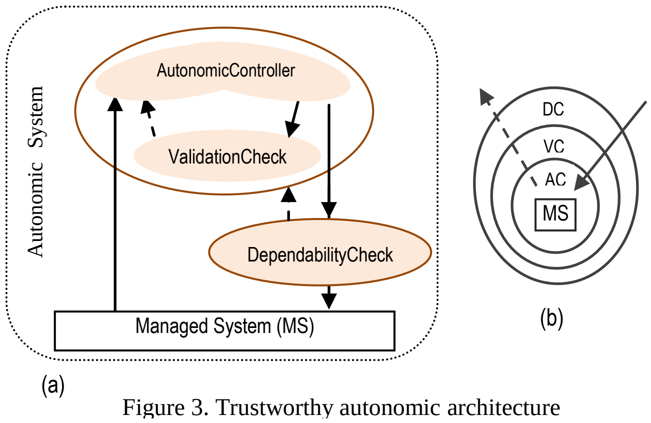

Figure 3. Trustworthy autonomic architecture

The *DependabilityCheck* component has a sub-component (*Predictive* sub-component) that allows it to predict the outcome of the system based on the validated decision. It either prevents execution and sends feedback in form of some input parameters to the *AutonomicController* or calls the actuator.

A. *Overview of the proposed architecture components*

We present the architecture in a number of progressive stages addressing it in an increasing level of detail. First, we define the self-management process as a ***Sense–Manage–Actuate*** loop where *Sense* and *Actuate* define Touchpoints (the AM's interface with a managed system) and '*Manage'* the embodiment of the autonomic management. Figure 4 is a detailed representation of the architecture.

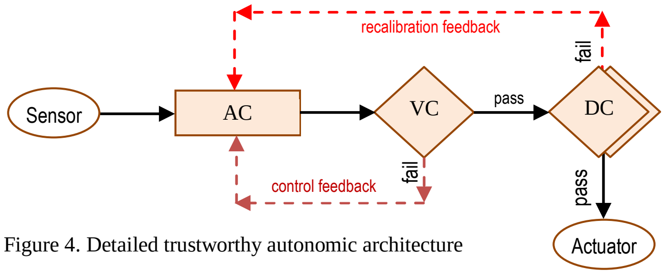

Figure 4. Detailed trustworthy autonomic architecture

Traditionally, the AutonomicController (AC) senses context information, decides (following some rules) on what action to take and then executes the action. This is the basic routine of an AM and is at the core of most of the autonomic architectures in use today (Figure 2). At this level the autonomic *unit* matters but the *content* of the *unit* does not matter much, i.e., it does not matter what autonomic logic (e.g., MAPE, IMD, etc.) that is employed as far as it provides the desired autonomic functionalities. So, the AC component provides designers the platform to express rules that govern target goal and policies that drive decisions on context information for system adaptation to achieve the target goal.

But, the nature of ASs raises one significant concern; input variables (context info) are dynamic and (most times) not predictable. Although rules and policies are carefully and robustly constructed, sensors sometimes do inject *rogue* variables that are capable of thwarting process and policy deliberations. In addition, the operating environment itself can have varying volatility –causing a controller to become unstable in some circumstances. Thus a mechanism is needed to mitigate behavioural (e.g., contradiction between two policies, goal distortion, etc.) and structural (e.g., illegal structure not conforming to requirement, division by zero, etc.) anomalies. This is where the ValidationCheck (VC) component comes in. It should be noted that AC will always decide on action(s) no matter what the input variable is. Once the AC reaches a decision, it passes control to the VC which then validates the decision and calls the Actuator (Figure 2c) or the DependabilityCheck (DC) (Figure 4) otherwise it sends feedback to AC if the check fails (while retaining previous *passed* decision). The VC is a higher level mechanism that oversees the AM to keep the system's goal on track. The ultimate concern here is to maintain system goal adhering to defined rules, i.e., adding a level of trust by ensuring that target goal is reached only within the boundaries of specified rules. It is then left for designers to define what constitute *validation pass* and *validation fail*. Actual component logic are application specific but some examples in literature include fuzzy logic [18], reinforcement learning [19], etc.

It is also important to consider situations above this level where, despite the AM taking legitimate decisions within the boundaries of specified rules, there's the possibility of overall inconsistency in the behaviour of the system. I.e., each individual decision could be correct (by logic) but the overall behaviour is wrong. A situation where the AM erratically (though legally) changes its mind, thereby injecting oscillation into the system, is a major concern especially in large scale and sensitive systems. Therefore it is necessary to find a way of enabling the AM to avoid unnecessary and inefficient change of decision that could lead to oscillation. This task is handled by the DC component. It allows the AM change its decision (i.e., adapt) only when it is necessary and safe to do so. Consider a simple example of a room temperature controller in which, it is necessary to track a dynamic goal –a target room temperature. The AM is configured to maintain the target temperature by automatically switching heating *ON* or *OFF*. A VC would allow any decision or action that complies with the basic logic 'IF *RoomTemperature* < *TargetTemperature* THEN *ONHeating* ELSE IF *RoomTemperature* > *TargetTemperature* THEN *OFFHeating*'. With the lag in adjusting the temperature the system may decide to switch *ON* or *OFF* heating at every slight tick of the gauge below or above target (when room temperature is sufficiently close to the target temperature). This may in turn cause oscillation which, can lead to undesirable effects. The effects are more pronounced in more sensitive and critical systems where such changes come at some cost. For example, a data centre management system that erratically switches servers between pools at every slight fluctuation in demand load is cost ineffective. One simple way of configuring a DC to mitigate this problem is by using dead-zone logic. In this case, a system has to exceed a boundary by a minimum amount before action is taken. Small deviations into the dead-zone do not result in actuations. The DC component may also implement other sub-components like *Prediction*, *Learning*, etc. This enables it to predict (based on knowledge, trend analysis, etc.) the outcome of the system and to decide whether it is safe to allow a particular decision or not. So after validation phase, the DC is called to check (based on specified rules) for dependability. DC avoids unnecessary and inefficient control inputs to maintain stability. If the check passes, control is passed to the Actuator otherwise feedback is sent to AC. DC is capable of tweaking input to the controller as feedback from its prediction. A particular aspect of concern is that for dynamic systems the boundary definition of DC may itself be context dependent (e.g., in some circumstances it may be appropriate to allow some level of changes which under different circumstances may be considered destabilizing).

Consider the whole architecture as a nested control loop (Figure 3b) with AC the core control loop while VC and DC are intermediate and outer control loops respectively. In summary, a system, no matter the context of deployment, is truly trustworthy when its actions are continuously validated (i.e., at run time) to satisfy set requirements (system goal) and results produced are dependable and not misleading.

IV. CASE EXAMPLE

This example is used to illustrate how powerful our proposed architecture is (in terms of cost savings, improved reliability and trustability) when compared to traditional architectures. We compare three autonomic managers that are based on AC (Figure 2a), AC+VC (Figure 2c) and AC+VC+DC (Figure 4). We use rule-based (policy autonomics) approach in this example.

The case example used deploys one of the current technology innovations –Autonomic Marketing. Autonomic Marketing employs the fundamentals of autonomic computing to monitor the market ambience and uses current (real-time) information to formulate appropriate marketing strategies for dynamic, adaptive and effective target marketing. The term is used to describe a step-change in the sophistication of automated marketing systems, in which the marketing activity itself is dynamically configured and contextualised to suit the current market conditions [9]. This has been proposed by the Autonomic Marketing Interest Group (AMIG) and they have in [9] defined some initial concepts and promise of the technology. An autonomic marketing system tracks current market state (which can be from several sources and is subject to influences such as market conditions, customer demographics, significant world events, trends from social media analysis, weather, seasonal information, etc.) and makes marketing decisions based on the analysis of the information gathered. This is representative of many real-world systems of high complexity and sensitive to several sources of environmental volatility.

In this example, we implement a particular aspect of Autonomic Marketing, that of targeted television advertising during a live sports competition airing. A company is interested in running an adaptable marketing campaign on television with different adverts (of different products appealing to audiences of different demographics) to be aired at different times during a live match between two teams. There are three adverts (Ad1, Ad2 and Ad3) to be run and the choice of an ad will be influenced by, amongst other things, viewer demographics, time of ad (local time, time in game, e.g., half time, TV peak/off-peak time, etc.), length of ad (time constraint), cost of ad, who is winning in the game, etc. This is a typical example of a system with many dimensions of freedom and very wide behaviour space. For brevity, we divide the behaviour space into four different zones and express them along two dimentions of freedom (*Mood* and *CostImplication*) as shown in Figure 5.

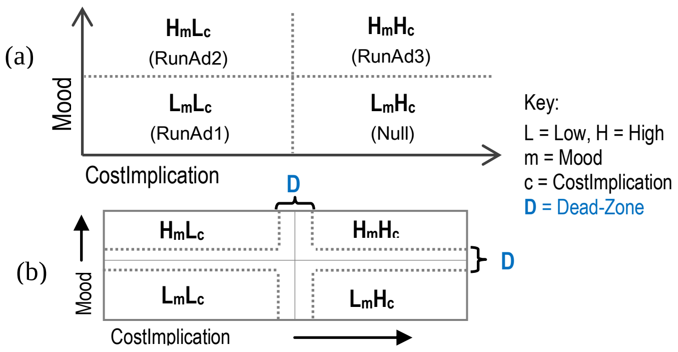

Figure 5. System behaviour space in two dimensions of freedom

The two dimensions of freedom represent a collation of all possible decision influencers into two key external variables –*Mood* and *CostImplication*. *Mood* is defined by two variables (*MatchScore* i.e., info about who's winning and *WeatherInfo*) while *CostImplication* is defined by another two variables (*TimeOfAd* and *LengthOfAd*). An action (in this case, RunAd 1, 2 or 3 or Null) is defined for each zone. Each action (ad) is thus activated only in its allocated zone following specified policy (excerpt shown in Figure 6). Internal variables (e.g., L_BenchMarkMatchScore and U_BenchMarkTimeOfAd), design-time specified, are used to define decision benchmarks.

```
If MatchScore < L_BenchMarkMatchScore And WeatherInfo < L_BenchMarkWeatherInfo Then
     Mood = "LOWMood"
  ElseIf MatchScore > U_BenchMarkMatchScore And WeatherInfo > U_BenchMarkWeatherInfo
          Then
      Mood = "HIGHMood"
  Else : Mood = "Null"
End If
If TimeOfAd < L_BenchMarkTimeOfAd And LengthOfAd < L_BenchMarkLengthOfAd Or
          TimeOfAd > U_BenchMarkTimeOfAd And LengthOfAd > U_BenchMarkLengthOfAd
          Then
     CostImplication = "LOWCostImplication"
  ElseIf TimeOfAd > U_BenchMarkTimeOfAd And LengthOfAd > M_BenchMarkLengthOfAd Then
     CostImplication = "HIGHCostImplication"
  Else :CostImplication = "Null"
 End If
Select Case DecisionParameter
    Case "LOWMoodLOWCostImplication"
         CurrentAction(CurrentActionCounter) = "RunAd1"
    Case "HIGHMoodLOWCostImplication"
         CurrentAction(CurrentActionCounter) = "RunAd2"
    Case "HIGHMoodHIGHCostImplication"
         CurrentAction(CurrentActionCounter) = "RunAd3"
    Case Else
         CurrentAction(CurrentActionCounter) = "NullAction"
 End Select
```

Figure 6. Excerpt of decision policy used.

The system goal is defined by a set of rules (Figure 7) that the AM must adhere to in making decisions. Basically, AC is concerned with making decisions within the boundaries of the rules while VC validates decisions for conformity with the rules. DC verifies that the measure of success is achieved. DC also improves reliability by instilling stability in the system. One way of achieving this is by introducing dead-zone boundaries (Figure 5b) within which, no action is taken (avoiding erratic and unnecessary changes) –in this case, a running ad is not changed. The size of the boundaries, which, though can be dynamically adjusted to suit real-time changes, is initially design-time specified.

1. Extract external variables (decision parameters) at defined time interval and decide on action
2. Send trap msg and change action if (*condition omitted*) otherwise retain previous action
3. …
4. If current action is same as previous action, do not send trap and do not change action
   ================Measure of Success================
5. Cost of action change (total ad run) must fall within budget
6. Rate of change should be considerably reasonable
7. …
8. Turnover should justify cost

Figure 7. Excerpt of rules defining system goal.

AC will, at every sample collection, decide (by running the policy in Figure 6) which action (ad) to run. Because it is wired to make fresh decision at every policy run, it is bound to send trap (notice of change of ad). But before that decision is implemented, VC validates it for *pass/fail*. It is important to define what *pass/fail* means in this context: if decided action is same as previous action (running ad), VC returns *fail* (then no trap is sent and no change is made) and passes control to AC while retaining previous action. VC also returns *fail* if policy is violated in decision making, i.e., decision must be within the boundaries of specified benchmarks (e.g., a "Null" return should not influence action change). Control is passed to DC each time VC returns a *pass*. DC is concerned with the measure of success aspect of the rule. In this case, a TRC (Tolerance-Range-Check) is implemented: DC returns *fail* if ActionChange is more than one within the first five sample collections and subsequently if action changes at every sample instance. So DC maintains action change at maximum of one within the first five sample collections and subsequently maximum of two in any three sample instances. This will help calm any erratic behaviour that could arise. Take for instance, there could be a 360 degrees change in 'Mood' within a short space of time (e.g., a team's status in a game can change from *winning→losing→winning* within a very short space of time) which is capable of adversely affecting the choice of an ad. Figure 8 (a) and (b) are excerpts of managers of VC and DC, respectively.

The need for a new and different approach is reinforced by the capabilities exhibited in DC. It addresses situations where it's possible for overall system to fail despite process (in terms of structural, legal, syntactical, etc.) correctness.

(a)
```
If Mood <> "Null" And CostImplication <> "Null" Then
     DecisionContainer(IntervalCounter) = Mood & CostImplication
     DecisionParameter = DecisionContainer(IntervalCounter)
  '<Omitted>
  '<Omitted>
  '<Omitted>
End if
If CurrentAction(CurrentActionCounter) = CurrentAction_
(CurrentActionCounter - 1)  Then
    'CurrentAction = CurrentAction(CurrentActionCounter - 1)
   '<Omitted>
   '<Omitted>
   '<Omitted>
```
(b)
```
If IntervalCounter - IntervalCounterDC(Interval - 1) > 4 Then
    ActionChangeCounterDC = ActionChangeCounterDC + 1
  '<Omitted>
  '<Omitted>
  '<Omitted>
End if
```
Figure 8. Excerpt of VC and DC managers

In the experiment presented here, a computer program is written to simulate three autonomic managers (AC, AC+VC and AC+VC+DC). Four external variables, now referred to as context samples, (*MatchScore*, *WeatherInfo*, *TimeOfAd* and *LengthOfAd*) are fed into the managers at every sample collection instance. Sample collection instances are defined by a set time interval which can be fixed (design-time specific) or dynamically tuned. Based on policies (Figure 6), the managers decide how, when and which ad to change. The simulation was run for a total duration of 50 sample collection instances. During this duration, the managers are analysed for total number of ad changes and the distributions of those changes. For accurate analysis and comparison, the same sample at the same time instance and interval are fed into the managers concurrently. Samples may (most likely) change at every time instance and separately feeding these to the managers will lead to unbalanced judgment.

A. *Experimental Results*

Results presented are for a simulation of 50 sample collections. All three autonomic managers (AC, AC+VC and AC+VC+DC) are analysed based on number of ad changes and number of ad distributions.

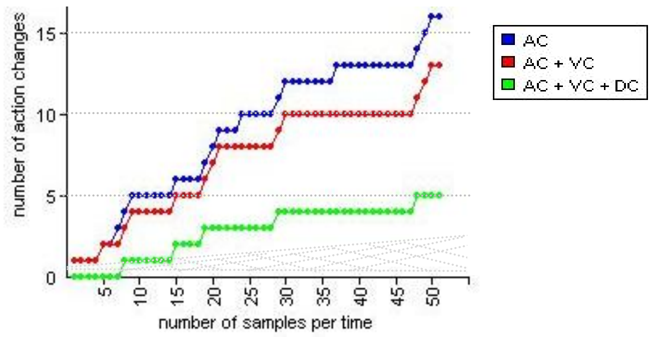

Figure 9. A sample of managers' behaviour in a 50 sample collection. (Note: If printed in black/white, the top graph is AC followed by AC+VC and then AC+VC+DC)

The optimisation of the proposed architecture in this autonomic marketing scenario is in terms of achieving balance between efficient just-in-time target-marketing decision and cost effectiveness (savings maximisation) while maintaining improved trustability and dependability in the process. Figure 9 shows the behaviour of the managers in 50 sample collections in a game duration in which the proposed architecture (AC+VC+DC) shows significant gain in stability and cost savings. It's clearly seen, for example, how (AC+VC+DC) smoothened the high fluctuation rate (high adaptability frequency) experienced between the 5th and 25th sample collections. In general, the average ad change ratio of about one change in three samples (1:3) is reduced to one change in ten samples (1:10), representing an overall gain of about 31.25% in terms of stability and cost efficiency.

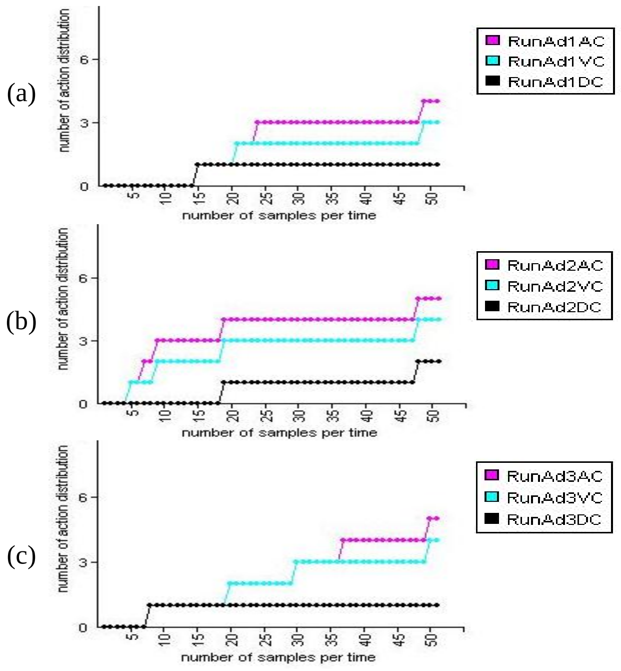

Figure 10. A distribution of the ads (Ad1, Ad2 and Ad3).
(Note: If printed in black/white, the top graph is AC followed by VC and DC)

Figure 10 shows the distribution of ads across the 50 sample duration ("NullActions" i.e., 'run no ad' are not shown). This also corroborates the significant gain by the DC component. In (c), for example, only one Ad3 is run while two Ad2 are run in (b) by the (AC+VC+DC) AM. This directly translates to adaptive cost savings. Recall from Figure 5(a) that Ad2 is run when *Mood* is high and *Cost* is low (best value for money) while Ad3 is run when *Mood* and *Cost* are both high (when it costs more to run an Ad).

While it has been shown that the proposed approach is capable of maintaining reliability by reducing inefficient adaptation (cutting off unnecessary adaptations), it should be noted that reducing alone is not the answer. If the rate is very low it will not be right either. For example, if the behaviour of the manager falls within the shaded area of Figure 9, it shows that the manager is almost inactive (or not making decisions frequently enough). For every application, it is necessary to determine which rate is appropriate or cost effective in the long run. The proposed approach provides a way for tuning this (e.g., through adjusting the width of the TRC dead-zone).

There is a cost associated with bad or over frequent decisions and also a cost with not making frequent enough decisions. Success is measured by striking a balance between the two.

V. CONCLUSION

A new architecture for trustworthy autonomic systems has been presented. Different from the traditional autonomic solutions, the proposed architecture consists of mechanisms and instrumentation to support run-time self-validation and trustworthiness. At the core of the architecture are three components, the AutonomicController, ValidationCheck and DependabilityCheck, which allow developers specify controls and processes to improve system trustability. An analysis of the current state of practice in autonomic architecture shows that a new approach is required in which validation and trustworthiness are not treated as add-ons as they cannot be reliably retro-fitted to systems. Validation alone does not always guarantee trustworthiness as logical processes/actions could sometimes lead to overall system instability. There are situations where, for example, despite the autonomic manager's legitimate decisions within the logical boundaries of specified rules, there's the possibility of overall erratic behaviour or inconsistency in the behaviour of the system. This is why autonomic systems need a new approach.

To demonstrate the feasibility and practicability of our approach, a case example scenario has been presented. The case scenario demonstrates how the proposed architecture can maximise cost, improve trustability and efficient target marketing in a company-centric Autonomic Marketing System that has many dimensions of freedom and is sensitive to a number of contextual volatility. As this approach is new, future research work will focus on improving the robustness of the proposed architecture. This includes adding a predictive/learning sub-component to the DependabilityCheck component and verifying how results of this approach can vary in other contexts to see which factors could influence its adoption or not in practice.

ACKNOWLEDGEMENT: Graphs are plotted using csXGraph ActiveX / OCX control by Chestysoft [10].

REFERENCES

[1] IBM, *An architectural blueprint for autonomic computing*, IBM Whitepaper, 2004

[2] Kephart Kephart and David Chess, *The Vision of Autonomic Computing*. Computer, IEEE, Volume 36, Issue 1, January 2003, pp. 41-50

[3] Andrew Diniz, Viviane Torres, and Carlos José, *A Self-adaptive Process that Incorporates a Self-test Activity*, Monografias em Ciência da Computação, No. 32/09, Rio De Janeiro – Brasil, Nov. 2009.

[4] Xiaolin Li, Hui Kang, Patrick Harrington, and Johnson Thomas, *Autonomic and trusted computing paradigms*, In Proceedings of ATC'2006, pp. 143-152

[5] Tariq King, Djuradj Babich, Jonatan Alava, Peter Clarke, and Ronald Stevens, *Towards Self-Testing in Autonomic Computing Systems*, Proceedings of the Eighth International Symposium on Autonomous Decentralized Systems (ISADS'07), Arizona, USA, 2007

[6] Tariq King, Alain Ramirez, Peter Clarke, and Barbara Quinones-Morales, *A Reusable ObjectOriented Design to Support SelfTestable Autonomic Software*, Proceedings of the 2008 ACM symposium on Applied computing, Fortaleza, Ceara, Brazil, 2008, pp. 1664-1669

[7] Stuart Anderson, Mark Hartswood, Rob Procter, Mark Rouncefield, Roger Slack, James Soutter, and Alex Voss, *Making Autonomic Computing Systems Accountable*, Proceedings of the 14th International Workshop on Database and Expert Systems Applications (DEXA), 2003

[8] Richard Anthony, *Policy-based autonomic computing with integral support for self-stabilisation*, Int. Journal of Autonomic Computing, Vol. 1, No. 1, pp. 1–33. 2009

[9] Carl Adams, Richard Anthony, Wendy Powley, David Bell, Chris White, and Chun Wu, *Towards Autonomic Marketing*, The 8th International Conference on Autonomic and Autonomous Systems (ICAS), pp. 28-31, St. Maarten 2012.

[10] Chestysoft csXGraph:www.chestysoft.com/xgraph/instructions.pdf Last accessed date 29th June 2012.

[11] Thaddeus Eze, Richard Anthony, Chris Walshaw, and Alan Soper, *Autonomic Computing in the First Decade: Trends and Direction*, The 8th International Conference on Autonomic and Autonomous Systems (ICAS), pp. 80-85. St. Maarten 2012.

[12] Richard Anthony, *Policy-centric Integration and Dynamic Composition of Autonomic Computing Techniques*, The 4th International Conference on Autonomic Computing (ICAC), 2007, Florida, USA

[13] Markus Huebscher and Julie McCann, *A survey of autonomic computing—degrees, models, and applications*, ACM Computer Survey, 40, 3, Article 7 (August 2008)

[14] Christoph Reich, Kris Bubendorfer, and Rajkumar Buyya, *An autonomic peer-to-peer architecture for hosting stateful web services*, The 8th IEEE International Symposium on Cluster Computing and the Grid (CCGRID 08), pp. 250-257, 2008.

[15] Fang Mei, Yanheng Liu, Hui Kang, and Shuangshuang Zhang, *Policy-based autonomic mobile network resource management architecture*, The 2nd International Symposium on Networking and Network Security (ISNNS 10), pp. 144-148, April 2010.

[16] Joao Ferreira, Joao Leitao, and Luis Rodrigues, *A-osgi: A framework to support the construction of autonomic osgi-based applications*, Technical Report RT/33/2009, May 2009.

[17] Haffiz Shuaib, Richard Anthony, and Mariusz Pelc, *A Framework for Certifying Autonomic Computing Systems*, The 7th International Conference on Autonomic and Autonomous Systems (ICAS), pp. 122-127, 2011, Venice, Italy

[18] Ting-Jung Yu, Robert Lai, Menq-Wen Lin, and Bo-Rue Kao, *A Fuzzy Constraint-Directed Autonomous Learning to Support Agent Negotiation*, The 3rd International Conference on Autonomic and Autonomous Systems (ICAS), pp. 28, 2007, Athens, Greece

[19] Han Li and Srikumar Venugopal, *Using Reinforcement Learning for Controlling an Elastic Web Application Hosting Platform*, The 8th International Conference on Autonomic Computing (ICAC), pp. 205-208, 2011, Karlsruhe, Germany