Self-healing for automotive control systems

26th May 2014

Richard John Anthony
University of Greenwich
Greenwich, London, UK
R.J.Anthony@gre.ac.uk

Roadmap

Definition of self-healing as a strand of autonomic computing

Specific challenges of automotive control and identification of opportunities for autonomic computing in this application domain

Context and context awareness

Architectures for autonomic systems

Choosing the reasoning system

First thoughts on applying self-healing to automotive control systems, in the context of Fault Tree Analysis
The need for systems that self-manage

The software complexity crisis causes a dilemma:

Going back to simpler systems seems unlikely in general; ► the other fork requires that we find ways to deal with the complexity.

The approach is to develop software components whose job is to manage the runtime behaviour of other software components.

► This implies that humans hand over low-level control, fine tuning and repetitive set-up and configuration actions to a software manager.

The software manager monitors the system state and/or behaviour and performs dynamic, self-adaptive actions to achieve low-level and frequent configuration to ensure the system meets its higher level goals.

Despite no-longer being in complete control of every aspect of behaviour, at a higher level the human manager should still govern, appreciate, have confidence in:

- The overall goal of the system,
- The longer term trends in behaviour,
- The general correctness / effectiveness of the system.

Self-healing

An adjustment or adaptation is made to mask or recover from an error or failure.

Related concepts:
- Fault-tolerance – being able to continue to operate despite faults occurring
- Failure transparency – hiding / masking faults

From an autonomic computing system point of view, a smart approach to handling faults is to be able to reconfigure such that the external view (i.e. the functional behaviour) is maintained.

Examples:
- Returning default fail-safe actions when input data is out of range – for example if a sensor is faulty it may send a meaningless value to the controller. The controller needs to determine the difference between an intended value and a value from a faulty sensor – perhaps by looking at the patterns of noise on the signal, or considering how abrupt the signal changes are.
- Dynamically re-coupling components to circumvent a failed component.
Other aspects of self-management

Self-configuration
The architecture or behaviour is reconfigured to meet changes in operating requirements or operating circumstances.

Self-optimisation
Adjustments to structure or behaviour to improve performance, effectiveness or efficiency.

Self-protection
Adjustments or adaptations are made to deal with (ideally to prevent) threats. This is concerned with robustness, in addition to security – a threat having a wider interpretation than ‘security threat’.

These forms of self management are not orthogonal – it is possible that a particular action fulfils a healing and also configuration role, for example.

Additional autonomic behaviours

Self-stabilisation will become increasingly important as the functionality and deployment scale of systems increases.

Self-validation is a hot current research topic. Once a system has adapted it should be able to test the effectiveness of the new configuration.

Self-evolvability. Leading-edge current research targets evolvable systems (i.e. beyond adaptive systems, where adaptive implies that the fundamental algorithmic basis of logic is not changed, but tuned, and evolvable implies that completely new logic can be discovered / learned or can emerge).

For evolvable systems, Self-awareness is necessary, in order that a system can appreciate the boundary between itself and the system it manages, and can thus reason about the effect it is having on its managed system – and thus can evaluate its own performance.
Waypoint

Definition of self-healing as a strand of autonomic computing

Specific challenges of automotive control and identification of opportunities for autonomic computing in this application domain

Context and context awareness

Architectures for autonomic systems

Choosing the reasoning system

First thoughts on applying self-healing to automotive control systems, in the context of Fault Tree Analysis
Complexity in automotive control / computing systems

Large scale of systems
- Many components within an application (connectivity, dependency, synchronisation)
- Many applications within a system (interactions, competition for resources)
- Different demands / requirements / priorities at various levels

Heterogeneity
- Many computational nodes, with differing physical configuration (resources, O/S)

Dynamic behaviour
- Dynamic network conditions (connectivity, traffic / congestion, message loss)
- External device connectivity
- Component failures

Software lifecycle
- New features / upgrades
- Version explosion / backwards (in)compatibility
- On-line updates

Security aspects
- Evolving threats
- Changing protection needs

The need for dynamic response in automotive control systems

Characteristics include:
- safety critical function
- unpredictable faults
- complex sequences
- fast operation rate
- many interacting parts
- dynamic environments

These systems need to deal with various issues in Real-time

Therefore
- Need to dynamically prioritise
- Need to dynamically adjust sensitivity
  (to changes in sensor values, to background noise)
- Need feedback and in some cases a means of verifying their own correctness
  (imagine how humans learn new tasks / learn to solve new problems)
Waypoint

Definition of self-healing as a strand of autonomic computing

Specific challenges of automotive control and identification of opportunities for autonomic computing in this application domain

Context and context awareness

Architectures for autonomic systems

Choosing the reasoning system

First thoughts on applying self-healing to automotive control systems, in the context of Fault Tree Analysis

Context in an automotive control system

Context is information about the current circumstances under which a system operates.

Context for the control system could include:
- Current set of services demanded (and their resource implications)
- Current resource usage, capacity
- Set of components present and their status: operating, failed, standby
- System history (previous states, previous faults, MTBF, trends etc.)

Context at the vehicle level could include:
- Location (country/region, proximity to a safe place to get help, proximity to convoy members)
- Mobility (speed, body roll, brakes temperature)
- Identity of user (user customisation, driving style, risk rating e.g. excessive dependency on brakes)
- Time (night/day, driver alertness)
- Urgency (mission, cargo, remaining distance, remaining battery)
**Context Awareness**

Context awareness implies that the system behaviour takes the operating context into account, i.e. dynamic adjustment of behaviour to suit environmental conditions:

1. Detect / recognise the current operating conditions
2. Determine if the current operation is suitable for the conditions
3. Change the behaviour to suit the specific operating conditions

Simplest form (see diagram):
- Two different control modes
- Switches between them based on some context input.

Mode-based example, context-awareness logic selects from pre-defined configurations

---

**Waypoint**

Definition of self-healing as a strand of autonomic computing

Specific challenges of automotive control and identification of opportunities for autonomic computing in this application domain

Context and context awareness

**Architectures for autonomic systems**

Choosing the reasoning system

First thoughts on applying self-healing to automotive control systems, in the context of Fault Tree Analysis
Architectures for Autonomic Computing

The fundamental architecture of an autonomic manager is a control loop:
Monitor → Analyse → Plan → Execute

in which the controlled component completes the loop.
Almost always these four steps require the ability to generate, store and access shared Knowledge hence the model is referred to as MAPE-K

The MAPE-K model - comprises four main functions

Monitor – The managed element is monitored. For most software systems this will comprise an instrumentation interface on the managed component. Indirect sensing can also be used; for example monitoring the output of a component, or the effect the component has on some system parameter.

Analyse – Using the data gathered by ‘Monitor’; determine if the system is on track to achieve current goal. Specifically, is the managed component doing what it should be doing, is it tuned correctly for current conditions. May take into consideration additional context information concerning the wider state of the system (not directly sensed from the managed component) – this is not explicitly represented in the MAPE model.

Plan – Based on the results from ‘Analyse’, determine if, and what, action is needed to adjust the managed component’s behaviour in line with current system goals and / or environmental conditions.

Execute – Implement the ‘Plan’. Using the Effector interface to the managed component, modify its behaviour.
MAPE-K – fit for purpose?

MAPE-K has been almost universally accepted for the first decade.
- Is it a good architecture for all scenarios?
- Does it have suitable flexibility?

It is essentially a more useful generic model or guide.
Rarely is it implemented in a pure sense:
- Some implementations combine logical functions into single entities
- Some separate the logical functions into separate components
- Some have different knowledge models – e.g. is passed through the chain rather than accessible across the four components

The MAPE-K control loop does not distinguish between different types of situation in the controlled system; its response is the same structurally in all cases (M → A → P → E).

Many implementation variations on MAPE-K exist.

The Intelligent Machine Design architecture (IMD)
IMD is significantly different to MAPE; both structurally, but most importantly behaviourally.

An alternative model in which behaviours are differentiated in terms of urgency of response and familiarity of conditions:

Consider (a human ... for now) driving a car in different scenarios:
- Finding yourself completely lost in a place you have never visited before having trusted the sat-nav – This is a new situation, need to plan a unique solution.
- Taking a corner – well known problem, practised many times but needs unique adaptation to each specific circumstance.
- Emergency stop – instinctive / rapid (pre-practised action, no time to ‘think’).

The Intelligent Machine Design model has three layers to handle events at three levels of complexity:
- Reflection Devise a new strategy to solve a new problem
- Routine Solve a familiar type of problem, but specifically parameterised
- Reaction Immediate response to a precisely known situation

See [1] and [2] for a detailed discussion of how IMD can be used in Autonomic Systems.
The Intelligent Machine Design architecture (IMD)

An architecture based around hierarchical control loops operating on different timescales in order that a component can:

1. Monitor its own correctness and performance
2. Monitor its own effect on the controlled system (and hence detect any instability caused)

There are three components:

- Autonomic Controller (AC) - makes a self-management decision
- Validation Check (VC) - monitors correctness and performance of AC
- Dependability Check (DC) - monitors impact on system / stability / long term effectiveness

See [3] for more details of the TAA.

The Trustworthy Autonomic Architecture (TAA)

An architecture based around hierarchical control loops operating on different timescales in order that a component can:

1. Monitor its own correctness and performance
2. Monitor its own effect on the controlled system (and hence detect any instability caused)

There are three components:

- Autonomic Controller (AC) - makes a self-management decision
- Validation Check (VC) - monitors correctness and performance of AC
- Dependability Check (DC) - monitors impact on system / stability / long term effectiveness

See [3] for more details of the TAA.
Loops within loops: the Autonomic Controller (AC) and Validation Check (VC) operate on shorter timeframes than the Dependability Check (DC).

**Trustworthy Autonomic Architecture**  (from [3] Eze and Anthony)
**Self-validation 1**

Self-Validation implies that an autonomic controller checks the effects of its own short-term actions, to determine whether it is having the desired effect on the system over a longer term.

It is a current research challenge.

Self-Validation can be achieved in two main ways:

**Reactive:** An outer control loop monitors the overall trend in behaviour of the system at a higher timescale than the lower level autonomic loop. This provides feedback to the controller concerning the actual impact it is having on the system.

**Key to figure**

Red inner loop is normal autonomic control loop, operates at a relatively fast rate.

Purple outer feedback loop operates at a slower rate and enables the manager to evaluate its own impact / effectiveness over time.

**Self-validation 2**

**Pro-active self-validation:** a run-time model of the managed system and its current state is constructed within the controller (primarily within the Plan function).

The proposed adaptation is applied initially to the model, to see if it predicts an actual improvement in performance.

The change is only applied where non-marginal improvements are predicted.
Waypoint

Definition of self-healing as a strand of autonomic computing

Specific challenges of automotive control and identification of opportunities for autonomic computing in this application domain

Context and context awareness

Architectures for autonomic systems

Choosing the reasoning system

First thoughts on applying self-healing to automotive control systems, in the context of Fault Tree Analysis

Policy-based computing

A technique where the business function / intent is expressed as a set of rules or configuration statements (a policy), and kept detached from the implementation mechanism.

The mechanisms are fixed at design time, but the actual behaviour is dependent on the policy, which is usually fixed for a given execution instance, but can be changed between executions.

Policy-based techniques are popular within the autonomies / self-management arena because they offer a means of dynamic reconfiguration.

Policy-based configuration enables a system’s configuration to be specified externally to the implementation mechanism. The system’s behaviour can be subsequently modified without changing the code-base or its deployment (i.e. no recompilation).

Usually the policy is written in a script.
What is a ‘Policy’?

Ranges from a simple Boolean condition to Event-Condition-Action rules, or (in the case of AGILE and some other advanced systems) a script comprising several rules and other decision-making components.

Basic concept ► change policy = change behaviour

Context information

Key benefits of Policies

- Allows late specialisation – specific functionality can be decided at point of commissioning (reduces production costs and lead time).
- Supports new use-cases not perceived at time of design / commissioning (future proofing).
- Facilitates context-aware behaviour (sensitive to operating conditions).
- Facilitates field-upgrades of software to enable the latest innovations to be added post-manufacture (long product lifetime).
- Supports flexible personalisation of applications.
- Enables post-sales upgrade for optional functionality (after sales support).
**Policy-based Context-aware control**

- Context can be:
  - Explicitly passed in,
  - Implicit from behaviour (such as event sequences),
  - Determined from interaction with other services.

- Policies are:
  - Held externally,
  - Run-time loadable.

**Interaction with other services**

*Any service or application component*

- Embedded policy engine
- Dynamic modification of behaviour

---

**Policy runtime environment**

- Context information (user preferences, sensors values, resource levels, etc.)
- Context Manager service

**Software component**

- Dynamic Wrapper (DW)
- AGILE-Lite policy evaluation module
- Decision Point (DP)

**Repository Manager service**

- Repository holds policies. Meta data relates a policy to a specific DP.
This new configuration has introduced an unexpected resource shortage.
**Self-stabilisation**

A control system can potentially cause instability. In particular when a system is close to some threshold, or goal state, a small change can push it over to the other side. Likewise, a second change, as correction, can push it back. It is possible for such systems to oscillate around the goal state but never reach it. This is particularly problematic for narrowly defined goal states.

► A truly ‘smart’ system should be able to **monitor the effects of its control actions** on the controlled system – and thus ‘self-stabilise’.

A **Deadzone** is a simple technique that can be used to counteract oscillatory tendencies. A Deadzone is a range of system states over which the control action is suspended.

► Effectively the Deadzone increases the width of the goal target, making it easier to hit, at the cost of reducing the precision by which the goal is defined. (see next slide)

---

**The need for stability in dynamic goal tracking**

- Hysteresis (lag) and/or control overshoot leads to oscillation in highly-tuned systems.
- A deadzone can reduce unnecessary control adjustments.
**Tolerance Range Check object (a specialised rule variant)**

Three way decision fork – useful complement to Boolean logic.

- Implements a **deadzone**, for stability.
- Increases flexibility of policy logic.
- Facilitates fuzzy matching of variables.
- ‘Tolerance’ is dynamically configurable.

---

**DySCAS Modal Use case – Load Balancing**

*(Dynamically Self-Configuring Automotive Systems – EC funded project 2006-2009)*

The Autonomic Configuration Manager manages the load balancing function

There are two main levels of decision:

- Should load balancing be active / disabled?
- Which configuration (algorithm / tuning) of the load balancer to use?

The functionality is split into two Decision Points to both maximize change flexibility and to simplify validation of dynamic aspects.

Safety / robustness is ensured in several ways:

- Each DW locally detects and handles faults
- Default behaviour is carefully selected by the component developer
- The actual LB algorithms are fixed, statically verified code
Utility Functions (UF)

UF provide a means of choosing from several options, based on the *instantaneous* merit of each option.

This technique allows context-sensitive decisions to be made dynamically.

Multi-dimensional problems (many different context parameters, each with a possibly *dynamic* weight) can be resolved with relatively low implementation complexity when compared to some of the alternatives provided by various artificial intelligence techniques such as artificial neural networks.

Utility functions make low demands on run-time processing and storage resources and can combine many contextual factors into one decision. Thus they are a good candidate technology for implementing autonomic computing in systems that use embedded controllers and other resource-constrained environments – such as sensor networks.

Utility Functions (continued)

A summative utility function can be expressed:

\[ U = (W_1 \times I_1) + (W_2 \times I_2) + (W_3 \times I_3) \]

Where \( W_1, W_2 \) and \( W_3 \) are weights and \( I_1, I_2 \) and \( I_3 \) are contextual inputs, \( U \) is the derived utility value for a particular option.

The weights may be dynamically adjustable to allow adaptation.

During the actual computation of the utility of different options, the weights are fixed so they are applied consistently for each option computed.

Each contextual parameter \( \{ I_1, I_2, I_3 \} \) will have dynamic option-specific values.

Since all RHS values are known it is straightforward to determine the instantaneous merit of each option \( \{ U_1, ... U_n \} \).
Self-stabilisation support at the policy level

Need means by which policy logic can detect unstable behaviour
– auto-generated properties:
  *SameDecisionCount*, *MostRecentDecision*, *DecisionChangeInterval*

Need means by which policy logic can reconfigure itself
– indirect addressing:
  *Variables of type* 'PolicyObject' (hold name of rule, action, UF etc.)

Waypoint

Definition of self-healing as a strand of autonomic computing

Specific challenges of automotive control and identification of opportunities for autonomic computing in this application domain

Context and context awareness

Architectures for autonomic systems

Choosing the reasoning system

First thoughts on applying self-healing to automotive control systems, in the context of Fault Tree Analysis
General Self-healing Strategy (loosely based on MAPE)

Detect fault or anomaly (fault MAY have occurred, MAY be about to occur)
(opportunity to detect patterns in the detectors own performance and in wider situation)
- How often has the fault been detected, how frequently, how recently?
- Is it the same component?
- Is it the same type of component?
- Was previous fix successful (why are we here again?)

Confirm fault (avoid false positives, avoid unnecessary actions)
(opportunity to self-tune)

Determine possible actions
- Benefits versus risks for each
- History of fixes and their effectiveness
- Predictive model of effectiveness of each possible fix?

Select ‘best’ option for current scenario (choose best option in context-aware way)
Utility of each fix may be context-dependent - current situation must be taken into account as well as the actual fault.
(opportunity to detect instability in decision making and to retune to self-stabilise)
- Consider the history of the system!
- Consider the history of interventions!!

Apply action and update logs

Applying principles of self-healing to automotive control - first thoughts

Dynamically configurable fault trees allow for different variants of a particular vehicle:
- When a vehicle has optional components, or upgrades, the fault model may change.
- It may be possible to use concepts drawn from policy-based configuration to facilitate dynamically loadable fault-tree components.

Error-prevention or prediction by incorporating some form of pre-error information, and analysing the tradeoffs of early maintenance costs and nuisance factors, against longer-term safety and robustness.
- Utility functions are a flexible way to perform run-time configuration choices, based on dynamic, weighted utility, or cost values.
Proactive approach

Augmentation of fault-tree logic with probability information
Makes the fault tree a real-time tool, as the system behaviour, current state, history of faults can inform the probability values.

Probability driven dynamic reconfiguration of fault trees?

Augmentation of fault-tree logic with environmental-context awareness
It may be possible to discover that certain types of faults are more likely to occur in certain circumstances, or have different severity depending on circumstances.

New patterns can be detected between not-known-to-be causally-related failures.

For (mechanical) example: a worn front wheel bearing might put more stress on the brake calliper during stopping, and thus while it does not directly lead to failure, it contributes over time. The addition of manufacturer’s specification information of the component, and the type of road surfaces driven on (as forms of context) could lead to smart failure prediction.

Synergies and Integration
Ways in which an Autonomic Manager (AM) could be combined with real-time control:

A) AM selects one of a pre-designed set of fixed Fault Tree Models (Mode-based).
   - Facilitates re-use of a bank of models.
   - Allows flexibility / scalability yet keeps complexity under control.
   - System can be easily extended to cope with initially unseen situation
     (only need to add a new pre-validated ‘model’ and upgrade the selection policy).

B) AM configures / customises / tunes a Fault Tree Model
   - Facilitates deferment of some specific decisions; faster development and greater flexibility.

C) AM dynamically composes modular Fault Model
   - Modular fault trees composed at run time
   - Dynamic re-mapping of fault modules to synchronise with system upgrades

D) AM incorporates a Fault Tree Model as a decision making component within a wider control framework.
   - Various types of reasoning can be combined to achieve greater intelligence / scope.

E) AM supervises a Fault Tree Model (self-validation)
   - Can ensure continued correctness / robustness / stability / effectiveness despite changes in environment.
Integration model A – MODAL
- AM dynamically selects appropriate Fault Tree Model
- Models can have narrower scope → higher optimality and easier to validate

Monitor environment, e.g., noise, volatility, linear vs non-linear areas ...
Policy upgraded to select new model when appropriate
Policy selects model based on suitability for current environment conditions
Environment state-space

Several models \( \{M_1, \ldots, M_4\} \) optimally cover wide or non-linear state space

\( M_n \) = Fault Tree Model instance

DySCAS Modal Use case – load balancing

General load level
Distribution of load across ECUs
Types of tasks present
Task priorities / periodicities
Current distribution of tasks
Current system volatility

Determine whether load balancing should be used
Post-deployment upgrade with more-sophisticated logic
Determine which load balancing technique to use

Fixed, validated load-balancing algorithms

KEY:
- Static function block
- Decision Point (DP)
- Dynamic Wrapper (DW)
- Default output of DP
Load Balancing  Normal behaviour

Context

1

2

A B C

Load Balancing  DW #2 detects run-time failure
(illustrates safety despite independent failures)

Context

1

2

A B C

DW #1 operates correctly

DW #2 detects run-time problem:
- No policy loaded
- Policy caused parse error
- Required context not available

DW #2 enforces default output for DP #2
Integration model B – AM Configures Fault Tree Model
- Policy contextually configures / tunes the Fault Tree Model
- Can change tuning policy (and contextual dependency) after system deployed

Integration model C – AM dynamically Composes Fault model
- Modular fault model reflects modular physical system
- Supports in-field upgrades of components
1. Fault model reflects physical configuration
2. Components removed from physical system (fault model updated)
3. New components added to physical system (fault model updated)
4. Fault model reflects physical configuration
Integration model D – AM Incorporates Fault Tree Model
- Fault Tree Model is a decision-making component in wider-scoped logic

Integration model E – AM Supervises Fault Model
- Policy monitors effectiveness / efficiency / stability of Fault Model
- Fault Model is tuned / swapped / disabled as required in the specific application and context
- Policy operates at a higher logical level than the Fault model (as opposed to model B, where policy ‘internal’ to Fault Model Component)
Challenges

How to model/capture the state of the system as input to a run-time model
- To what extent is historical state included
- Control loops in safety critical systems are based primarily on instantaneous state (and perhaps recent history – the previous time step only?)
- History allows trend analysis, prediction.

Are intermittent faults allowed?
- What is the system's response if a failed component comes back on line?

Conflicts may be possible?
- What to do if the run-time and design time components conflict?

The self-management system must not:
- Inject noise, e.g. in the form of unnecessary changes (self-stabilisation, TAA?)
- Use excessive resource
- Add excessive latency (IMD architecture?)
- Introduce new failure modes (self-validation?)

(Stability is a specific challenge – especially near margins)

How to dynamically upgrade a fault model
- Post-production discovery of latent faults
- Synchronisation of fault tree model with system upgrades in future

Tradeoffs

Self-management requires resources

Marginal improvements or false triggers have an overall cost on the system – avoid

Context-awareness needed, may need variable thresholds / triggers for some corrective behaviours.
Will need a means to rank errors in terms of importance and short-term urgency
- How to deal with two errors occurring at same time.
- Under what circumstances can one error mask another?
- Can fixing a minor error ever expose a more serious error that is currently latent?

Granularity of operation
Responsiveness versus resource usage
- Periodic or event triggered
- Dynamic thresholds for sensitivity to some fault categories?

Pre-determined behaviour or some autonomy?
- Ability to learn?
Main messages

Modern systems are too complex to fully appreciate every possible combination of states at design time:
- simultaneously across an entire system
- behaviour and structural

Need to also support:
- event histories
- non-perfect symmetry in built instances
- post-deployment upgrades

Operating context changes continuously, can lead to changes in:
- priorities
- probability of failure
- outcomes

Architecture is a key issue for self-management in a safety-critical domain
- error modelling and analysis must be facilitated
- responsiveness must be proportionate to the severity of fault

Self-management is an evolving field
- adoption in automotive control could help create / drive standards for self-healing

References

