Unbeatable Pi:

Daniel Almeida
University of Waterloo
Electrical & Computer Engineering
Student ID: 20986346
d2almeid@uwaterloo.ca

I. A Self-Adaptive Video Encoder

The video encoder case study presented tackles familiar challenges and implications of the management of computing systems. That is, due to the increasing complexity of systems and uncertain operating conditions, maintaining optimal performance becomes challenging. Thus, systems may experience degradation in quality. Furthermore, manual management is prone to error, costly and not efficient or responsive to dynamic changes [**?**].

With the goal of establishing a self-managing system that aligns with the Self-CHOP (Self-Configuring, Self-Healing, Self-Optimizing & Self-Protecting) vision, the system will be able to obtain knowledge at runtime to resolve uncertainties, reason about itself and surrounding context in a manner such that it is able to adapt and achieve its realization goals, ultimately ensuring business continuity [**?**].

In order to enhance the video encoder that takes a stream of video frames with self-adaptation capabilities, we aim to address the conflicting goals. That is, we would like to compress the frames such that the video stream fits a given communication channel, but also, while doing so, maintain a required quality of the manipulated frames compared to the original. This comparison would be expressed using the structural similarity index measure (SSIM) [**?**], [**?**].

In the subsequent subsections $A - E$, we identify the basic concepts of the proposed self-adaptive system:

A. Adaptation Goals

*1) **Identification**:* The main goal is to compress video frames such that they meet specific size constraints, thus adhering to the communication bandwidth. The secondary objective is to maintain a structural similarity index above a defined threshold to ensure frame quality between the manipulated frames and the original frames [**?**]. Together, these form the global goal that the adaptation strategy aims to achieve [**?**].

*2) **Responsibilities**:* Through the aforementioned adaptation goals, it is possible to establish the desired outcomes that would guide the feedback loop planning and execution. It would be necessary to find balance between the conflicting objectives, in order to meet the objective of small encoded size and high quality [**?**].

B. Managed Element

*1) **Identification**:* The managed element (or system) is the component responsible for providing the primary application functions. Thus, the video encoder itself, which simulates the recording and manipulation of a video using an mp4 stream, processing each original frame to obtain a compressed version of the stream, acts as the managed system [**?**].

*2) **Responsibilities**:* The managed system's goal is to handle compressing the video so that each frame occupies a specified size and obtaining a SSIM for comparison to the original frames. It achieves this by adjusting parameters such as:

Compression rate control: Influencing size and image quality. The compression factor has a value between 1 and 100, where 100 preserves all frame details and 1 produces highest compression [**?**]. The encoder measures each frame size after its processing.

Sharpening filter: With a value between 0 and 5, where 0 indicates no sharpening and 5 maximum sharpening.

Size of noise reduction: This noise reduction filter shares similar parameter values to that of the sharpening filter. [**?**]

Together, the sharpening filter and noise reduction filter modify certain pixels for post-compression artefact removal; enhancing details and improving perceived quality, but at the cost of impacting compression. [**?**].

C. Autonomic Element and Managing System

An autonomic element represents the fundamental block that enables self-management within the system. In the case of our video encoder (the managed element), the autonomic element would consist of the autonomic manager operating under the MAPE-K (Monitor, Analyze, Plan, Execute and Knowledge) model that will be responsible for the self-management capabilities. However, for this case study, we may interchangeably identify the autonomic manager as the managing system moving forward.

The managing system will consist of the following components:

Monitor: Responsible for gathering the frame/file size (in kilobytes) and metrics necessary for the SSIM after each encoding cycle. The SSIM is a unit-less metric ranging from 0 to 1, quantifying similarity [**?**]. SSIM calculation is done using the $ssim.py$ script. According the the script, the calculation is as follows:

$$\text{SSIM}(x,y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)}$$

where μ_x and μ_y are the local means, σ_x^2 and σ_y^2 are the local variances, and σ_{xy} is the local covariance of the two images. The constants C_1 and C_2 are defined as:

$$C_1 = (K_1L)^2, \quad C_2 = (K_2L)^2$$

with $K_1 = 0.01$, $K_2 = 0.03$, and $L = 255$ representing the dynamic range of pixel values.

Both frame size and SSIM may be monitored by the $encode.py$ script using the $compute\_ssim(img\_in, img\_out)$ and $os.path.getsize(img\_out)$ functions in an iterative loop alongside the encoding process [**?**].

Analyze: Makes use of an evaluation metric to determine whether adaptation goals are being met and output the adaptation signal if necessary. This could be achieved by incorporating a utility function, which would combine the multiple metrics into a single scalar value representing the system's utility.

Borrowing from Maggio et al. [**?**], we place emphasis on hitting the target file size over SSIM. Additionally, in terms of actuators, the preferred choice is said to be quality, followed by sharpening and noise. Results provided by Maggio et al. also suggest that expected SSIM values will range from 0.7 to 0.9 such that the file size does not exceed 15000. It was found that it is possible to achieve values of 0.9 for SSIM and 15000 for file size with the appropriate parameter settings for quality, sharpening and noise.

With this in mind, we define the following:

TABLE I
WEIGHTS AND THRESHOLDS FOR UTILITY FUNCTION (EXCELLENT = E, ACCEPTABLE = A)

Metric	Weight (w_i)	E	A
File Size (% deviation)	0.6	≤5%	≤10%
SSIM	0.4	≥0.9	≥0.8

where TFS represents the Target File Size. More emphasis is placed on File Size with the selection of 0.6 as the associated weight to start with. Through experimentation, it could be the case that an even higher weight could be placed on file size without sacrificing SSIM too significantly.

To calculate the utility component u_i for each metric, we use the following conditions:

$$u_{\text{size}} = \begin{cases} 1.0 & \text{if } \frac{|\text{TFS}-\text{CFS}|}{\text{TFS}} \leq 0.05 \\ 0.5 & \text{if } 0.05 < \frac{|\text{TFS}-\text{CFS}|}{\text{TFS}} \leq 0.10 \\ 0.0 & \text{if } \frac{|\text{TFS}-\text{CFS}|}{\text{TFS}} > 0.10 \end{cases} \quad (1)$$

$$u_{\text{SSIM}} = \begin{cases} 1.0 & \text{if CSSIM} \geq 0.9 \\ 0.5 & \text{if } 0.8 \leq \text{CSSIM} < 0.9 \\ 0.0 & \text{if CSSIM} < 0.8 \end{cases} \quad (2)$$

where CFS is Current File Size and $CSSIM$ is Current SSIM.

The utility score would then be calculated using Equation **??**:

$$U = w_{\text{size}} \cdot u_{\text{size}} + w_{\text{SSIM}} \cdot u_{\text{SSIM}} \quad (3)$$

Plan: Responsible for determining the response strategy based on the analysis output. This could be achieved by adjusting the quality parameters, or adjusting the sharpening and/or noise filter size vales. This would in turn affect the file size as well as the SSIM. When the utility score U is found to be low, the plan component identifies which metric (File Size or SSIM) is causing the deviation from desired performance and adjusts the aforementioned parameters. Based on findings from Maggio et al. [**?**]:

- **Compression Quality** directly influences both File Size and SSIM. If the utility score is low due to File Size exceeding the target, the Plan component reduces the compression quality to decrease the file size. If SSIM is below the acceptable threshold, the Plan component increases the compression quality to preserve more details. However, this adjustment might also cause file size to increase, so it is balanced carefully. This balance is achieved by further suggestions by Maggio et al., where we assign weights to the actuators as well. A secondary cost function could be used to assign higher weight (higher penalty) to specific parameters. The weights that mirror Maggio et al. could be defined as:

  TABLE II
  WEIGHTS FOR ACTUATORS AND PARAMETER RANGES

Actuator	Weight (d_i)	Primary Influence
Compression Quality	100	File Size, SSIM
Sharpening Filter	10^5	SSIM
Noise Reduction Filter	10^5	SSIM

  A function to determine the utility score of each actuator could then be designed in a similar fashion to equations

?? and **??**.

- **Sharpening Filter** primarily affects SSIM by enhancing details in the compressed image. If SSIM is below target but file size is within the acceptable range, the Plan component increases the sharpening filter value. If SSIM is within range but file size is close to or above the target, the Plan component may decrease the sharpening filter to avoid file size increase.
- The **Noise Reduction Filter**, like sharpening, also primarily impacts SSIM by reducing compression artefacts. If SSIM is below target and file size is not an issue, the Plan component increases the noise reduction filter, improving SSIM. If SSIM meets the threshold but file size is not optimal, the Plan lowers noise reduction filter to save file size.

Execute: Implements the decisions made by the planning component as described above in attempts to achieve the desired adaptation goals, which according to Maggio et al., this could be done using the command line.

D. Environment

*1) **Identification**:* The video streams are assumed to be sent over a network. The implications of this shared network means the network bandwidth is prone to demand fluctuations, thus a scarce resource and therefore requiring the video encoder to achieve predictability in the amount of information streamed for every frame [**?**].

*2) **Responsibility**:* The environment provides the constraints and conditions that the encoder should adapt to. That is, the bandwidth limits and conditions. Changes in said conditions would be responsible for triggering adaptation by working in conjunction with the managing system.

E. Feedback Loop

The feedback loop is used by the self-adaptive system (SAS) to convert an open-loop system to a closed-loop system using feedback, achieving adaptivity [**?**]. As described by **??**, we summarize the full feedback loop as follows:

Execution begins at the **Monitor** component; collecting real-time data on the current file size and SSIM values after each encoding cycle. These values are interpreted by the **Analyze** component using a utility function (see Equation **??**) to determine if the adaptation goals are being met. If the utility score U falls below acceptable thresholds, an adaptation signal is sent to the **Plan** component to adjust actuator parameters based on the primary source of deviation—whether file size or SSIM is out of range.

The **Plan** component adopts an adaptation strategy by adjusting compression quality, sharpening, and noise filter settings. [**?**]. Finally, the **Execute** component applies these adjustments to the encoder, influencing the encoding of subsequent frames.

We note that with this self-adaptive compression approach, the encoder starts with potentially suboptimal settings and progressively fine tunes itself in response to the data. Each encoding cycle ultimately refines the balance between file size and SSIM, adapting to content characteristics and bandwidth constraints. With caching, we could take advantage of learned parameters for faster encoding on similar content. However, due to the priori unknown problem, this may not always be applicable. By introducing a quick pre-scan of the video, similar content could be identified, but this introduces additional latency overall.

Alternatively, one could opt for the use of Pareto optimization over a utility function as it is a multi-objective optimization that may be better suited for the conflicting goals. And this could be achieved without combining them into a single utility score. The optimal solution is known as the Pareto front - when no other solution can improve one objective without worsening at least one other. However, although it could be a more flexible and comprehensive approach, the added latency from the increased computational requirements of Multi-objective optimization algorithms could be significant, especially given the real-time nature of the encoder.

F. Conceptual Model

This final subsection presents a detailed conceptual model of the proposed adaptation solution above, now illustrated by Figure **??**:

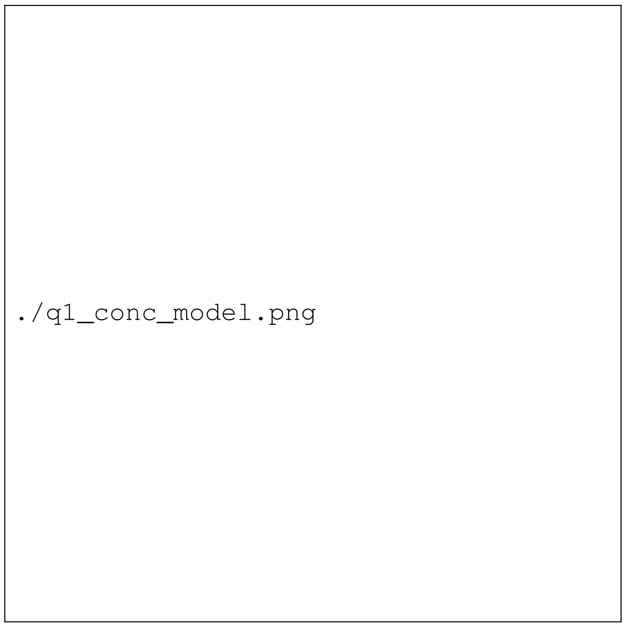

Fig. 1. Conceptual Model illustrating solution to adaptation problem

II. Architecture-Based Adaptation

The adaptive video compression solution of section **??** focused on a single system implementing the MAPE-

K loop targeting specific, predefined parameters and a specific application. The approach was tightly coupled to the video encoder, aiming to balance the conflicting adaptation goals and operating locally without the need for a larger architectural framework. However, this tight coupling approach foreshadows the necessity to construct a specific software configuration for each new application, which would be wasteful [**?**]. To tackle this problem, Architecture-Based Adaptation (ABA) is introduced. While software-based adaptation adjusted parameters within a fixed architecture, ABA is an approach where the system's architecture is explicitly modelled and manipulated. It takes a multi-layered structure approach that places more emphasis on separation of concerns, abstraction and system-wide reasoning, making it more modular and able to support complex and distributed adaptation. That is, the ability to adapt across different components or layers in a distributed manner, where components can make independent local or coordinated global adjustments. A more flexible framework, responding to environmental changes at different layers, supporting both parameter adjustments and dynamic component and/or service substitutions. Ultimately, moving away from the redundancies of software-based adaptation when attempting to cover every unique application, such as extensive coding, testing and maintenance for each use case.

The three layer model for SAS is one such architecture; inspired by the classic architecture of robotic systems, which consists of a controller (reactive feedback control mechanism, a sequencer (reactive plan execution mechanism) and a deliberator/planner (mechanism for performing time-consuming deliberative computations) [**?**]. The three layers are as follows:

Goal Management (GM): Focuses on high-level goal management of the system by operating with a MAPE-K workflow centered on defining and adjusting overarching goals [**?**]. GM translates said goals into ”Change Plans” to fulfill requests from the layer below.

Change Management (CM): Responsible for the creation of actionable plans based on the high-level goals provided by GM. It operates with a MAPE-K workflow, but with a focus on planning and execution of specific changes. It may request plans from GM using ”Plan Request” based on goals, which are then fed to the layer below via actionable ”Change Actions”.

Component Control (CC): The third layer manages the system's interconnected operational components, containing internal mechanisms to adjust system behaviour. It receives ”Change Actions” from the CM layer and executes them. Component status' are monitored by the CM layer.

To further our understanding, we apply the concept of ABA in studying a component-based planning platform named ”MUSIC” [**?**], which optimizes the overall utility of applications that are subject to modification in operating conditions. The platform allows interchangeable components and services to be automatically plugged when the execution context changes [**?**]. This is particularly useful in the domain of ubiquitous computing due to the unexpected changes of execution context [**?**]. In studying the platform, we present the following:

A. *Three-layer model mapping*

Figure **??** maps the MUSIC platform's architecture to the three-layer model for self adaptation.

./q2_three_layer_model.png

Fig. 2. MUSIC platform three-layer model

The GM layer consists of the Adaptation Reasoner, Plan Repository, Quality of Service (QoS) Manager, Service Discovery and Service Level Agreement (SLA) Negotiation.

The CM layer is made up of the Context Manager, Configuration Executor, Adaptation Controller, Remoting Service, Binder and Factory.

The CC layer includes the Application Components and Service Proxies. It would also contain any other components providing services.

As for the environment(s), Execution Context, Service Landscape, User Context and Environmental Conditions are present.

B. *Elements and Responsibilities*

We now present more thoroughly the elements of of each layer in Figure **??** based on the descriptions of Rouvoy et. al [**?**]:

- The **Context Manager** is responsible for monitoring execution context changes and triggering the planning [**?**].
- The **Adaptation Controller** is responsible for coordinating the adaptation process once it is triggered by the context change [**?**].
- The **Adaptation Reasoner** executes planning heuristics driven by metadata included in plans to generate valid application configurations. It discards any configurations whose dependencies are unresolved. Heuristics then rank configurations based on the computed predicted properties. Values for such computations are retrieved from the QoS Manager [**?**].
- The **Plan Repository** provides the $IPlanResolver$ interface to the Adaptation Reasoner for recursive retrieval of plans. These plans represent alternative implementations for components and services. Plans are discarded automatically whenever remote services disappear or become unavailable [**?**].
- The **Configuration Executor** handles the reconfiguration process, which makes use of the set of plans selected by the planner to reconfigure the application. This occurs in collaboration with the component(s), which provide reconfiguration interface(s) to allow middleware to safely replace/transfer their state to alternative component(s) [**?**].
- The **QoS Manager** hosts the QoS property values to be used by the Adaptation Reasoner to evaluate the utility of different configurations using the utility function [**?**].
- **Service Discovery** publishes and discovers services using different discovery protocols. Service descriptions defining functionality and containing relevant information for service access can also be published by the Discovery Service [**?**].
- The **Remoting Service** exports services at the service provider side, and binds services at the service consumer side. And communication are handled through service proxies and skeletons in collaboration with the Binder and Factory elements that provide the Binding Framework and Proxy Factories [**?**].
- The **Binder and Factory** provide the Binding Framework and dedicated proxy factories [**?**].
- **SLA Negotiation** handles negotiation of SLAs with service providers, ensuring QoS requirements are met [**?**].
- **SLA Monitoring** is enabled on service proxies to ensure compliance of SLAs during execution. The MUSIC middleware can check the current state of the agreement during execution. If violations are detected, termination occurs and a new adaptation process is triggered if needed [**?**].
- **Service Proxies** The MUSIC platform may instantiate service proxies to act as local representatives of remote services. They encapsulate the communication protocols necessary to access said remote services in a location-transparent way [**?**].
- **Application Components and Components Providing Services** Components with core functionality of application, which may require or provide services to other components. They may also offer services to other applications to external consumers outside of the adaptation domain [**?**].
- The **Execution Context** in the environment refers to resources such as CPU, memory, battery, network bandwidth and connectivity. These resources are subject to fluctuations in ubiquitous computing environments where the device may move and experience varying resource availability [**?**].
- The **Service Landscape** in the environment refers to the set of available services, both internal within the adaptation domain and external provided by third parties. Availability, performance, QoS of said services may fluctuate and trigger adaptation [**?**].
- The **User Context** would include preferences, profiles and activities that may influence utility functions, thus adaptation decisions. If a user prefers higher quality video, this could affect battery consumption for example [**?**].
- The **Environmental Conditions** are other external factors like location, time of day, sensor data, that may affect application behaviour or requirements [**?**].

C. *Flow of activities across layers*

Activities of a MUSIC platform across the three layers would be as follows:

In the GM layer, services are made discoverable and accessible through discovery protocols. For a given instance of a MUSIC platform, the discovery service would make use of said protocols to retrieve service descriptions from available services in the environment, which include the service capabilities, semantics and QoS in the form of an agreement template. These are then converted to a service plan by Discovery Listeners, representing an alternative realization, and stored in the Plan Repository [**?**].

In the CM layer, context monitoring and adaptation triggering occur. The Context Manager is continuously monitoring execution context for changes, such as resource availability, connectivity or service performance. Both parties (service consumer and service provider) are responsible for SLA Monitoring and the status of the agreement is checked routinely to ensure that no violations are occurring. If at any give time a violation is found during monitoring, the MUSIC middleware will terminate the service, and trigger a new adaptation process to find the best replacement candidate if necessary. It is the Context Manager who notifies the Adaptation Controller of such event which then triggers the

Adaptation Reasoner [?].

Back in the GM layer again, during the planning phase, the Adaptation Reasoner retreives alternative plans from the Plan Repository using the $IPlanResolver$ interface. Through heuristics, it evaluates and ranks the possible configurations in order to devise the plan which would optimize the utility of the running application. For this, the Adaptation Reasoner works in conjunction with the QoS Manager which provides the values for the utility calculation. It may also engage in SLA Negotiation to negotiate QoS properties with service providers if dynamic properties exist [?].

In the CM layer once more, once a plan is selected, the configuration is passed to the Configuration executor. The executor will now be responsible for reconfiguring the application by deploying, replacing, or removing components in the CC layer. If remote services are included, then the Configuration Executor evokes the Remoting Service to instantiate service proxies. These proxies serve as local representatives of the remote service, providing access to the service in a location-transparent way. The Remoting Service uses the binding framework and Proxy Factories to create the appropriate service proxies supporting various communication protocols. During binding, the SLA contract is provisioned, that is, computing resources are reserved alongside SLA monitoring facilities, allowing the contract to be enforced by parties involved [?].

To handle disconnections, the service proxy implements disconnection detection algorithm inspired by ambient programming principles. When the remote connection to the service is lost, the proxy stores incoming service requests in a queue, and returns non-blocking future object to the application that includes actions to be triggered when the connection is restored. If the connection is not restored within acceptable time, the agreement is terminated, removing the plan from the Plan Repository and triggering a new adaptation cycle to find a replacement. The request queue is transferred to the new component or service proxy selected by the middleware [?].

The application now continues to operated under the updated configuration. And SLA Monitoring continues to check for SLA compliance, monitoring the status of agreements. On the next SLA violation, the SLA Monitoring will notify the Context Manager once more, potentially triggering another adaptation cycle.

On the other hand, if a service is being provided - that is, the MUSIC platform is now playing the role of a service provider - behaviours in the GM and CM layers differ slightly. The MUSIC platform publishes service descriptions and agreement templates via the Discovery Service. SLA Negotiation component handles any incoming agreement offers from service consumers, and determines whether to accept or reject these offers based on resource availability and their impact on existing agreements which are currently being monitored. Agreements which are accepted are tracked, and the Remoting Service creates service skeletons to interface and handle communication protocols. Skeletons are also instrumented with SLA Monitoring and violations are treated in a similar manner to service consumption within MUSIC [?].

III. Runtime Models

Autonomic Computing (AC) aims for a system that is capable of evolving without human intervention. It should be able to handle installs, configurations and component maintenance, all at runtime. However, many AC solutions result in systems with high complexity. With runtime models, we may manage the complexity of concrete designs of SAS [?]. Runtime models operate at the model level, which decouples the adaptation logic from the underlying system, introducing a new level of modularity and scalability. To explore this approach further, we turn to the ”MoRE” (Model-Based Reconfiguration Engine); a platform for self-adaptation based on the principles of dynamic software product lines. It has been applied to a smart home case with a focus on self-healing and self-configuring. The proposed approach ”reuses variability models at runtime to provide richer semantic base for decision making”, and claims that autonomic behaviour can be achieved by leveraging variability models at runtime [?], [?].

A. *Core Concepts and Element Responsibilities*

The two core concepts of the MoRE platform are that:

- It focuses on the reuse of design knowledge to achieve AC (variability models) using feature modelling. The feature models represent the possible configuration variants in terms of features and whether they are optional, mandatory, single-choice or multiple-choice. Using these models at runtime, the system can adaptively activate or deactivate features when responding to context changes [?].

  and

- It focuses on the reuse of existing model-management technologies at runtime (dynamic product-line architecture) using the XML Metadata Interchange (XMI) standard. This makes it possible to apply the same technologies used at design time to manipulate models at runtime reducing complexity [?].

More thoroughly, the responsibilities of the different elements are described by Cetina et. al as follows:

- **Domain-Specific Language (DSL) models:** Define the system’s architecture in terms of services and devices and their communication channels. In this case study specifically, an example is PervML for smart homes [?].

- **Model Operations:** Two operations have been defined to determine architecture increment and decrement calculations: Architecture reIncrement ($A\Delta$) and ArchitectureDecrement ($A\nabla$). They take the resolution as input and calculate modifications in terms of components and channels that would need to be modified [**?**].
- **Feature Models:** Model the possible features and configurations of the application, as well as the variability relationships. It specifies the system in a coarse-grained fashion [**?**].
- **Context Monitor:** Uses runtime state as input to check context conditions. When conditions are met, MoRE proceeds to query the runtime models for necessary architectural modifications based on the resolution and previous model operations [**?**].
- **Reconfiguration Engine (MoRE):** Acts on context changes by using model query responses to generate reconfiguration plans which contain a set of reconfiguration actions to modify system architecture [**?**].
- **Condition:** Represents the specific context condition or event as seen by the Context Monitor. Used to trigger adaptations in the system in combination with resolutions [**?**].
- **Resolution:** Represents the set of changes a condition triggers. It is a list of feature-feature state pairs. Each resolution is associated with a context condition, representing the change in terms of activation or deactivation of features that occur when the condition is met [**?**].
- **Reconfiguration Actions:** These actions modify the system architecture. **Component Actions** describe a component's transition from active to idle in order to perform adaptation. **Channel Actions** establish or break communication with other services/components (called bindings). Lastly, the **Model Actions** updates the feature model according to the new system functionality using model introspection, which is the ability for the model to inspect and modify its own structure and behaviour at runtime [**?**].

B. *Identifying the Principal Strategy*

The three principal strategies discussed in class were:

- **Strategy #1:** MAPE components share a common set of runtime. This is known as the Graph-based Runtime Adaptation Framework (GRAF).
- **Strategy #2:** MAPE components exchange runtime models. This is known as Dynamic Variability in complex Adaptive Systems (DiVA).
- **Strategy #3:** MAPE models share a common set of runtime models. This is known as ExecUtable RuntimE MegAmodels (EUREMA).

In MoRE, the MAPE components all share a common set of runtime models, in this case, the feature models. The feature models are used by all MAPE components: The Context Monitor (Monitor) uses the models to trigger adaptations based on context changes. The MoRE engine (Analyze and Plan) queries feature models to determine necessary adaptations and generate the reconfiguration plan. The Reconfiguration Engine (Execute) executes reconfiguration actions that modify system architecture. The feature models are updated to reflect the new configuration.

It is thus clear that, given the **reasoning** above, the MoRE platform aligns with **Strategy #1**.

a) Advantage to Strategy #1: It takes a model-centric approach; leverage runtime models for adaptation with clear representation and management of system state and behaviour. This also provides a structured way to query, transform and interpret the shared runtime models in a consistent manner, reducing inconsistencies and improving reliability. [**?**].

b) Disadvantage to Strategy #1: Depending on approach, system adaptation could fall out of variability scope for some specific requirements and struggle with Self-Adapting Scenarios. And it would not be economically realistic to build individual features to suit each user [**?**].

C. *Adaptation Workflow*

To conclude, Figure **??** illustrates the adaptation workflow for a concrete self-healing scenario with a detailed diagram:

./MORE_workflow.png

Fig. 3. Adaptation workflow for self-healing scenario

Cetina et al. presented a self-healing example scenario for a smart home that provides autonomic capabilities. When an alarm fails, the smart home can adapt by making the home lights blink as a replacement for the failed alarm [**?**]. Similarly, as illustrated by the concrete example of Figure

??, if a motion sensor responsible for detecting occupancy in a room were to fail, the smart home could adapt by utilizing nearby security cameras with motion detection capabilities instead.

First, Context Monitoring detects that the motion sensor has failed, and the $MotionSensorFailure$ condition is fulfilled. This sends a signal to the MoRE engine, which proceeds to retrieve the associated Resolution. Based on [**?**], the Resolution could look like Equation **??**:

$$\begin{aligned} R\_\text{MotionSensorFailure} = \{&(\text{MotionSensorFeature}, \text{Inactive}), \\ &(\text{SecurityCameraMotionDetectionFeature}, \text{Active})\} \end{aligned} \tag{4}$$

where $MotionSensorFeature$ is the functionality provided by the motion sensor, and $SecurityCameraMotionDetectionFeature$ is the security cameras' motion detection.

Next, MoRE queries the Feature Model to determine communication channels for each feature, as well as which components need to be deactivated and which ones needs to be activated or reconfigured. The Architecture modifications are calculated using the Reconfiguration Actions. MoRE then generates the Reconfiguration Plan. This will involve the component actions, channel actions as well ad the model action which will update the Feature Model to the new configuration. And then finally, MoRE proceeds to execute the actions outlined in the plan.

References

[1]