1st International Workshop on Control Theory for Software Engineering (CTSE 2015)

Proceedings

Antonio Filieri and Martina Maggio

August 31, 2015
Bergamo, Italy

The Association for Computing Machinery, Inc.
2 Penn Plaza, Suite 701
New York, NY 10121-0701

Notice to Past Authors of ACM-Published Articles

ACM intends to create a complete electronic archive of all articles and/or other material previously published by ACM. If you have written a work that was previously published by ACM in any journal or conference proceedings prior to 1978, or any SIG Newsletter at any time, and you do NOT want this work to appear in the ACM Digital Library, please inform permissions@acm.org, stating the title of the work, the author(s), and where and when published.

ACM ISBN: 978-1-4503-3814-1

Additional copies may be ordered prepaid from:

ACM Order Department
P.O. BOX 11405
Church Street Station
New York, NY 10286-1405

Phone: 1-800-342-6626
(U.S.A. and Canada)
+1-212-626-0500
(All other countries)
Fax: +1-212-944-1318
E-mail: acmhelp@acm.org

Production: Conference Publishing Consulting
D-94034 Passau, Germany, info@conference-publishing.com

Message from the Chairs

The Software Engineering community is pushing a significant effort on self-adaptive systems. Such systems are required to modify their behavior to maintain goals in response to unpredicted changes in their execution environment. Key challenges for self-adaptive systems include time-efficient diagnosis of requirements violation, fast decision making, and systematic procedures to assess their effectiveness and dependability. Further challenges arise in correlating local and global decision-making for larger-scale or distributed systems. Despite a variety of approaches has been proposed for self-adaptive software, only a few of them can provide formal guarantees about the quality of adaptation, mainly due to the difficulty of grounding the adaptation mechanisms within suitable theoretical frameworks.

Control theory has established effective mechanisms to make controlled plants behave as expected. Although the similarity with software adaptation is self-evident, most of the attempts to apply "off-the-shelf" control theory to software applications have been unsuccessful. The main challenge has been model software systems as dynamical system -- i.e., by means of differential or difference equations -- because of the intrinsic non-linearities, the variety of usage profiles, and the interconnection of heterogeneous components, together with the common lack of control theory skills in the software engineering education, research, and practice. As result, the current use of control theory is limited to very specific applications and hard to generalize to large classes of software.

The aim of this workshop is to provide a forum to discuss a different route. Bringing together researchers from the communities of software engineering and control theory and fostering their debate and cooperation, our goal is twofold. On one hand, exploring new modeling strategies to incorporate control in software systems design and development, empowering software engineers with theoretical and practical skills to bring control to the core of adaptation. On the other hand, outlining the new challenges and opportunities the very nature of software and computing systems place to established control theory, due to its higher decoupling from the physical constraints of an classic plant. For the future, we envision a fruitful hybridization of the two disciplines for engineering adaptive software.

We received a total of 11 submissions, out of which 6 were accepted for presentation. To complement the program, the workshop proposes a keynote speech by Prof Karl-Erik Årzen about his experience with the design of self-adaptive resource managers for embedded systems and an introductory tutorial on control theory for computer scientists by Prof Alberto Leva, to provide a first contact point for the software engineering researchers aiming at knowing more about this discipline.

We would like to thank the program committee members for providing valuable and constructive feedback to the authors under a tight schedule, as well as each author and presented who submitted their work to the CTSE 2015 workshop.

Antonio Filieri and Martina Maggio
(CTSE 2015 Chairs)

CTSE 2015 Organization

Workshop Chairs

Antonio Filieri	University of Stuttgart, Germany
Martina Maggio	Lund University, Sweden

Publicity Chair

Ilias Gerostathopoulos	Charles University, Czech Republic

Program Committee

Javier Camara	Carnegie Mellon University, USA
Nicolas D'Ippolito	University of Buenos Aires, Argentina
David Garlan	Carnegie Mellon University, USA
Carlo Ghezzi	Politecnico di Milano, Italy
Henry Hoffmann	University of Chicago, USA
Samuel Kounev	University of Wuerzburg, Germany
Filip Krikava	University of Lille/INRIA, France
Marta Kwiatkowska	Oxford University, UK
Alberto Leva	Politecnico di Milano, Italy
Marin Litoiu	York University, Canada
Sasa Misailovic	Massachusetts Institute of Technology, USA
Alessandro V. Papadopoulos	Lund University, Sweden
David Parker	University of Birmingham, UK
Anders Robertsson	Lund University, Sweden
Eric Rutten	INRIA Grenoble, France
Sebastian Uchitel	University of Buenos Aires, Argentina
Thomas Vogel	Hasso-Plattner-Institut, Germany
Danny Weyns	Linnaeus University, Sweden

Contents

SimCA vs ActivFORMS: Comparing Control- and Architecture-Based Adaptation on the TAS Exemplar

Stepan Shevtsov
Linnaeus University
Växjö, Sweden
stepan.shevtsov@lnu.se

M. Usman Iftikhar
Linnaeus University
Växjö, Sweden
usman.iftikhar@lnu.se

Danny Weyns
Linnaeus University
Växjö, Sweden
danny.weyns@lnu.se

ABSTRACT

Today customers require software systems to provide particular levels of qualities, while operating under dynamically changing conditions. These requirements can be met with different self-adaptation approaches. Recently, we developed two approaches that are different in nature — control theory-based SimCA and architecture-based ActivFORMS — to endow software systems with self-adaptation, providing guarantees on desired behavior. However, it is unclear which of the two approaches should be used in different adaptation scenarios and how effective they are in comparison to each other. In this paper, we apply SimCA and ActivFORMS to the Tele Assistance System (TAS) exemplar and compare obtained results, demonstrating the difference in achieved qualities and formal guarantees.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures; I.2.8 [**Computing Methodologies**]: Problem Solving, Control Methods, and Search—*Control theory*

Keywords

Adaptation, self-, adaptive system, software, MAPE, control theory, controller, architecture, feedback, SimCA, activforms

1. INTRODUCTION

The burden on software developers has drastically increased in recent years as customers expect software to cope with continuous change. They expect the software to run seamlessly on different platforms, deal with varying resources, and adapt to changes in system goals. Often, these runtime changes are difficult to predict, requiring software engineers to design the software with incomplete knowledge.

Self-adaptation is widely encouraged to handle software design with incomplete knowledge [4, 7]. A classic approach to realize self-adaptation is architecture-based adaptation [15, 13], where a system maintains an explicit architectural model of itself, reasons about this model, and adapts itself to particular adaptation goals when relevant changes occur. However, achieving guarantees of the system behaviour with architecture-based adaptation is hard [3]. For this reason adaptation mechanisms based on control theory [10, 9] attracted the attention of self-adaptive systems community[1].

Recently, we developed two approaches for runtime adaptation, one from each of the mentioned fields: *ActivFORMS* (Active Formal Models for Self-adaptation) [11] an architecture-based approach, and *SimCA* (Simplex Control Adaptation) [16] that is based on principles from control theory and linear optimization. In this paper, we present a comparative evaluation of SimCA and ActivFORMS on a set of scenarios. To the best of our knowledge, no systematic comparison between approaches for runtime adaptation based on control theory and architecture-based adaptation has been performed so far. We compare obtained results of runtime adaptation (the system output), and we analyze the provided guarantees and system behavior in the presence of disturbances.

The evaluation is conducted using the TAS exemplar [17]. TAS provides remote health support to patients by composing a number of services. Each service can be implemented by multiple providers. These implementations have different reliability, performance and cost, which affect the overall quality of the application. Choosing a concrete service provider to ensure the required quality of service (QoS) at runtime is a key factor to adaptation of service-based systems as TAS.

The remainder of the paper is structured as follows. An adaptation scenario with the TAS exemplar is introduced in Section 2. Section 3 summarises the two studied adaptation approaches: SimCA and ActivFORMS. In Section 4 these two approaches are applied to TAS and compared in multiple scenarios. Section 5 provides discussion of received results. Finally, conclusions and directions for future research are presented in Section 6.

2. ADAPTATION SCENARIO: TAS

In this section, we introduce a scenario of TAS used for evaluating SimCA and ActivFORMS. The main goal of TAS is to track a patient's vital parameters in order to adapt the drug or drug doses when needed, and take appropriate actions in case of emergency. To satisfy this goal, TAS combines three types of services in a workflow, shown on Figure 1.

Each incoming request is first processed by the Medical Service. This service receives messages from patients with their

CTSE'15, August 31, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3814-1/15/08...$15.00
http://dx.doi.org/10.1145/2804337.2804338

[1] In this paper by self-adaptive system we mean a system equipped with any kind of adaptation/control mechanism which may or may not be adaptive itself.

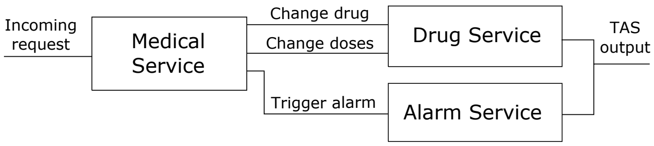

Figure 1: TAS workflow.

vital parameters, analyses the data, and replies with instructions to (1) change the drug or (2) change the drug doses, or (3) invoke an alarm at the First-Aid squad in case of an emergency. When invoked, the Drug Service notifies a local pharmacy to deliver new medication to the patient or change his/her dose of medication. When the Alarm Service is invoked, it dispatches an ambulance to the patient.

For service-based systems such as TAS, the functionality of each service can be implemented by a number of providers that offer services with different quality properties: reliability, performance, and cost. The system design assumes that these properties can be quantified and measured. E.g., reliability is measured as a percentage of service failures, while performance is measured as the service response time. At runtime, it is possible to pick any of the services offered by the providers. The services are considered to be part of the environment because they are not under control of TAS. For example, the failure profile of a concrete service implementation may change at runtime, due to the changing workloads at the provider side or unexpected network failures.

We consider that five service providers offer the Medical Service, three providers offer the Alarm Service and only one provider offers the Drug Service. Table 1 shows example properties of available services based on data from [2].

Table 1: Properties of all services used in TAS.

Service	Name	Fail.rate, %	Resp.time, ms	Cost, ¢
S1	Medical Service 1	0.06	22	9.8
S2	Medical Service 2	0.1	27	8.9
S3	Medical Service 3	0.15	31	9.3
S4	Medical Service 4	0.25	29	7.3
S5	Medical Service 5	0.05	20	11.9
AS1	Alarm Service 1	0.3	11	4.1
AS2	Alarm Service 2	0.4	9	2.5
AS3	Alarm Service 3	0.08	3	6.8
D	Drug Service	0.12	1	0.1
	Requirements (Goal)	**0.03**	**26**	**min**

The system requirements are the following:

R1. The average failure rate should not exceed 0.03 %*

R2. The average response time should not exceed 26 ms

R3. Subject to R1 and R2, the cost should be minimized.

The requirements R1-R3 as well as the properties of the services may change at runtime and the system should adapt accordingly. The adaptation task is to decide, for each incoming messages with a patient's vital parameters, which combination of services to select in order to continuously satisfy the three requirements.

2.1 The Adaptation Problem

Generalizing from the concrete TAS scenario, the adaptation problem we are aiming to solve is the following:

> *Maintain a desired level of quality for multiple goals and optimize the solution according to another goal, regardless of possible fluctuations in the system parameters, measurement accuracies, requirement changes, and dynamics in the environment that are difficult to predict.*

Defining and developing such an adaptive solution introduces several key challenges. First, the appropriate sensors (measured variables) and actuators (knobs that can influence the software behavior) must be carefully chosen. Second, the software system must be properly modeled. Finally, the appropriate adaptation mechanism that achieves the goals, rejecting external disturbances and optimising the solution according to additional goal, must be developed. The following section describes two approaches that tackle these challenges.

3. STUDIED APPROACHES

3.1 SimCA

The adaptation logic of SimCA consists of multiple SISO controllers that independently compute control signals for each of the goals, and a simplex block that receives the control signals as input and produces an output that is used for adapting the software system. Figure 2 schematically shows the primary building blocks of SimCA.

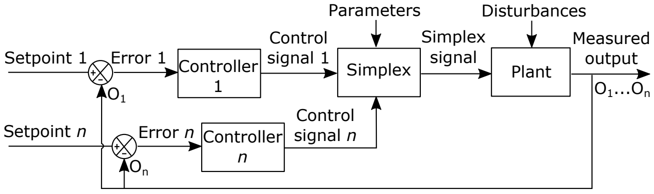

Figure 2: A self-adaptive software with SimCA.

A detailed explanation of SimCA is available in [16]. In short, each system goal (reliability, performance) except cost is represented as a setpoint $s_i(k)$. On every adaptation step k the system outputs $O_i(k)$ are measured. Based on the error $e_i(k) = s_i(k) - O_i(k)$ and a linear model $\mathcal{M}_i$ described below, each controller C_i produces a signal $u_i(k)$ which represent the value of each goal that should be reached by the system. The simplex block receives $u_i(k)$ and additional environment/plant parameters (e.g., the invocation cost of external services) as inputs and produces a simplex signal $u_{sx}(k)$ as output. $u_{sx}(k)$ contains the values of system knobs which affect the plant behaviour. For TAS, $u_{sx}(k)$ is a vector containing the probabilities to select each of the available service providers. Disturbances affecting $O_i(k)$ are handled by controller C_i via adjusting the control signal $u_i(k)$.

SimCA works in three phases: identification, control, and optimization.

In the *identification* phase, n linear models of the controlled system are built. Each model $\mathcal{M}_i$, $i \in [1, n]$, is responsible for one goal s_i. The identification phase starts by feeding all possible control signal values to the plant. The goal of identification is to determine the influence of control signal u_i on the corresponding system output O_i at every time step

*The system design assumes that in case of a failure the request is not dropped and can be send to the same or another service provider for re-execution. Hence, the goal failure rate is lower than the rates of individual implementations.

k. The dependency between u_i and O_i is captured by the coefficient α_i which is further used to build controller i. α_i is calculated during identification based on linear regression using the APRE tool [14]. As a result, the following set of linear models is obtained:

$$O_i(k) = \alpha_i \times u_i(k-1) \qquad (\mathcal{M}_i)$$

In $\mathcal{M}_i$ the control signal is a value from the interval $[min_i, max_i]$, where max_i and min_i are the maximal and minimal values that can be achieved by TAS for the i-th goal:

$$u_i(k) = (max_i - min_i) \times \eta_i(k) + min_i \qquad (1)$$

η_i is the control coefficient. During the identification phase, η_i changes from 0.0 to 1.0 using an increment of 0.05 on every step k. As a consequence, most feasible values of goal i are produced as u_i. This allows us to measure all the possible values of the outputs $O_i(k)$, calculate α_i, and build $\mathcal{M}_i$.

The model $\mathcal{M}_i$ generally describes the system behavior but does not take into account small disturbances or sudden failures that typically occur in software systems. To deal with inaccuracies in $\mathcal{M}_i$, SimCA uses a Kalman filter to adapt the model at runtime, and a change point detection mechanism that allows reacting to critical changes in the system.

An important note concerning the control methodology of SimCA is that the simplex method does not change the value of control signal i. Instead, simplex is responsible for seamless translation of control signals into a proper actuation signal. For example in TAS, if the control signal equals to failure rate = 0.03, simplex finds a combination of services (S1-S9) that assures this failure rate is not exceeded. Hence, simplex is not considered when building a system model and synthesizing controllers that manage this model. Instead, controllers are assumed to affect the plant output via control signals. Every goal is controlled separately during the first two phases of SimCA. This means that during identification and control phases SimCA works in parallel with multiple Single-Input-Single-Output (SISO) controllers, and then combines control outputs with the help of simplex during optimization phase.

In the *control* phase, a set of n controllers is synthesized. Each controller C_i is responsible for the i-th goal. C_i calculates the control coefficient η_i at the current time step k depending on the previous value of control coefficient $\eta_i(k-1)$, adjustment coefficient α_i, controller pole p_i and the error e_i:

$$\eta_i(k) = \eta_i(k-1) + \frac{1-p_i}{\alpha_i} \times e_i(k) \qquad (2)$$

The controller pole p_i belongs to the open interval $(0, 1)$ to maintain stability and avoid oscillations. The pole also allows to trade-off robustness to external disturbances with the convergence speed (known as settling time): higher values of p_i lead to slower convergence [8].

In the *optimization* phase, SimCA combines the signals $u_i(k)$ from multiple controllers using the simplex method to optimally drive the measured output of the system towards its desired behavior. Simplex solves the following problem:

$$\text{Minimize Cost:}\, min\, C = \sum_{j=1}^{p} c_j \cdot x_j$$

subject to:

$$\begin{cases} \sum_{i=1}^{n} \sum_{j=1}^{p} a_{ij} \cdot x_j = u_i \\ x_j \geq 0,\, u_i \geq 0,\, with\ i = 1 \ldots n,\, j = 1 \ldots p \end{cases} \qquad (3)$$

where:

- p: a number of variables — each variable represents one service provider;
- n: a number of equations — each equation represents a goal to be satisfied;
- x_j are the values of system knobs, i.e. probabilities to select each of the service providers;
- u_i are the control signals;
- a_{ij} and c_j are the monitored parameters: the failure rate, response time and cost of external services.

SimCA finds the variables of the problem x_j and passes them to the plant in the form of a simplex signal. A detailed explanation on how simplex solves the system of equations (3) can be found in [16], and additional background in [5].

3.2 ActivFORMS

ActivFORMS is an architecture-based approach for self-adaptation that uses an integrated formal model of the adaptation components of the feedback loop and the knowledge they share [11]. ActivFORMS distinguishes itself from existing architecture-based approaches in three ways:

- The formally verified model of the feedback loop is directly executed by the virtual machine, hence called the active model. This allows to guarantee at runtime the system properties verified at design time. As the active model is directly executed, the approach does not require coding.
- ActivFORMS supports dynamic changes of the active model. A new feedback loop model can be deployed at runtime to meet new or changing goals.
- ActivFORMS supports goal management and model verification at runtime.

ActivFORMS follows a three-layered reference model proposed by Kramer and Magee [13], see Figure 3. The bottom layer comprises the managed system[2] that implements the domain-specific functionality. The active model monitors and adapts the managed system through probes and effectors respectively[3]. The goal management layer monitors the active model and environment, and deals with adaptation issues that cannot be handled by the current active model; e.g., it dynamically changes the active model to deal with changing system goals.

The active model realizes a MAPE-K (Monitor-Analyze-Plan-Execute-Knowledge) feedback loop [12] that monitors the managed system and adapts it according to the system goals. ActivFORMS supports feedback loops modeled using networks of timed automata. A timed automaton is a finite-state machine that models a behavior, extended with clock variables.

Recalling the TAS scenario, we used a utility function as a part of the planner component of a MAPE-K formal model to provide the required adaptation, see Figure 4. When triggered by a signal from the analysis component, the planner calculates the utility function coefficients and sends a signal to the execution component to update the managed system.

[2] Managed system is called plant in control theory.

[3] Probe corresponds to sensor in control theory terminology, effector corresponds to actuator.

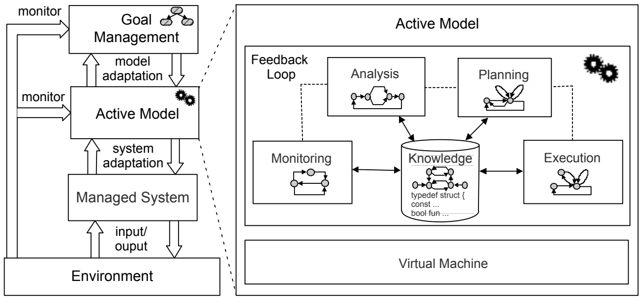

Figure 3: The ActivFORMS approach

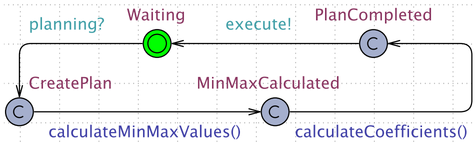

Figure 4: Planning automaton

The utility function works as follows: all system goals (reliability, performance) and optimization goal (cost) are represented as G_i. During the first adaptation period the cost goal is set to a minimum achievable value: min_{cost}. On every adaptation step k the system outputs $M_i(k)$ are measured. Knowing the error $e_i(k) = G_i - M_i(k)$, we calculate how big it is in relation to the actual values of the goal i:

$$\lambda_i(k) = \frac{e_i(k)}{max_i - min_i} \tag{4}$$

For example, if the measured response time $M_{RT}(k)$ is 29.04 on a first adaptation step and other service parameters are equal to the values from Table 1, we get:

$$\lambda_{RT}(1) = \frac{29.04 - 26}{35.4 - 21.68} = 0.22$$

Then we calculate coefficients $cf_i(k)$ for the utility function. $cf_i(k)$ represents the selection probability for a QoS strategy in the next adaptation period:

$$cf_i(k) = \frac{\lambda_i(k)}{\sum_{j=1}^{n} \lambda_j(k)} \tag{5}$$

Using the same example, we get:

$$cf_{RT}(1) = \frac{0.22}{0.30 + 0.22 + 0} = 0.43$$

In the provided example, a coefficient for performance of $cf_{RT} = 0.43$ means that in the next adaptation period the service provider with the lowest difference between G_{RT} and service response time will be chosen 43% of the times.

For offline model-checking we use Uppaal [1], a tool that supports modeling of behaviors and verification of properties of networks of timed automata. For the specification of properties, we use Timed Computation Tree Logic (TCTL). TCTL allows checking individual states of the system state space as well as traces over the state space. The latter allows to verify reachability, safety, and liveness properties.

The ActivFORMS virtual machine can perform the following functions: execute the formal model according to the semantics of timed automata, interact with the managed system and environment through probes and effectors, support online verification of the active model, and update the active model when requested. ActivFORMS provides a set of formal templates to design the MAPE-K elements [6], and abstract Java classes to implement probes and effectors. Probes track the managed system and the environment and transfer data to the Monitor automata of the feedback loop, while Effectors transfer actions generated by the Execution automata to the managed system.

Goal Management comprises a tree-based goal model where nodes have associated MAPE-K models to realize adaptations. Goal management monitors goals via the virtual machine. When a goal violation is detected, the models associated with an alternative goal that matches the changing conditions are used to update the deployed models via the virtual machine. Goal models can be updated at runtime. Here, we do not focus on the goal layer, we refer the interested reader to [11].

4. COMPARATIVE EVALUATION

We now evaluate the approaches. Section 4.1 describes the experimental setting. Section 4.2 shows the adaptation behavior of SimCA and ActivFORMS on a basic TAS scenario and in response to different runtime changes. Finally, Section 4.3 discusses guarantees provided by both approaches.

4.1 Experimental Setting

We use the TAS case described in Section 2 to compare the adaptation approaches. The TAS exemplar [17] is realized as a Java application and extended with SimCA and ActivFORMS classes. The starting parameters of the services and system requirements are specified in Table 1.

Adaptation is performed once per 100 invocations of TAS (k = 100 inv.). At each adaptation step the application calculates the average measured value of the i-th goal (e.g., measured failure rate) during the past 100 inv. Then it calculates error e_i as the difference between i-th setpoint (e.g., target failure rate) and measured value of the i-th goal. The application also monitors the cost of serving the incoming requests.

The task of both SimCA and ActivFORMS is to keep the goals at their setpoints and minimize the cost. SimCA achieves this task by calculating the value of the *simplex signal*, which represents the probability of selecting the services in the list $\{S1, S2, ..., S9\}$. ActiveFORMS achieves this task by *prioritizing goals* with the help of utility function and updating cost setpoint in case of goal violations. Due to high runtime fluctuations in the values of service parameters, the controller pole p in SimCA is set to 0.98 which allows to reject errors of high magnitude. For the same reason, the cost increment Δ_C in ActivFORMS is set to a low value of 0.1¢ [4].

The application collects the data of the system and the service implementations to build performance graphs, which are used to compare the adaptation approaches in the following section. The x-axis of the graphs are time instants t, each instant corresponds to a series of Δt inv. of TAS. Thus, the y-axis shows the average values of the measured feature per Δt inv. of TAS. Δt can be changed in the TAS interface.

4.2 Adaptation Results

The graphs in Figure 5 show adaptation results of SimCA and ActivFORMS on TAS configured according to Table 1.

[4] We use ¢ symbol to represent cost throughout the paper.

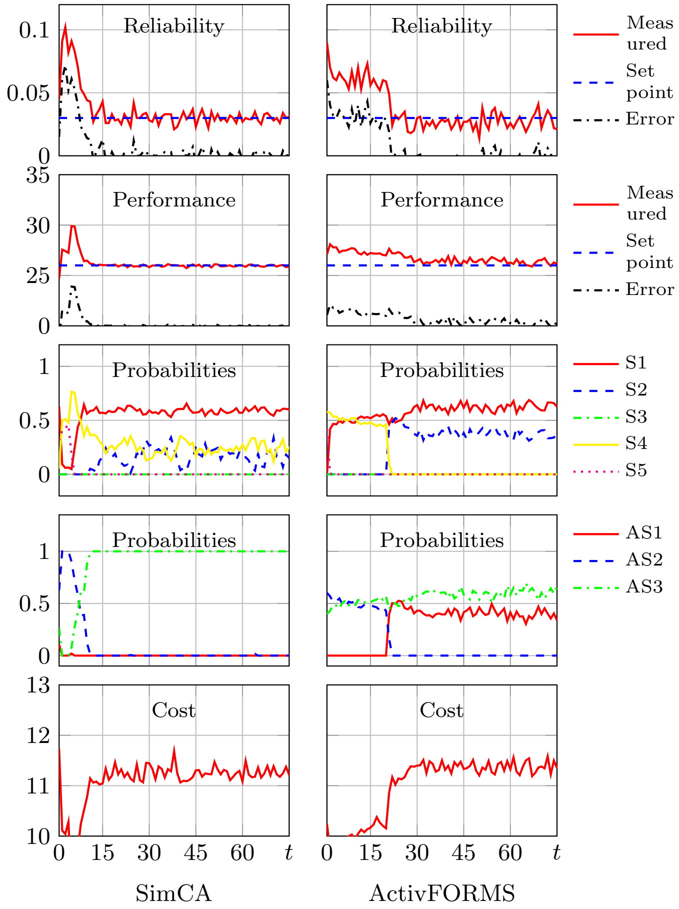

Figure 5: SimCA vs activFORMS on a TAS scenario (Δt = 1000 inv.)

SimCA starts with an identification phase ($t < 7$). The control phase, which is immediately followed by an optimization phase, starts after the relationship between control signals, simplex signal and system output is identified (from t equal 7 onwards). This phase is stable, all minor spikes on the SimCA graphs are caused by the random nature of failures in TAS, e.g., a failure rate of 3% does not always lead to 3 fails per 100 inv.

In the same scenario ActivFORMS starts with a cost setpoint of 8.5¢ which is the average minimal invocation cost of the TAS workflow. In this scenario we assume no prior offline verification of properties so the optimal cost is considered unknown. As a result, both the reliability and performance goals are violated ($t < 20$ on the right graphs). The adaptation slowly increases the cost setpoint which leads to a zero reliability error and a small performance error after $t = 30$.

Comparing TAS reliability achieved by SimCA and ActivFORMS, it is notable that the latter approach requires three times more time to reach a stable state with no goal violation. Comparing performance, ActivFORMS slightly violates the goal even after $t = 75$, but the most notable error can be observed when $t < 30$. So in this case SimCA needs 4 times less time in order to reach a setpoint. As for the picked services, the strategies differ. Though both approaches use S1 approximately equal amount of times (≈ 60%), SimCA prefers a combination of S2, S4 and AS3, while ActivFORMS uses S2, AS1 and AS3.

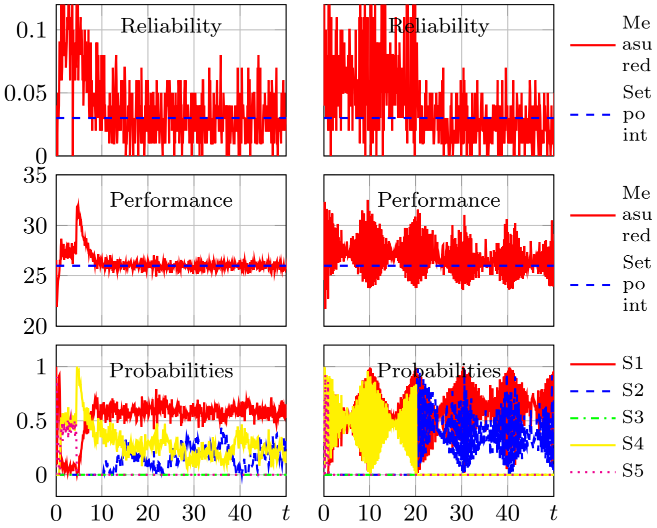

Figure 6: SimCA vs ActivFORMS on a TAS scenario (Δt = 100 inv.)

The Cost graphs (Figure 5) seem to be similar, but closer examination shows that the average cost measured during stable state (t between 30 and 75) is 11.26¢ for SimCA vs 11.37¢ for ActivFORMS. This means a saving of 110¢ per 1000 workflow invocations for SimCA compared to ActivFORMS.

To further compare the approaches, we decrease the measurements interval Δt from 1000 to 100 workflow invocations, see Figure 6. The measured reliability spikes are almost equal on the top plots, however performance and probabilities to select services in ActivFORMS have noticeably higher oscillation amplitude than in SimCA. Such effect is caused by the nature of the algorithm that selects utility function coefficients: the closer the measured value of a goal i gets to the setpoint i, the lower is the priority of goal i.

It is worth mentioning that simplex operating under disturbances has the property of distributing probabilities for selecting services equally among all available services [16]. We do not observe this behaviour in the solution with ActivFORMS. With ActivFORMS, the probabilities of selecting some of the services can suddenly change from 0 to 100% in 1000 workflow inv. (e.g., see the probability to select S2 in $30 < t < 40$). Hence, with SimCA, the load on service providers will be relatively smooth over time, while the load with the ActivFORMS solution can change abruptly, requiring the service providers to keep 100% of their resources available all the time.

Both adaptation approaches have a mechanism that allows to trade-off settling time for other properties, so we study how such trade-off will affect the system output, see Figure 7.

Lowering pole p in SimCA from 0.98 to 0.7 drastically increases the oscillation amplitude of all outputs. After $t = 40$ the system even triggers re-identification as the measured system output values are too far from their setpoints. This is caused by a combination of low settling time (i.e., the system immediately follows any change in the output) and small adaptation period that leads to high measurement errors.

Raising the cost increment Δ_C in ActivFORMS from 0.1 to 1.0 also increases oscillations of all system outputs. However, their amplitude is much lower, see the 'Reliability' and 'Cost' plots. The system is also more stable than in SimCA. This experiment allows to conclude that ActivFORMS is a better

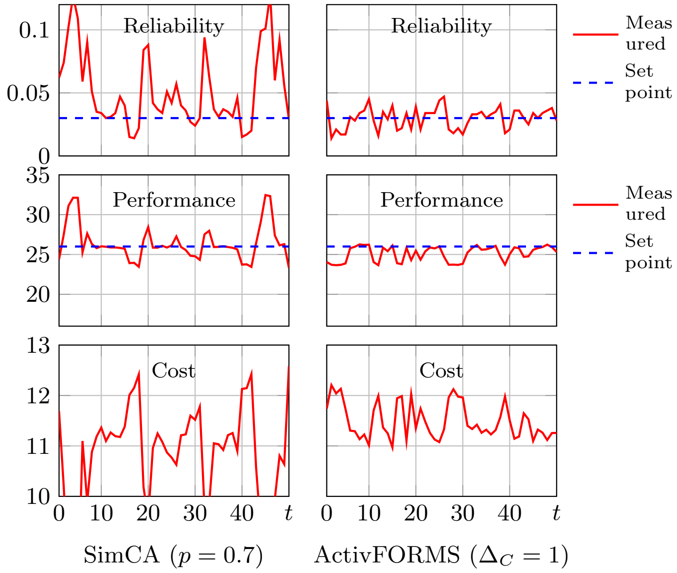

Figure 7: SimCA vs ActivFORMS, the settling time trade-off (Δt = 1000 inv.)

adaptation solution for a system with settling time priority.

Finally, we show how both approaches react to runtime changes, see Figure 8. First of all, having the same initial conditions as in the previous experiments, we set ActivFORMS' cost setpoint to 11.0¢ instead of 8.5¢ because we already know the approximate optimal cost value. This adjustment results almost immediately in convergence to the desired setpoint values. Hence, by doing a system pre-run or offline validation, the ActivFORMS adaptation time can be greatly decreased.

Coming back to Figure 8, there are three major changes happening in the system at runtime:

- *Goal change.* Both goals are changed at t=22: failure rate from 0.03 to 0.05%, resp.time from 26 to 28 ms;
- *Abrupt change.* Medical Service 2 breaks at t=33;
- *Parameter change.* Response time of Medical Service 1 increases from 22 to 52 ms at t=55.

The plots show that both adaptation approaches deal with all types of change: the measured values of both goals follow their setpoints. As in the previous experiments, ActivFORMS requires more time to adapt to the new setpoints.

More importantly, the cost of TAS workflow execution at t=22 decreases to 10.2¢ with SimCA vs 10.8¢ with ActivFORMS. When Service 2 shuts down at $t = 33$, it is already barely used by SimCA because there is another combination of services that can produce the same result with lower cost. However, the ActivFORMS solution even increases the utilization of Service 2 when t is between 22 and 33. When the response time of Service 1 increases at $t = 55$, the difference between the workflow invocation cost is around 1¢ per invocation. This scenario shows that SimCA better solves optimization tasks.

4.3 Guarantees

By using the simplex method, SimCA guarantees optimal cost in TAS without violating the failure rate and response time goals. Simplex has proven to be a practical and fast algorithm for solving this kind of optimization problems [5].

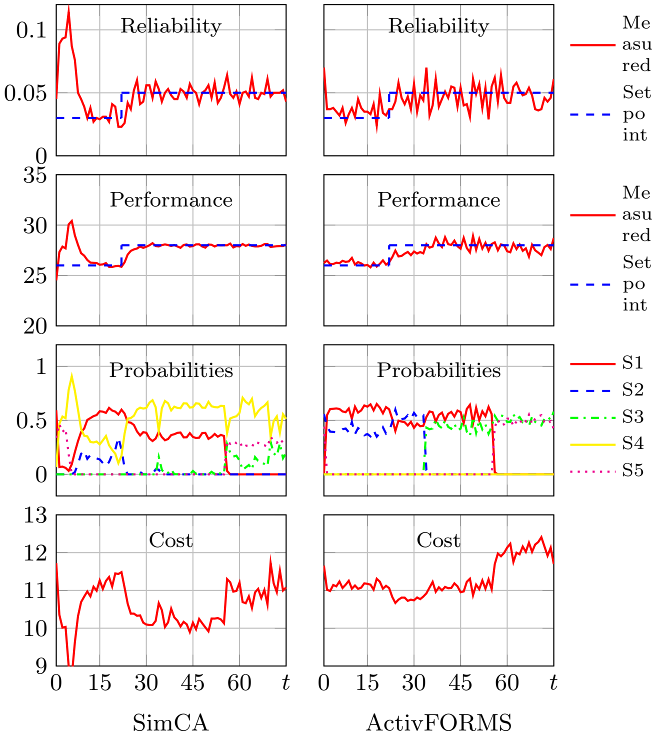

Figure 8: SimCA vs ActivFORMS during runtime changes (Δt = 1000 inv.)

The control-theoretical guarantees provided by SimCA are divided into four main categories: asymptotic stability, steady-state error, settling time and overshoot. If a closed-loop system is asymptotically stable, it reaches the proximity of the setpoint s_i. If the system has zero steady-state error, its setpoint s_i is reached after a certain time and $O_i(k) = s_i(k), k \geq \bar{K}$. The amount of time $\bar{K}$ after this happens is called settling time. If the controlled value exceeded the setpoint before reaching a stable area, this is called overshoot and should be avoided.

The control-theoretical guarantees provided by SimCA are confirmed by the data shown on the Figures:

- The control system is asymptotically stable and converges without overshooting, since it is designed to have only a single pole p which belongs to the open interval $(0, 1)$;
- According to the system output equation of SimCA [16], the output O_i during steady-state equals s_i which leads to a zero steady-state error: $\Delta e = O_i - s_i = 0$. The absence of a steady-state error can be observed on the performance plot of Figure 5 when $t > 15$;
- The settling time $\bar{K}$ of a unit step of every controller i depends on the pole p_i and a constant Δs_i chosen by the system engineer: $\bar{K} = \frac{\ln \Delta s_i}{\ln p_i}$. According to [10, p.85], the commonly used value of Δs is 0.02 (2%). Hence $\bar{K} = \frac{\ln 0.02}{\ln 0.98} = 193.6$. This means that changing the response time from 26 to 28 (step of amplitude 0.2) would take around $193.6 * 0.2 \approx 40$ adaptation steps. This guarantee can be observed on the performance plot of Figure 8 when t is between 20 and 24*;

*As adaptation is performed every 100 workflow invocations

Due to the nature of the simplex method, SimCA will only work when the setpoint of every goal lays between minimal and maximal values of that goal: $min_i \leq s_i \leq max_i$. Therefore, when having an infeasible goal, a system with SimCA will not converge to the closest feasible value for that goal. Instead, the user will be notified that the task is not solvable and the change point detection mechanism will cause an infinitely loop of identification phases, see Figure 9. On the contrary, ActivFORMS will adapt the system output so that it converges to the closest feasible goal value.

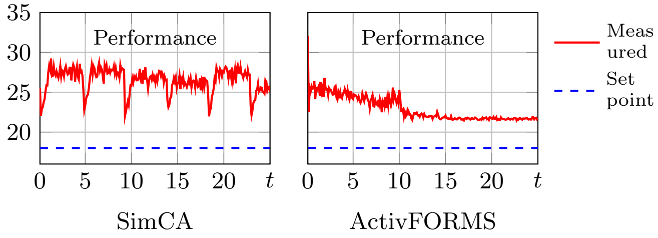

Figure 9: SimCA vs ActivFORMS with infeasible goal (Δt = 1000 inv.)

The ActivFORMS approach provides offline and online guarantees on the system behaviour. The offline guarantees come from the verification of the formal model through TCTL properties at the design time with the Uppaal model checker; verified properties are: reachability, safety, liveness, and deadlock freeness [1].

To achieve the guaranteed cost optimality required by TAS, we first implemented the utility function algorithm as a part of the planner component of a MAPE-K loop. Then, we determined the lowest value of cost that does not violate the goal failure rate and response time with the help of formal offline verification of the following safety property:

```
A[] (gCost == MIN_VALUE) imply
  (measuredFR <= gFR AND measuredRT <= gRT)
```

The system parameters and goals for verification were taken from Table 1. The MAPE-K model is also verified to guarantee the correctness of the algorithm's functionality. E.g., to guarantee that the system model is deadlock free, i.e. it does not have erroneous states and the system does not get stuck in any particular state, Uppaal provides a special formula:

```
A[] not deadlock
```

The following liveness property guarantees that whenever the planning behavior is invoked to create a plan, then eventually a plan is executed.

```
Planning.CreatePlan --> Execution.PlanExecuted
```

The runtime guarantees of ActivFORMS are provided in two ways. First, the model that was formally verified offline is directly executed by the virtual machine at runtime so these guarantees are preserved at runtime. Second, as offline verification is limited to the input provided by the verification models, ActivFORMS allows to continue verification of properties at runtime and inform the user about violations. We refer the interested reader to [11] for details.

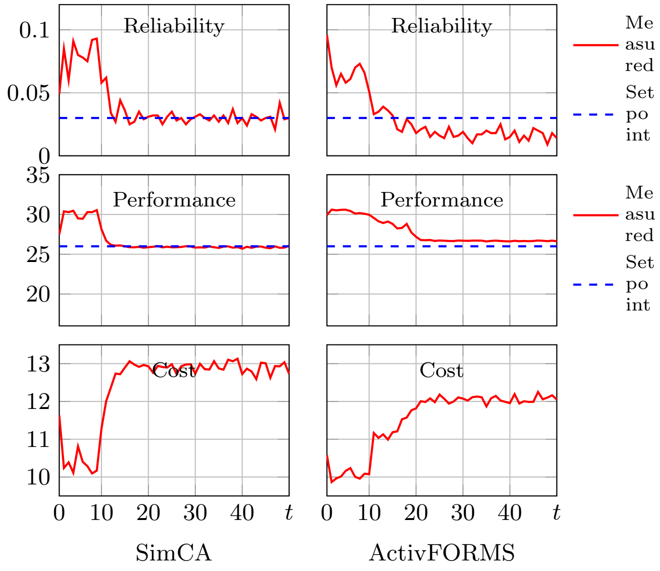

Figure 10: SimCA vs ActivFORMS with a constant disturbance (Δt = 1000 inv.)

In the studied TAS scenarios we used both types of runtime guarantees provided by ActivFORMS. The direct execution of the verified model allowed to guarantee the correctness of the adaptation mechanism implementation; the runtime verification mechanism of ActivFORMS guaranteed that the minimal cost, previously determined offline, does not lead to violation of the failure rate or response time goals at runtime. To insure the latter, we added the following runtime goals:

```
G1: A[] (measuredFR <= gFR AND measuredRT <= gRT)
G2: A[] (measuredFR > 0.9*gFR AND measuredRT > 0.9*gRT)
```

Goal $G1$ assures that the failure rate and response time of TAS do not exceed their goals. Goal $G2$ assures that the values of both goals will not drop lower than 90% of their goals, i.e. the system will not waste resources. In case of a runtime goal violation, the cost setpoint $gCost$ is adapted accordingly: for $G1$ violation $gCost$ is increased by the cost increment Δ_C, for $G2$ violation $gCost$ is decreased by Δ_C.

Changing Δ_C allows trading-off the speed of the system reaction to runtime changes (settling time) for the amplitude of system outputs oscillations around the goal value (accuracy).

4.4 Disturbance Handling

Throughout the experiments discussed in Section 4.2, both approaches handled randomly distributed measurement disturbances caused by the changes of service parameters at runtime. Different to ActivFORMS, SimCA allows to formally evaluate the amount of disturbance the system can withstand.

According to the PBM approach [8] that lays at the core of SimCA, the amount of disturbance the system can withstand $\Delta(d)$ by using a PBM controller can be estimated as follows: $0 < \Delta(d) < \frac{2}{1-p}$. This means that the value of the pole p defines how SimCA will react to disturbances. For $p = 0.98$ that was used in most experiments, the measurement can be inaccurate by a factor of 100, and the controller of SimCA will still adapt the system to follow the goals.

However, as mentioned in Section 4.3, the simplex method will not be able to reach an infeasible goal. Hence, the measurement inaccuracy on values of goal i that SimCA can with-

and Δt = 1000 invocations, 4 time steps on the graph equals to 40 adaptation steps stand is: $min_i \leq \Delta(d) \leq max_i$. This property of SimCA can be observed on Figure 7 after $t = 40$: the reliability output reaches a value that is higher than maximal achievable value for a TAS workflow which causes a re-identification phase.

When it comes to constant disturbances, both approaches successfully perform adaptation. In the scenario shown on Figure 10, there is a constant response time measurement disturbance of +3 ms (e.g., the sensor shows 29 ms instead of 26 ms). As in the previous experiments with the same pole and cost increment, SimCA converges to the setpoint faster. It is notable that ActivFORMS achieves better cost by slightly violating the performance goal.

5. DISCUSSION

Both SimCA and ActivFORMS successfully solved the adaptation problem. The choice between these approaches depends on the particular scenario. When it comes to formal guarantees, both approaches provide the main guarantee required from adaptation, i.e. to minimize the cost without violating performance and reliability goals. SimCA guarantees this by using the simplex method and ActivFORMS verifies the solution optimality offline. However, at runtime SimCA performs better because offline system validation of ActivFORMS relies on particular system properties that may not hold online.

Other formal guarantees provided by the two approaches are different. With SimCA it is possible to formally prove a number of system properties, such as stability, settling time, amount of disturbance the system withstands, etc., which do not depend on the particular system parameters. On a contrary, with ActivFORMS these properties are inherent to the underlying algorithm, such as a utility function that was used in our study, and do not come from the design of the adaptation mechanism. With ActivFORMS it is possible to formally verify these properties of the algorithm on a particular set of system parameters by exploring the whole state space. However, trying to verify the TAS system with all possible combinations of service parameters will lead to an explosion of the state space. This is compensated by runtime verification.

On the other hand, with ActivFORMS the correctness of the implementation of adaptation mechanism can be formally proven, in particular the absence of erroneous states and correct interaction between adaptation components.

6. CONCLUSIONS

In this paper we presented a comparative evaluation of the two different approaches for self-adaptation: SimCA which is based on control theory and linear optimization, and architecture-based ActivFORMS equipped with a utility function. The study was performed using the TAS exemplar. The analysis of adaptation results have shown that both approaches can deal with multiple goals and provide guaranteed optimality with respect to an additional goal. However, SimCA achieves better results in the presence of runtime changes as it does not rely on data verified at design time. At the same time, ActivFORMS is a good choice for systems that require low settling time.

Except optimality, the two adaptation approaches offer different guarantees. The design of SimCA adaptation mechanism, such as the pole value or the system output equation, allows to formally prove the properties of underlying system and guarantee that they will hold at runtime independent of system parameters. ActivFORMS allows formal verification of system properties based on the particular input data. Runtime verification allows complementary verification when the system parameters change. An important feature of ActivFORMS is that it allows to formally guarantee the functional correctness of the implementation of adaptation algorithm.

This work is a first step towards understanding the effectiveness and formal power of different adaptation mechanisms. As a part of future efforts, we want to compare other adaptation approaches, such as QoSMOS [2] to SimCA and ActivFORMS. We are also planning to test the adaptation mechanisms on different types of systems and scenarios.

7. REFERENCES

[1] G. Behrmann, R. David, and K. G. Larsen. A tutorial on UPPAAL. pages 200–236. Springer, 2004.

[2] R. Calinescu et al. Dynamic qos management and optimization in service-based systems. *IEEE TSE*, 37(3):387–409, May 2011.

[3] R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola. Self-adaptive software needs quantitative verification at runtime. *Commun. ACM*, 55(9), 2012.

[4] B. H. Cheng et al. Software engineering for self-adaptive systems: A research roadmap. In *SEfSAS*, 2009.

[5] G. B. Dantzig and M. Thapa. *Linear Programming 2: Theory and Extensions*. Springer, 2003.

[6] D. G. de la Iglesia and D. Weyns. MAPE-K formal templates to rigorously design behaviors for self-adaptive systems. *ACM TAAS*, 2015.

[7] R. de Lemos et al. Software engineering for self-adaptive systems: A second research roadmap. In *SefSAS II*, 2012.

[8] A. Filieri, H. Hoffmann, and M. Maggio. Automated design of self-adaptive software with control-theoretical formal guarantees. In *ICSE*, 2014.

[9] A. Filieri, M. Maggio, et al. Software Engineering Meets Control Theory. In *SEAMS*, 2015.

[10] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[11] M. U. Iftikhar and D. Weyns. ActivFORMS: Active formal models for self-adaptation. In *SEAMS*, 2014.

[12] J. Kephart and D. Chess. The vision of autonomic computing. *Computer*, 36(1), 2003.

[13] J. Kramer and J. Magee. Self-Managed Systems: an Architectural Challenge. *FOSE*, 2007.

[14] M. Maggio and H. Hoffmann. ARPE: A tool to build equation models of computing systems. In *Feedback Computing*, 2013.

[15] P. Oreizy, N. Medvidovic, and R. Taylor. Architecture-based runtime software evolution. In *ICSE*, 1998.

[16] S. Shevtsov, D. Weyns, and M. Maggio. Keep it SIMPLEX: Handling multiple concerns in control-based self-adaptive systems. *Technical Report, 2015. Online: http://homepage.lnu.se/staff/daweaa/papers/SimCA-report.pdf.*

[17] D. Weyns and R. Calinescu. Tele assistance: A self-adaptive service-based system exemplar. In *SEAMS*, 2015.

MORPH: A Reference Architecture for Configuration and Behaviour Self-Adaptation*

Victor Braberman[1], Nicolas D'Ippolito[1], Jeff Kramer[2], Daniel Sykes[2], Sebastian Uchitel[1,2]
[1] Departamento de Computación, FCEN, Universidad de Buenos Aires, Argentina
[2] Department of Computing, Imperial College London, UK

ABSTRACT

An architectural approach to self-adaptive systems involves runtime change of system configuration (i.e., the system's components, their bindings and operational parameters) and behaviour update (i.e., component orchestration). Thus, dynamic reconfiguration and discrete event control theory are at the heart of architectural adaptation. Although controlling configuration and behaviour at runtime has been discussed and applied to architectural adaptation, architectures for self-adaptive systems often compound these two aspects reducing the potential for adaptability. In this paper we propose a reference architecture that allows for coordinated yet transparent and independent adaptation of system configuration and behaviour.

Categories and Subject Descriptors

D.2 [**Software Engineering**]

General Terms

Design

Keywords

Self-adaptive Systems, Software Architecture

1. INTRODUCTION

Self-adaptive systems are capable of altering at runtime their behaviour in response to changes in their environment, capabilities and goals. Research and practice in the field has addressed challenges of designing these systems from multiple perspectives and levels of abstraction.

It is widely recognised that an architectural approach to achieve self-adaptability promises a general coarse grained framework that can be applied across many application domains, providing an abstract mechanism in which to define runtime adaptation that can scale to large and complex systems [21].

Architecture-based adaptation involves runtime change of system configuration (e.g., the system's components, their bindings, and operational parameters) and behaviour update (e.g., component orchestration).

Existing approaches to architectural adaptation (e.g. [16, 10] incorporate elements from two key areas to enable runtime adaptation: Dynamic reconfiguration [9, 21] and discrete-event control theory [12, 22, 8]. The first, key for adapting the system configuration, studies how to change component structure and operational parameters ensuring that on-going operation is not disrupted and/or non-functional aspects of the system are improved. The second, key for adapting behaviour, studies how to direct the behaviour of a system in order to ensure high-level (i.e., business, mission) goals.

Although the notions of configuration and behaviour control are discussed and applied by many authors, they are typically compounded when architectures for adaptation are presented, reducing overall architectural adaptability. Automated change of configuration and behaviour address different kinds of adaptation scenarios each of which should be managed as independently as possible from the other. Nonetheless, configuration and behaviour are related and it is not always possible to change one without changing the other. The need for both capabilities of independent yet coordinated adaptation of behaviour and configuration requires an extensible architectural framework that makes explicit how different kinds of adaptation occur.

Consider a UAV on a mission to search for and analyse samples. A failure of its GPS component may trigger a reconfiguration aiming at providing a location triangulating over alternative sensor data. The strategy may involve passivating the navigation system, unloading the GPS component and loading components for other sensors in addition to the component that resolves the triangulation. A behaviour strategy that is keeping track of the mission status (e.g. tracking areas remaining to be traversed, samples collected, etc.) should be oblivious to this change.

A reconfiguration adaptation strategy that can cope with the GPS failure can be computed automatically using approaches based on, for example, SMT solvers or planners [22] that consider the structural constraints provided in the system specification (e.g., the need for a location service), requirements and capabilities of component types (e.g., the requirements of a triangulation service) and runtime infor-

*This work was partially supported by grants ERC PBM-FIMBSE, ANPCYT PICT 2012-0724, UBACYT W0813, ANPCYT PICT 2011-1774, UBACYT F075, CONICET PIP 11220110100596CO, MEALS %295261.

CTSE'15, August 31, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3814-1/15/08...$15.00
http://dx.doi.org/10.1145/2804337.2804339

mation of available component instances (e.g., the availability of other sensors).

The arrival of the UAV at an unexpected location due to, say, unanticipated weather conditions may make the current search and collection strategy inadequate. For instance, the new location may be further away from the base than expected and the remaining battery charge may be insufficient to allow visiting the remaining unsearched locations before returning to base. In this situation the behaviour strategy would have to be revised to relinquish the goal of searching the complete area before returning to base in favour of the safety requirement that battery levels never go below a given threshold. The new behaviour strategy may reprioritise remaining areas to be searched (in terms of importance and convenience), visiting only a subset of the remaining locations as it moves towards the base station for recharging. Once recharged, the strategy may attempt to revisit the entire area under surveillance but prioritising locations previously discarded. Such behavioural adaptation should be independent to the infrastructure supporting reconfiguration control.

A behaviour strategy that can deal with unexpected deviations in the UAV's navigation plan can be computed automatically using approaches based on, for instance, controller synthesis [8] that consider a behaviour model describing the capabilities of the UAV (e.g. autonomy), environment (e.g., map with locations of interest and obstacles) and system goals (e.g. UAV safety requirements and search and analyse – liveness – requirements). Indeed, our proposal of an explicit separation of reconfiguration and behaviour strategy computation and enactment is in line with the design principles of a separation of concerns and information hiding. The behaviour strategy is oblivious to the implementation that provides the services it calls and the reconfiguration strategy supports the injection of the dependencies that are required by the behaviour strategy oblivious to the particular ordering of calls that the behaviour strategy will make. In a sense, the design principle which is known to support changeability supports runtime changeability, which ultimately is what adaptation is about.

Configuration and behaviour adaptation may however need to be executed in concert. *Consider the scenario in which the gripper of the UAV's arm that is to be used to pick up samples becomes unresponsive. With a broken gripper the original search and analyse mission is unachievable. This should trigger an adaptation to a degraded goal that aims to analyse samples via on-board sensors and remote processing. This goal requires a different behaviour strategy (e.g. circling samples once found to perform a 360 degree analysis) but also a different set of services provided by different components (e.g. infra-red camera). Not only are both behaviour and configuration adaptation required, but also their enactment requires a non-trivial degree of provisioning: To set up the infra-red camera, the UAV requires folding the arm to avoid obstructing the camera's view; performing such an operation while in the air is risky. Hence, coordination between configuration and behaviour adaptation is needed: First, a safe landing location must be found, then arm folding must be completed, and only then can the reconfiguration start. New components are loaded and activated, and finally, a strategy for in-situ analysis, rather than analysis at the base, can start.*

It is in the combined configuration and behaviour adaptation where the need for both separation of concerns and explicit architectural representation of coordination becomes most evident. Approaches to automated computation of configuration and behaviour adaptation strategies require different input information and utilise different reasoning techniques. Both automated reasoning forms are of significant computational complexity and require careful abstraction of information. Keeping resolution of configuration and behaviour adaptation separately allows reusing existing and future developments in the fields of dynamic reconfiguration and control theory and also helps keep computational complexity down.

The broken UAV gripper scenario requires a coordinated behaviour and reconfiguration adaptation strategy. The adaptation required can be decomposed into a behaviour control problem that assumes that a reconfiguration service is available and a reconfiguration problem. The resulting behaviour strategy will be computed on the assumption that the UAV's capabilities will conform to the current configuration (e.g. grip command fails) until a reconfigure command is executed, and that from then on different capabilities will be available (e.g. infra-red camera getPicture command available). The behaviour strategy computation will also consider restrictions on when the reconfigure command is allowed (e.g. when arm is folded) and new goals (360 degree picture analysis rather than collect). The computation of the reconfiguration strategy does not entail additional complexity and is oblivious to the fact that a behaviour strategy that involves a reconfiguration halfway through is being computed.

In the above scenario, what needs to be resolved at the architectural level of the self-adaptation infrastructure is which architectural element is responsible for the decomposition of the adaptation strategy into a behaviour strategy and a reconfiguration strategy, and also how strategy enactment is performed to allow the behaviour strategy to command reconfiguration at an appropriate time (and possibly even account for reconfiguration failure). Indeed, an appropriate architectural solution to this would enable guaranteeing that given a correct decomposition of the overall composite adaptation problem into configuration and behaviour adaptation problems, and given correct-by-construction configuration and behavioural strategies for these problems, the overall adaptation problem is correct.

In this paper we present MORPH, a reference architecture for behaviour and configuration self-adaptation. MORPH makes the distinction between dynamic reconfiguration and behaviour adaptation explicit by putting them as first class entities. Thus, MORPH allows both independent reconfiguration and behaviour adaptation building on the extensive work developed but also allowing coordinated configuration and behavioural adaptation to accommodate for complex self-adaptation scenarios.

2. MORPH

The MORPH reference architecture builds upon the large body of work related to engineering self-adaptive emphasising the need to make behaviour and reconfiguration control first-class architectural entities. In particular, it draws inspiration from [16, 10], which are discussed below.

2.1 Background

The MAPE-K model shows how to structure a control loop in adaptive systems. The four key activities (Monitor, Analysis, Plan and Execute) are performed over a shared data structure that captures the knowledge required for adaptation. The MAPE-K model does not prescribe what knowledge is to be captured nor what aspect of the system is to be controlled. Thus, there is no explicit treatment or distinction between configuration and behaviour adaptation let alone prescribed mechanisms for dealing with coordinated and independent configuration and behaviour adaptation.

The need to deal with hierarchies of control loops in autonomous systems is widely recognised (e.g. [11]). Lower levels are typically low latency loops that focus on more tactic and stateless objectives that involve less monitored and controlled elements while higher levels tend to focus on more stateful and strategic objectives involving multiple controlled and monitored aspects that require higher latency loops. The need for hierarchy in architectural self-adaptation is discussed in [16]. A three-tier architecture is proposed to provide a separation of concerns and to address a key architectural concern related to dealing with the complexity of run-time construction of adaptation strategies. The architecture structures hierarchically the MAPE-K loops introducing a separation of concerns in which complex, strategic, resource consuming analysis is performed in top layers while simple, more tactical adaptation is performed in lower layers.

Unlike the MAPE-K model, the architecture in [16] prescribes the kind of control that is effected on the adaptive system by establishing a clear interface between the adaptation infrastructure and the component based system to be adapted. The architecture assumes an interface on which it can take action on the current system configuration by creating and deleting components, binding and unbinding components through their required and provided ports and setting component modes (i.e., configuration parameters).

Although the three-tier architecture provides a clear separation of the concerns of behavioural planning from component reconfiguration, this is purely hierarchical, with the behaviour plan dictating the required structural (re) configuration at lower layers. Although this allows for independent structural configuration alone if the behaviour plan is still satisfied, it is less clear how behaviour control and configuration control should work together, hindering the possibility of more elaborate behaviour and configuration control and the potential for reasoning about adaptation guarantees.

The Rainbow [10] framework instantiates and refines the MAPE-K architecture providing an extensible framework for sensors and actuators at the interface between the control infrastructure and the target system. The architecture recognises the complexity of the interface between the MAPE-K infrastructure (referred to by the authors as the architecture layer) and the component system to be adapted (referred to as the system layer). The Rainbow framework introduces additional infrastructure into the architecture and system layers in addition to accounting for an extra layer between the two: the translation layer. Monitoring is split amongst the three layers: probes are introduced as system layer infrastructure to support observation and measurement of low-level system states. Gauges are part of the architectural infrastructure layer and aggregate information from the probes to update appropriate properties of the knowledge base used for the MAPE activities. The translation layer resolves the abstraction gap between the system layer and the architectural layer, for instance relating abstract component identifiers in the later concrete process identifiers and machine identifiers in the former.

Rainbow focuses on achieving self-adaptation through configuration adaptation. Thus, as in [16] focus is on changing component instances and bindings and also effecting behaviour by changing operational parameters (thread pool size, number of servers, etc.). Indeed, the framework does not account explicitly for automated construction of behaviour strategies that control the functional behaviour of the system layer components. As in [16], the distinction between configuration and behaviour control is not elaborated explicitly in the architecture.

In the following, we propose an architecture that takes inspiration from the architectures discussed above but that includes the design concern related to supporting independent yet coordinated behaviour and configuration adaptation. The architecture combines the three tier structure from [16] to address varying latency of architectural self-adaptation control. We design each layer as a MAPE-K control loop, resulting in a hierarchical control loop structure as in [17]. As in Rainbow [10] we address the problem of bridging the gap between the managed component architecture and the adaptation infrastructure encapsulating the former and also providing a decoupled mechanism for aggregation and inference over logged system data.

2.2 Architectural Overview

We start with a very brief introduction of the main architectural elements to give a general picture of how the architecture works before we go into detail of the workings and rationale of each element. A graphical representation of the architecture can be found in Figure 1. In the remaining text, when we want to emphasise traceability to the figure we will use an alternative font.

The architecture is structured in three main layers that sit above the target system: Goal Management, Strategy Management and Strategy Enactment. Orthogonal to the three layers is the Common Knowledge Repository. Each layer can be thought of as a implementing a MAPE-K loop. The top layer's MAPE-K loop is responsible for reacting to changes in the goal model that require complex computation of strategic, possibly configuration and behavioural, adaptation. Its knowledge base is the Common Knowledge Repository. The Strategy Management layer's MAPE-K loop is responsible for adapting to changes that can be addressed using pre-processed strategies. It selects pre-computed strategies based on the Common Repository Knowledge and a set of internally managed pre-computed strategies. The Strategy Enactment layer's MAPE-K loop is responsible for executing strategies; its knowledge base is primarily the strategy under enactment.

The Target System abstracts the Component Architecture that provides system functionality. The Component Architecture is harnessed by effectors and probes which allow the Strategy Enactment Layer to interface with system components. The Knowledge Repository stores in a Log the execution data produced by Target System and also stores in the Goal Model) the result of Inference procedures that produce knowledge regarding the system state, goals and environment assumptions. We expect users, administrators and other stakeholders to also produce modifications to the Knowledge Repository, and in particular the goals and environment assumptions.

The three layers that provide the architectural adaptation infrastructure are each split into reconfiguration and a behaviour aspects. The Goal Management layer has a Goal

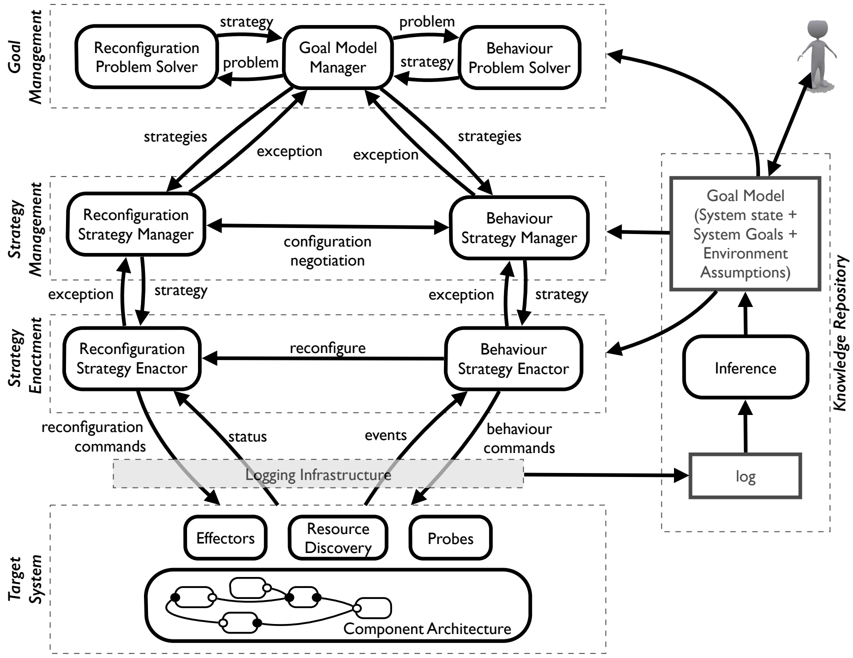

Figure 1: The MORPH Reference Architecture.

Model Manager whose main responsibility is to decompose adaptation problems into reconfiguration and behaviour problems, each of which is given to a specific Solver to produce a strategy that can achieve the required adaptation. The top layer sends reconfiguration and behaviour strategies downwards. The bottom two layers have architectural elements to handle reconfiguration and behaviour strategies separately but interact with each other when and if needed to maintain overall consistency. The Strategy Management layer entities interact to ensure that they select consistent strategies to be executed by the Strategy Enactment layer (c.f., configuration negotiation). The Strategy Enactment layer entities interact to ensure that the execution of their respective strategies is done consistently over time (c.f., reconfigure command).

2.3 Target System

Responsibility: The main responsibility is to achieve the system goals, encapsulate implementation details and provide abstract monitoring and control mechanisms over which behaviour and structure of the system can be adapted.

Rationale: This is to encapsulate the instrumentation of the system-to-be-adapted to support a flexible and reusable framework for monitoring, analysing, planning and executing adaptation strategies in the layers above.

Structure and Behaviour: The Target System, strongly inspired by [10], contains the component architecture that provides the managed system's functionality *(*e.g., GPS, video, telemetry and navigation components). It also contains instrumentation to monitoring and control of the component architecture. Two types of effectors are provided. The first provides an API to add, remove and bind components, in addition to setting operational parameters of these components. We refer to the invocation of operations on these effectors as reconfiguration commands. These effectors are application domain independent, and they provide the adaptation infrastructure an abstraction over the concrete deployment infrastructure on which the component architecture runs *(e.g., the UAVs operating system)*. The second effector type, behaviour actions, is domain dependent and provides an API that invokes functional services provided by components of the component architecture. *The UAV's navigation component may exhibit a complex API which is abstracted into simple commands (e.g. goto(Location)) that are to be used as the basis for behaviour strategies.*

The mechanism for monitoring of the component architecture can be provided by probes that reveal state information. As with effectors, monitoring information can be classified into two kinds. We have on one hand information regarding the status of components. This kind of information is application independent. Status of a component may indicate if it is active, inactive, connected or killed. On the other hand we refer to as events the application domain relevant information that flows from the Target System to the Strategy Enactment Layer. *UAV events may include notifications regarding battery depletion, or acknowledgements of having reached a requested location.*

Between the target system and the adaptation infrastructure a translation layer is required to provide translation services that aim to bridge the abstraction gap between the knowledge representation required to perform adaptation at the architectural level and the concrete information of the actual implementation. *In the UAV, this may include resolving event handlers, process ids, in addition to domain specific translations such the conversion of continuous variable for battery level to a discrete battery depleted event.*

2.4 Common Knowledge Repository

Responsibility: The key responsibility of the repository is to keep an up to date goal model at runtime based on inferences made over continuous monitoring of the environment to detect changes in goals, behaviour assumptions and available infrastructure.

Rationale: The design rationale for the repository is to decouple the accumulation of runtime information of the target system from the complex computational processes involved in abstracting and inferring high-level knowledge that can be incorporated, for subsequent adaptation, into a structured body of knowledge regarding stakeholder goals, environmental assumptions and target system capabilities.

Structure and Behaviour: The common knowledge repository stores information about the target system, the goals and environment assumptions. It consists of two data structures (a log and a goal model) and an Inference procedure.

The Goal Model: This is the key data architectural element of the repository. We use the term "goal model" in the sense of goal oriented requirements engineering [18].

The point of keeping a structured view of the world that includes requirements and assumptions, with multiple ways of achieving high-level goals and preference criteria over these alternatives is that at runtime it is possible to change the way a goal is achieved by selecting a different OR-refinement. The combinatorial explosion of possible OR-refinement resolutions can be a rich source for adaptation which is exploited in the Goal Management Layer. In addition, this representation of rationale, is amenable to being updated and changed as new information is acquired.

2.5 Goal Management Layer

Responsibility: The main responsibility of the Goal Management Layer is to deal with and anticipate changes in the stakeholder goals, environment assumptions and system capabilities by pre-computing adaptation strategies consisting of separate behaviour and reconfiguration strategies.

Rationale: The rationale for this layer is based on two core concepts: The first is that the adaptive system must be capable of performing strategic, computationally expensive, planning independently of and concurrently with the execution of pre-computed strategies (occurring in lower layers). The second is to decompose adaptation into a behaviour strategy that controls the system to an interface and a reconfiguration strategy that injects the dependencies on concrete implementations that the behaviour strategy will use. Decomposing adaptation along the modular design improves support for adaptability allowing behaviour and configuration changes independently.

Structure and Behaviour: The layer has three main entities, the Goal Model Manager, the Behaviour Problem Solver and the Reconfiguration Problem Solver.

The Goal Model Manager: This is the key element of the layer responsible for three core tasks: The first is to decide when a new adaptation strategy must be computed, the second is to resolve all OR-refinements in the goal model and select the requirements for to be achieved by the system, and third to decompose the requirements into achievable reconfiguration and behaviour problems. Concrete strategies for reconfiguration and behaviour are computed by the solvers.

Production of adaptation strategies can be triggered by requests for plans from layers below or internally due to the identification of significant changes in the goal model. The former may correspond to a scenario in which a failure is propagated rapidly upwards from the target system: *For instance, the UAV's gripper component fails. The Strategy Enactment layer, which is executing a strategy that requires the gripper, immediately informs that its current strategy is unviable and requests a new strategy to the Strategy Management Layer.* If all pre-computed plans require the arm to pick up objects, a new strategy for achieving system goals is requested to the Goal Model Manager.

The alternative, internal, triggering mechanism corresponds to scenarios in which the goal model is changed because of new information inferred from the log or input manually by some stakeholder. *For instance, weather conditions may lead to inferring higher energy consumption rates from logged information. What would follow is a revision of the assumptions on UAV autonomy stored in the Goal Model. Such alteration may trigger the re-computation and downstream propagation of search strategies to make more frequent recharging stops.*

Adaptation strategies are decomposed into a strategy for achieving the component configuration that can provide the functional services required achieve selected requirements and a behaviour strategy that can call these services in an appropriate temporal order to satisfy the requirements. *The adaptation strategy that deals with the broken gripper must reconfigure the system to use a different set of components (e.g. the infra-red camera) and coordinate its use upon reaching a position where there is a sample to be inspected.*

As discussed in the Introduction, decomposition allows adaptation of the system configuration transparently to the behaviour strategy being executed (*e.g., changing the location mechanism*) or the behaviour strategy transparently to the configuration in use (*e.g., changing the route planning strategy*). In addition, decomposition allows for the computation of multiple behaviour strategies for a given configuration (*e.g. different search and collect strategies that assume different UAV autonomy can be run on a configuration that has a gripper component*) and different configurations can be used for a given behaviour strategy (*e.g. different configurations for providing a positioning service can be used for the same search strategy*).

One of the design rationales for this layer is the precomputation of expensive adaptation strategies that are then ready to use when needed. This means that multiple reconfiguration and behaviour strategies may be constructed. Indeed, the Goal Model Manager can pre-compute, and propagate downwards, many reconfiguration and behaviour strategies for one resolution of the OR-refinements of the goal model. *This may be useful, for example, if it is known that information regarding UAV autonomy is imprecise, multiple (behaviour) search strategies for searching the area may be developed so that the infrastructure can adapt quickly as soon as the predicted UAV autonomy differs significantly from what can be inferred from the monitored energy consumption. Similarly, should the GPS-based location service be known to fail (perhaps do to environmental conditions), then various reconfiguration plans may be pre-computed to allow adaptation to alternative positioning systems when needed.*

Configuration Problem Solver: The layer has two entities capable of automatically constructing strategies for given adaptation problems. The Configuration Problem Solver focuses on how to control the target system to achieve a specified configuration given the current system configuration, configuration invariants that must be preserved and component availability. Configuration invariants may include structural restrictions forcing the architecture to conform to some architectural style or other considerations based, for instance, on non-functional. In the UAV example such restrictions may include that the attitude control components never be disabled or the total number of active components never be beyond a given threshold to avoid battery overconsumption.

Reconfiguration problem solvers build strategies that call actions that add and remove components, activate and passivate them, and establish or destroy bindings between them. These reconfiguration actions are part of the API exposed by the target system. The strategy may sequence these actions or have an elaborate scheme that decides which actions to call depending on feedback obtained through the information on the status of components exhibited by the target system API.

To automatically construct strategies the solvers can build upon a large body of work developed in the Artificial Intelligence and Verification communities, including automatic planners (e.g. [3]), controller synthesis (e.g. [8]), and model checking. Such techniques have been applied to construction of reconfiguration strategies in [21].

Behaviour Problem Solver: This entity focuses on how to control the target system to satisfy a behaviour goal. In contrast to reconfiguration problems, the behaviour goal may not be restricted to safety and reachability (i.e. reach a specific global state while preserving some invariant). Behaviour goals may include complex liveness goals such as to have the UAV monitor indefinitely an area for samples to inspect. Behaviour problem solvers produce strategies, which can be encoded as automata that monitor target system events and invoke target system actions.

In addition to the expressiveness of goals that behaviour strategies must resolve, there is an asymmetry between reconfiguration and behaviour problems. To resolve the coordination problem between strategies (*as with folding the UAV arm before a reconfiguration to deal with a gripper failure can be executed, see Introduction*), the behaviour strategies produced by the solver can invoke a reconfigure command, which triggers the execution of a reconfiguration strategy. We explain how this triggering works in Section 2.6.

2.6 Strategy Management Layer

Responsibility: This layer selects and propagates pre-computed behaviour and reconfiguration strategies to be enacted in the layer below. For this, the layer must store and manage pre-computed behaviour and reconfiguration plans, and request new strategies to the layer above when needed. It must also ensure that the behaviour and reconfiguration strategies sent to the lower layer are consistent, indicating their relationships.

Rationale: They main concept for the layer is to allow rapid adaptation to failed strategy executions (or capitalizing rapidly on opportunities offered by new environmental conditions) by having a restricted universe of pre-computed alternative behaviour and reconfiguration strategies that can be deployed independently or in a coordinated fashion.

Structure and Behaviour The layer has two entities that work in similar fashion mimicking much of the layer's responsibilities but only on either behaviour or reconfiguration strategies. However, the Behaviour Strategy Manager and the Reconfiguration Strategy Manager are not strictly peers. In some adaptation scenarios the former will take a Master role in a Master-Slave decision pattern.

Behaviour Strategy Manager: This entity stores and manages multiple behaviour strategies. From these strategies it picks a behaviour strategy to be enacted in the layer below. The selection of strategy may be triggered by an exception raised by the layer below or internally due to a change identified in the common knowledge repository. The former may occur when the behaviour strategy being executed finds itself in a unexpected situation it cannot handle. *For instance, the UAV executing a particular search strategy expects to be at a specific location with at least 50% of its battery remaining but finds that it is below that threshold, invalidating the rest of the strategy for covering the area to be searched.* At this point the Strategy Enactment Layer signals that the assumptions for its current strategy are invalid and requests a new strategy to this layer.

The other scenario that can trigger the selection of a new strategy is a change in the common knowledge repository. *Consider again the problem of unexpected energy overconsumption. An inference process in the knowledge repository may update the average energy consumption rate periodically based on Target System information being logged. This average may be well above the assumed consumption average for the behaviour strategy being executed. The Behaviour Strategy Manager may decide that it is plausible that the current behaviour strategy will fail and may decide to deploy a more conservative search strategy.*

Note that the two channels that may trigger the selection of a new strategy differ significantly in terms of latency and urgency. The exception mechanism provides a fast propagation of failures upwards, indicating that the strategy being currently enacted is relying on assumptions that have just been violated. This means that any guarantees on the success of the current strategy in satisfying its requirements are void and a new strategy is urgently required. The monitoring of changes in the knowledge repository is a process that incurs in comparatively significant delays as the inference of goal model updates based on logged information may be performed sporadically and consume a significant amount of time. The upside of this second channel is that it may predict problems sufficiently ahead of their occurrence, providing time to select pre-computed strategies to avoid them.

The selection of a behaviour strategy is constrained by the current configuration of the target system (which determines the events and actions that can be used by the strategy) and the alternative configurations that may be reached by enacting one of the pre-computed re-configuration strategies. Furthermore, the selection is informed by preferences defined in the goal model on which OR-refinement resolution is preferred. *Thus, a new strategy that can be supported by the current UAV configuration may be selected to alter the search strategy. Alternatively or a strategy that no longer picks samples up to avoid the extra consumption produced by load carrying may be chosen. In the later case, in-situ analysis is required and hence a reconfigured UAV with an infra-red camera in place is required. Selecting such a pre-computed behaviour strategy is subject to the availability of a pre-computed reconfiguration strategy that can reach a configuration with an active infra-red camera module.*

The Behaviour Strategy Manager deploys the selected strategy by performing two operations. Firstly, should the selected strategy require a configuration with characteristics that are currently not provided, it commands the Reconfiguration Strategy Manager to deploy an appropriate reconfiguration strategy (c.f. Master-Slave relationship). Secondly, the manager hot-swaps the current behaviour strategy being executed in the layer below with the newly selected strategy, setting the initial state of the new strategy consistently with the current state of the Target System. Note that should the new strategy be replacing a strategy that is still valid (i.e. no exception has been raised) then the hot-swap procedure may also exploit information extracted by the current state of the strategy to be swapped out.

Should the Behaviour Strategy Manager fail to select a pre-computed behaviour strategy, the manager requests new strategies from the layer above. This may happen, for example, because none of pre-computed strategies it manages have assumptions that are compatible with the actual observed behaviour of the system (e.g., *energy consumption is far worse than what is assumed by any pre-computed strategy*) or that they all rely on unachievable configurations (e.g. *the joint failure of the gripper component and infra-red camera was a operational scenario not considered in any of the pre-computed strategies*).

Reconfiguration Strategy Manager: This entity works similarly to its behaviour counterpart. It stores and manages multiple reconfiguration strategies and selects them for deployment constrained by the availability of instantiatable components in the Target System while maintaining consistency with the configuration requirements of the current behaviour strategy. Selection is also informed by preferences specified in the goal model. *Consequently, a precision preference may lead to selecting a reconfiguration strategy that attempts to use a GPS rather than hybrid positioning.*

When negotiating with the Behaviour Strategy Manager on a pair of strategies to be deployed, the Reconfiguration Strategy Manager takes the slave rol, informing the configurations requirements that are achievable and then selecting an appropriate reconfiguration strategy based on the selection made by the Behaviour Strategy Manger.

There are three channels that can trigger the selection of a new reconfiguration strategy. Two are similar to those that trigger the Behaviour Strategy Manager: An exception from the Reconfiguration Strategy Enactor and a change in the goal model. *Examples of these are the failure of the GPS component triggering a rapid response by the manager which selects an alternative configuration (using the hybrid positioning component) and deploys an appropriate reconfiguration strategy, or an increased response time of the GPS component leading to the decision of changing the positioning system before it (most likely) fails.* The third channel is the request of a new configuration by the Behaviour Strategy Manager (which in turn may have been triggered via de exception mechanism or a change in the goal model).

Note that deployment of new strategies at the Strategy Management layer may respond not only to problems (or forseen problems) while enacting the current strategies, but also deploy new strategies to capitalise on opportunities afforded by a change in the environment. For instance, should a new component become available, or statistics on its performance improve, (e.g. *the GPS component*) this would be reflected in the knowledge repository and an alternative preferred pre-computed strategy may be deployed.

2.7 Strategy Enactor

Responsibility: This layer's main responsibility is to execute behaviour and reconfiguration strategies provided by the layer above. Strategy execution involves monitoring the target system and invoking operations on it at appropriate times as defined by the strategy. The layer must also ensure that if the target system should reach a state unexpected by the strategy, and that consequently cannot be dealt with by the strategy, is reported to the layer above. The other key responsibility of the layer is to support both independent and transparent update of behaviour and reconfiguration strategies in addition to supporting a master-slave relation between behaviour and reconfiguration strategy execution in which the former can initiate the execution of the later.

Rationale: The aim is to provide a MAPE loop with low latency analysis to allow rapid response to changes in the state of the target system based on pre-computed strategies. In other words to achieve fast adaptation to anticipated behaviour of the target system. Allow independent handling of failed assumptions made by either the behaviour or reconfiguration strategies, thereby adapting one strategy in a way that is transparent to the other.

Structure and Behaviour: The layer has two strategy enactors, one for behaviour strategies and the other for reconfiguration strategies. Both enactors work very similarly. They monitor the target system and react to changes in the system by invoking commands on the target system. The decision of which command to execute is entirely prescribed by the strategy being enacted and requires no significant computation. The two enactors do, however, differ in the instrumentation infrastructure they use to monitor and effect the target.

Reconfiguration Strategy Enactor: This entity invokes reconfiguration commands and accesses individual software component status information through an API provided by the Target System layer. The aspects monitored and effected by this enactor are application domain independent; commands and status data are related to the component deployment infrastructure and allow operations such as adding, removing and binding components, setting operational parameters of these components and checking if they are idle, active, and so on.

In addition to sequencing reconfiguration commands, the enactor has to resolve the challenge of ensuring that state information is not lost when the configuration is modified. This can involve ensuring stable conditions such as quiescence [15] passive or quiescent before change.

Behaviour Strategy Enactor: The entity monitors and effects the target system through application domain services provided by the components of the target system via behaviour commands and event abstractions exhibited by the Target System layer. The enactor starts executing the behaviour strategy assuming that there is a configuration in place that can provide the events and commands it requires. *Thus, a new search and analyse behaviour strategy using the gripper is assuming the gripper component configured.*

Should the behaviour strategy require a different configuration at any point, it must request the configuration change explicitly. In this case a reconfigure command will be part of the behaviour strategy and the behaviour enactor will command the execution of the reconfiguration strategy stored by the reconfiguration strategy enactor *(e.g., the behaviour strategy folds the arm holding the broken gripper and then requests reconfiguration to incorporate the infra-red camera to only then proceed with in situ analysis).* Note that in this case the behaviour enactor assumes that the reconfiguration strategy is attempting to reach a target configuration that is consistent with the behaviour strategy.

Assumptions regarding the current configuration and the target configuration of the strategy loaded on the reconfiguration strategy enactor are assured by the layer above that feeds consistent behaviour and reconfiguration strategies to this layer.

3. RELATED WORK

The last decade has seen a significant build up on the body of work related to engineering self-adaptive systems. This work builds on this knowledge, emphasising the need to make behaviour and reconfiguration control first-class architectural entities. As discussed in Section 2, the architecture proposed builds on those of [16] and others. However, existing work does not provide support for both independent and also coordinated structural and behavioural adaptation at the architectural level.

The MORPH reference architecture is geared towards the use of strategies derived from the field of control engineering referred to as discrete event dynamic system (DEDS) control [4] which naturally fits over the system abstractions used at the architecture level, which is the level we envisage self-adaptation supported by our architecture to operate. DEDS are discrete-state, event-driven system of which the state evolution depends entirely on the occurrence of discrete events over time. The field builds on, amongst others, supervisory control theory [19] and reactive planning [5].

Automated construction of DEDS control strategies have been applied for self-adaptation in many different forms. For instance, in [1] temporal planning is used to produce reconfiguration strategies that do not consider structural constraints and the status of components when applying reconfiguration actions. In [22], an architecture description language (ADL) and a planning-as-model-checking are used to compute and enact reconfiguration strategies. In [7, 2, 14] automatic generation of event-based coordination strategies is applied for runtime adaptation of deadlock-free mediators. In [13], a learning technique (the L* algorithm [6]) is applied for automatically generating component's behaviour. Note that strategies do not have to be necessarily temporal sequencing of actions or commands. For instance, in [20] reconfiguration strategies used are one-step component parameter changes.

an executable modelling language for runtime execution of models (EUREMA) facilitates seamless adaptation.

4. CONCLUSIONS

An architectural approach to self-adaptive systems involves runtime change of system configuration (e.g., the system's components, their bindings and operational parameters that act as knobs) and behaviour update (e.g., components orchestration, reactive behaviour, etc). In this paper we present MORPH, a reference architecture for behaviour and configuration self-adaptation. MORPH allows both independent reconfiguration and behaviour adaptation building on the extensive work developed but also allows coordinated configuration and behavioural adaptation to accommodate for complex self-adaptation scenarios.

5. REFERENCES

[1] N. Arshad, D. Heimbigner, and A. L. Wolf. Deployment and dynamic reconfiguration planning for distributed software systems. *Software Quality Journal*, 2007.

[2] A. Bennaceur, P. Inverardi, V. Issarny, and R. Spalazzese. Automated synthesis of connectors to support software evolution. *ERCIM News*, 2012.

[3] P. Bertoli, A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. MBP: a model based planner. In *IJCAI*, 2001.

[4] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer, 2010.

[5] A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Weak, strong, and strong cyclic planning via symbolic model checking. *Artif. Intell.*, 2003.

[6] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. Learning assumptions for compositional verification. In *TACAS*, 2003.

[7] A. Di Marco, P. Inverardi, and R. Spalazzese. Synthesizing self-adaptive connectors meeting functional and performance concerns. In *SEAMS*, 2013.

[8] N. D'Ippolito, V. A. Braberman, N. Piterman, and S. Uchitel. Synthesizing nonanomalous event-based controllers for liveness goals. *TOSEM*, 2013.

[9] A. Filieri, H. Hoffmann, and M. Maggio. Automated design of self-adaptive software with control-theoretical formal guarantees. In *ICSE*, 2014.

[10] D. Garlan, S. Cheng, A. Huang, B. R. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 2004.

[11] E. Gat, R. P. Bonnasso, R. Murphy, and A. Press. On three-layer architectures. In *AIMR*, 1997.

[12] C. Ghezzi, J. Greenyer, and V. P. La Manna. Synthesizing dynamically updating controllers from changes in scenario-based specifications. In *SEAMS*, 2012.

[13] D. Giannakopoulou and C. S. Pasareanu. Context synthesis. In *SFM*, 2011.

[14] V. Issarny, A. Bennaceur, and Y. Bromberg. Middleware-layer connector synthesis: Beyond state of the art in middleware interoperability. In *SFM*, 2011.

[15] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *TSE*, 1990.

[16] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *FOSE*, 2007.

[17] P. D. L. and M. A. K. *Artificial Intelligence: Foundations of Computational Agents*. Cambridge Uni. Press, 2010.

[18] A. V. Lamsweerde. Goal-Oriented Requirements Engineering: A Guided Tour. In *RE*, 2001.

[19] P. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proc. IEEE*, 1989.

[20] J. Swanson, M. B. Cohen, M. B. Dwyer, B. J. Garvin, and J. Firestone. Beyond the rainbow: self-adaptive failure avoidance in configurable systems. In *FSE*, 2014.

[21] D. Sykes, W. Heaven, J. Magee, and J. Kramer. From goals to components: A combined approach to self-management. In *SEAMS*, 2008.

[22] H. Tajalli, J. Garcia, G. Edwards, and N. Medvidovic. Plasma: A plan-based layered architecture for software model-driven adaptation. In *ASE*, 2010.

Adaptive Predictive Control for Software Systems

Konstantinos Angelopoulos
DISI, University of Trento
Trento, Italy
angelopoulos@disi.unitn.it

Alessandro Vittorio Papadopoulos
Department of Automatic Control, Lund University
Lund, Sweden
alessandro@control.lth.se

John Mylopoulos
DISI, University of Trento
Trento, Italy
jm@disi.unitn.it

ABSTRACT

Self-adaptive software systems are designed to support a number of alternative solutions for fulfilling their requirements. These define an adaptation space. During operation, a self-adaptive system monitors its performance and when it finds that its requirements are not fulfilled, searches its adaptation space to select a best adaptation. Two major problems need to be addressed during the selection process: (a) Handling environmental uncertainty in determining the impact of an adaptation; (b) maintain an optimal equilibrium among conflicting requirements. This position paper investigates the application of Adaptive Model Predictive Control ideas from Control Theory to design self-adaptive software that makes decisions by predicting its future performance for alternative adaptations and selects ones that minimize the cost of requirement failures using quantitative information. The technical details of our proposal are illustrated through the meeting-scheduler exemplar.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design—*methodologies*

Keywords

Control theory, self-adaptive systems, software requirements, predictive control

1. INTRODUCTION

Self-adaptive systems are designed to maintain the fulfillment of their requirements in dynamic environments. When a failing requirement is encountered the system adapts by switching to an alternative configuration. The set of the available configurations constitute the system's adaptation space. Unfortunately, configuration selection is not an easy task as requirements are often conflicting and an adaptation that restores a failed requirement may break another one. For example, restoring a failed performance requirement by adding a server to a system may fail an operating costs requirement. In addition, stakeholders often over-constrain the system-to-be, or propose unrealistic unfeasible requirements [14]. Such conflicts can be accommodated by setting realistic thresholds, making a satisfactory equilibrium attainable. However, for many software systems setting accurate thresholds is at best guesswork, since there are no physical laws that account for the relationship between control parameters and requirements for an adaptive system.

Current approaches [5, 8, 21] deal with conflicting requirements and adaptation costs by making predictions about the system's environment and anticipate failures by making reconfiguration plans that optimize the utility output over time using a control theoretic technique, named Model Predictive Control (MPC), and variations of it [12]. However, these approaches are specific to resource provisioning and therefore architectural configurations, ignoring the dimensions of requirements and behavior of the adaptation space [2]. Moreover, the lack of a software engineering methodology which relates the elements of MPC and those of a self-adaptive software system prevents designers from applying this technique to domains other than service-based applications, where the current approaches focus on.

In this position paper we describe how a combination of concepts from Software Engineering (SE) and Control Theory (CT), in the same line of work as in [4], can tackle in a systematic way the problem of conflicting requirements and overestimation of system capabilities while applying adaptation strategies that maximize the system's outcome over time with minimum adaptation effort. Towards this direction we propose a combined use of MPC [12] and an online learning mechanism [10], similarly to [13, 15], to predict the future behavior of the controlled system within a specified horizon and dynamically compose adaptation strategies that will minimize the divergence of each requirement from the specified threshold prescribed by the stakeholders. Furthermore, we examine how the synthesis of a controller can be part of a SE process for designing self-adaptive systems. The adoption of MPC guarantees the avoidance of overshooting, management of constraints, and optimal tradeoff among conflicting requirements across time using prioritization techniques. In particular, we use Analytic Hierarchy Process (AHP) [1, 7].

The rest of the paper is organized as follows. Section 2 introduces the baseline of our proposal. Section 3 investigates how MPC can be applied for quantitative adaptation. Finally, Section 4 concludes the paper.

CTSE'15, August 31, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3814-1/15/08...$15.00
http://dx.doi.org/10.1145/2804337.2804340

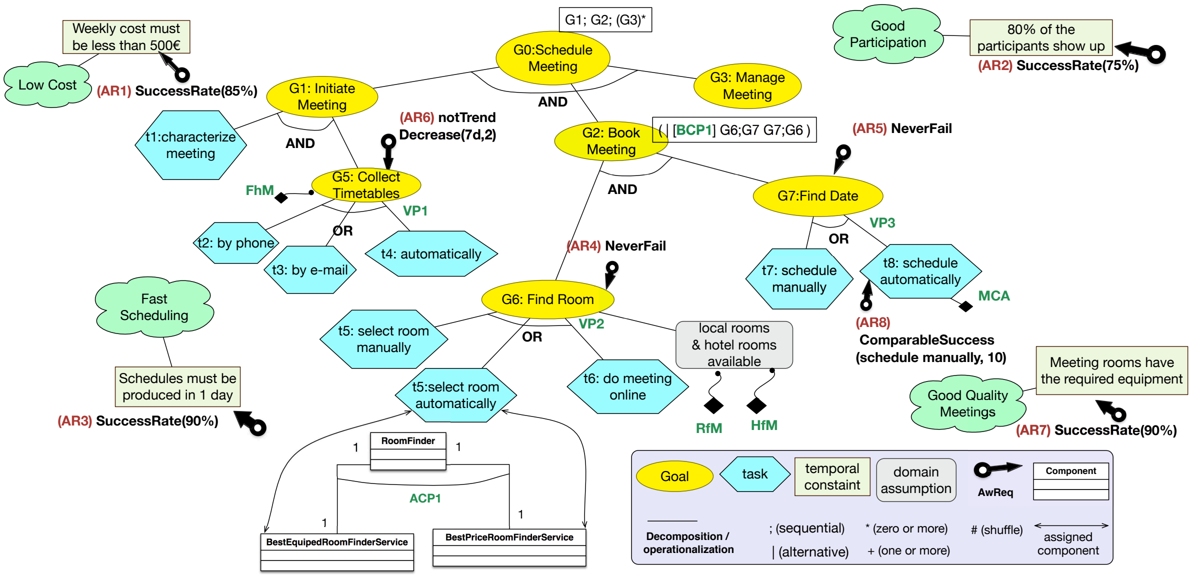

Figure 1: Three-peaks model for the Meeting Scheduler case study.

2. PRELIMINARIES AND MOTIVATING EXAMPLES

Our proposal adopts concepts from Goal-Oriented Requirements Engineering (GORE) such as goals for modeling stakeholder requirements, *softgoals* for modeling quality requirements, and AND/OR refinements that refine goal G into simpler goals whose satisfaction (all/at least one) implies the satisfaction of G, following traditional boolean semantics. Tasks are actions that a component (hardware /software components, or an external actor) can implement to fulfill or operationalize a goal. For example, in Fig. 1, *Schedule meeting* is a top-level goal, while *Good participation* is a softgoal. The goal *Schedule meeting* is AND refined into subgoals *Initiate Meeting*, *Book Meeting* and *Manage Meeting*.

In our previous work [1] we proposed a qualitative adaptation process inspired by feedback loop control. The design of this process includes three basic concepts: Awareness Requirements, Evolution Requirements and System Identification.

Awareness requirements (*AwReqs*) impose constraints on the failure of other requirements and correspond to the set-points of the adaptation mechanism. *AwReqs* are associated with variables named indicators that measure their success degree. For example, $AR4$ dictates that the goal *Find Room* must never fail, whereas $AR1$ prescribes that 85% of time the weekly cost of meetings must be less than 500 Euros. Hence, the associated indicators are $I_4 = 100\%$ and $I_6 \geq 85\%$. Every indicator is constantly monitored and when its value diverges from the one prescribed by the stakeholders, the associated *AwReq* fails and adaptation is triggered. In control-theoretical terms, indicators correspond to the system's *outputs*.

Evolution requirements (*EvoReqs*) [18] describe when and how other requirements should change at runtime. For example, an *EvoReq* may be "If requirement R fails three times in a row, replace it with requirement R^-", where R^- is a weaker (i.e., easier to fulfill) requirement. Such requirements are useful to evolve unfeasible requirements that were initially elicited from the stakeholders.

System Identification. Indicators are controlled by control parameters (*CPs*) set by the adaptation mechanism. In our recent work [2] we proposed an iterative process, named *three-peaks* to guide the software designers elicit a larger adaptation space that includes control parameters from the three dimensions of a software system, requirements, behavior and architecture. *Requirement control parameters* (*ReqCPs*) are derived either from OR-refinements or physical, information resources required by the system, in order to operate. For instance, $VP1$ in Fig. 1 is a *ReqCP* that gets its values from the ordered set $\{t2 \longrightarrow t3 \longrightarrow t4\}$. On the other hand, RfM and HfM are integer variables that represent how many local rooms and hotel rooms respectively are provided for meetings. MCA is yet another *ReqCP* that represents how many conflicts for the timeslot chosen and the participant timetables, while FhM represents *from how many* participants the system should collect time tables in order to satisfy the goal $G5$. *Behavioral control parameters* (*BCPs*) stem from system behavior, modeled with flow expressions (see Fig. 1), that capture allowed sequences of fulfillment of subgoals in order to fulfill a parent goal. For example, the goal *Book Meeting* can be fulfilled either by finding room first and then a date $(G6; G7)$ or find a date first and a meeting room afterwards $(G7; G6)$, see Fig. 1. These two potential sequences constitute the set of values for $BCP1$. Finally, Architectural Control Parameters (*ACPs*) capture variability in cases where more than one component are assigned with the fulfillment of the same goal or task, such as the task *select room automatically* which is carried out either by a component that finds the best equipped room or the another one which finds the cheapest room available. $ACP1$ gets as value the selected component's name.

Apart from *CPs*, the system's indicators are influenced by parameters found in the system's environment that cannot be controlled, named Environmental Parameters (*EPs*). Examples of such parameters include the price of hotel rooms, the response time of invited participants to timetable collection requests and their punctuality in attending the meetings after confirming their presence. *EPs* capture environmental uncertainty, since they are changing in a non-deterministic manner, adding disturbances to the system. For instance, the hotel room prices in certain periods rise, sometimes decreasing the indicator I_1 since the costs of meetings exceeds the allocated budget. Therefore, the system should respond to such changes of the environment and if possible anticipate them.

The qualitative positive or negative influence of *CPs* on controlled indicators is captured by differential relations. For instance, the differential relation $\Delta(I_2/MCA) < 0$ means that by increasing MCA by one unit I_2 will decrease, while $\Delta(I_5 MCA) > 0$ means that by increasing MCA I_5 will also increase. Similarly, the differential relations $\Delta(I_1/ACP1)$ $[BestEqip\ Room\ Service \longrightarrow BestPriceRoomService] > 0$ (the arrows indicate growing enumeration values), means switching to the component that finds the available room with the best price increases the success rate of I_1. The differential relations are symmetric and are provided by domain experts, which makes them prone to human errors and inaccuracies.

Indicators related to the same *CP* with conflicting influence, such as I_5 and I_2 from the previous example, are called conflicting indicators. When multiple failures are detected the adaptation mechanism must perform trade-offs, fixing the most important requirements first. Towards this direction, we set priorities over indicators using AHP. Due to lack of quantitative information on the impact of *CPs* on indicators, we define alternative conservative and optimistic adaptation policies to guide the trade-off process. A conservative adaptation policy forbids the adaptation mechanism from fixing a failing indicator if the value of a non-failing indicator is about to decrease. The absence of quantitative information limits the precision of the adaptation process, given that there might exist values of *CPs* that fix low priority indicator while all the other affected indicators still remain above their thresholds. On the other hand, optimistic adaptation policies allow tuning parameters that decrease indicators of higher priority requirements hoping they remain above their threshold. Again, the lack of quantitative relations and planning in the adaptation process can result in leading to failure important requirements in order to fix other, less significant ones.

3. A CONTROL-BASED APPROACH

Defining an adaptation strategy able to satisfy the most important requirements under the presence of uncertainty is not an easy task without adopting quantitative approaches. This section sketches a general control-based design procedure that can be used to accommodate this task.

A design process. As in various types of systems, some of the indicators might depend not only on the chosen value of the control parameter, but also on its past and on the values of other indicators [9, 16]. For instance, if the participation to the meetings drops and the value indicator I_2 is 60% instead of 75%, decreasing MCA, in order to fix this failure will not have immediate impact, but gradually I_2 will increase until it reaches the desired value. Such systems are called dynamic systems.

The first step is to better understand how the control parameters affect the indicators. The differential relations presented in the previous section, provide only qualitative information, and cannot be easily exploited with control techniques. A more expressive way to capture these relations is to consider the relation between control parameters $CP(\cdot) \in \mathbb{R}^m$, and indicators $I(\cdot) \in \mathbb{R}^p$, as a discrete-time linear dynamic system

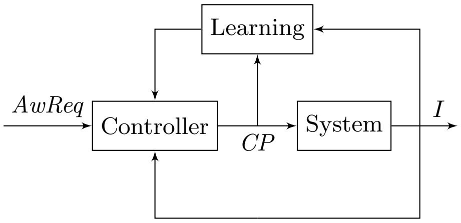

Figure 2: Reference control scheme.

$$\begin{cases} x(t+1) = A \cdot x(t) + B \cdot CP(t) \\ I(t) = C \cdot x(t) \end{cases} \tag{1}$$

where $x(\cdot) \in \mathbb{R}^n$ is the *state* of the system—notice that the state might not have a meaningful interpretation, but it is functional to defining in a more compact form the relation between $CP(\cdot)$ and $I(\cdot)$. Since the system has m inputs, and p outputs, it is a Multiple-Input and Multiple-Output (MIMO) system.

The matrices (A, B, C) describe the dynamics of the system, and can be identified from experimental data through system identification techniques for MIMO systems, e.g., subspace identification methods see [10, 19, 20]. Note that in case A is a matrix of all zeros, the system is characterized as static. However, our MPC is also applicable to static MIMO systems. This analytical model can be used to predict the future behavior of the system over a finite time horizon, and as such is also referred as *prediction model*. Having identified the model of the system (1), the control scheme of Fig. 2 can be set up.

The next step is to design a control mechanism, therefore, decide what type of "Controller" to use [4, 13, 15]. MPC is a natural choice for the problem-at-hand for various reasons [12]. First of all, it is naturally formulated for MIMO systems. There are generally many control parameters, and indicators that need to be controlled. Moreover, both indicators and control parameters have upper and lower bounds that the decision-making strategy should take into account. MPC allows one to formulate the problem as the minimization of a functional subject to given constraints as follows:

$$\begin{aligned} \text{minimize}_{CP_{t+k}} \quad & \sum_{k=0}^{N-1} J(AwReq_{t+k}, I_{t+k}, CP_{t+k}) \qquad (2) \\ \text{subject to} \quad & I_{\min} \leq I_{t+k} \leq I_{\max} \\ & CP_{\min} \leq CP_{t+k} \leq CP_{\max} \\ & x_{t+k+1} = A \cdot x_{t+k} + B \cdot CP_{t+k} \\ & I_{t+k} = C \cdot x_{t+k} \\ & x_t = x(t), \quad k = 0, \ldots, N-1. \end{aligned}$$

The optimization problem (2) is solved over a finite horizon of N steps ahead, thanks to the prediction model (1). The solution of the optimal control problem is a plan of future control parameter values $CP^{\star}_t, \ldots, CP^{\star}_{t+N-1}$. Only the first element of this plan is applied, i.e., $CP(t) = CP^{\star}_t$. At time $t+1$, a new optimization problem is solved analogously.

It is important to notice that the cost function has to be designed according to the specific domain. A common choice is:

$$J(AwReq_t, I_t, CP_t) = \sum_i q_i \left(AwReq_{t,i} - I_{t,i}\right)^2 + \sum_j r_j CP^2_{t,j}$$

where $q_i \geq 0$ and $r_j > 0$. The values q_i are weighting the error between the setpoint and the output, using the requirement prioritization result of AHP. On the other hand, the values r_j represent penalties of using one control parameter over others, and it can be chosen according to the specific domain [3]. Finally, minimizing the cost function is interpreted as an effort by the adaptation mechanism to anticipate requirement failures using analytical prediction models, giving priority to those of higher importance, while minimize the aggregate penalties of the planned adaptation strategy. Whenever, it is unfeasible to eliminate completely within the time horizon all the failures, *EvoReqs* can be used to weaken the failure-insisting requirements, lowering their thresholds.

Apparently, the quality of the obtained solution depends on the accuracy of the extracted prediction model. It is possible that the dynamics relating the inputs and the outputs of the system are changing over time, or that the linear model (1) is not able to capture more complex dynamics of the system. Therefore, the "Learning" block, in the control scheme in Fig. 2 is in charge to monitor the input-output relation of the system, and update the model that is used in the control mechanism according to the actual behavior of the system. The learning mechanism can be based on many different algorithms, ranging from recursive least squares to recursive subspace identification [6, 11]. Independently of the learning mechanism, the "Learning" block is in charge of updating online the model adopted by the MPC algorithm to predict the future behavior of the system.

The proposed solution is able to cope with environmental uncertainty, while overcomes the limitation of having qualitative information coming from domain experts. In principle, designers can collect data on the input-output behavior of the system and therefore to identify a reasonably accurate linear model describing the dynamics of the system. The control algorithm exploits such a model for planning a suitable adaptation strategy, named plan of control signals in control-theoretical terms. Finally, the learning mechanism is in charge of updating the prediction model whenever it behaves differently than expected.

It is worth noticing that the proposed control scheme is generic in that the "Controller" and "Learning" blocks can be realized in terms of any controller or learning algorithm respectively. For our case, tools that combine optimization and satisfiability modulo theories, e.g. [17], can handle also boolean constraints to the MPC optimization problem. Such constraints are "If timetables are collected automatically, then the schedule is made automatically as well" (see Fig. 1).

The main limitation of the approach comes whenever new requirements are added to the problem. In that case, there are two possibilities. First, new indicators (i.e., new outputs) are introduced to the system and second new control parameters (i.e., inputs) are added. In both cases, the whole design procedure needs to be repeated, since no control-based technique is able to manage structural changes in the systems.

4. CONCLUSIONS AND FUTURE WORK

In this position paper we investigated the problem of designing an adaptation mechanism for MIMO software systems that operate within environmental uncertainty. We used meeting-scheduler as an exemplar to demonstrate the weaknesses of qualitative adaptation and the need of analytical models for better adaptation.

We address this problem by proposing the use of MPC. This type of control uses analytical models that describe the system's behavior allowing to forecast future failures in our requirements and anticipate them in an optimal way with respect to their priorities.

Finally, we plan to implement an MPC controller and experiment with simulations of the meeting-scheduler and other case studies to further evaluate our proposal.

5. ACKNOWLEDGMENTS

This work has been supported by the ERC advanced grant 267856 Lucretius: "Foundations for Software Evolution" (April 2011 - March 2016, http://www.lucretius.eu). This work was partially supported by the Swedish Research Council (VR) for the projects "Cloud Control", and through the LCCC Linnaeus and ELLIIT Excellence Centers.

6. REFERENCES

[1] K. Angelopoulos, V. E. S. Souza, and J. Mylopoulos. Dealing with multiple failures in zanshin: a control-theoretic approach. In *SEAMS 14*, pages 165–174. ACM, 2014.

[2] K. Angelopoulos, V. E. S. Souza, and J. Mylopoulos. Capturing variability in adaptation spaces: A three-peaks approach. In *Conceptual Modeling – ER 2015*. Paul Johannesson and Mong Li Lee and Stephen W. Liddle and Oscar Pastor, 2015 (to appear).

[3] A. E. Bryson and Y.-C. Ho. *Applied Optimal Control: Optimization, Estimation and Control*. Taylor & Francis, 1975.

[4] A. Filieri, M. Maggio, K. Angelopoulos, N. D'Ippolito, I. Gerostathopoulos, A. B. Hempel, H. Hoffmann, P. Jamshidi, E. Kalyvianaki, C. Klein, F. Krikava, S. Misailovic, A. V. Papadopoulos, S. Ray, A. M. Sharifloo, S. Shevtsov, M. Ujma, and T. Vogel. Software engineering meets control theory. In *10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 71–82, 2015.

[5] H. Ghanbari, M. Litoiu, P. Pawluk, and C. Barna. Replica placement in cloud through simple stochastic model predictive control. In *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*, pages 80–87, June 2014.

[6] I. Houtzager, J.-W. Wingerden, van, and M. Verhaegen. Fast-array recursive closed-loop subspace model identification. In *15th IFAC Symposium on System Identification*, volume 15, pages 96–101, 2009.

[7] J. Karlsson and K. Ryan. A cost-value approach for prioritizing requirements. *Software, IEEE*, 14(5):67–74, Sep 1997.

[8] D. Kusic, J. Kephart, J. Hanson, N. Kandasamy, and G. Jiang. Power and performance management of virtualized computing environments via lookahead control. In *Autonomic Computing, 2008. ICAC '08. International Conference on*, pages 3–12, June 2008.

[9] A. Leva, M. Maggio, A. V. Papadopoulos, and F. Terraneo. *Control-based operating system design*. Control Engineering Series. IET, 2013.

[10] L. Ljung. *System Identification: Theory for the User*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[11] M. Lovera, T. Gustafsson, and M. Verhaegen. Recursive subspace identification of linear and non-linear wiener state-space models. *Automatica*, 36(11):1639–1650, 2000.

[12] J. Maciejowski. *Predictive Control: With Constraints*. Prentice Hall, 2002.

[13] M. Maggio, H. Hoffmann, A. V. Papadopoulos, J. Panerati, M. D. Santambrogio, A. Agarwal, and A. Leva. Comparison of decision making strategies for self-optimization in autonomic computing systems. *ACM Transactions on Autonomous and Adaptive Systems*, 7(4):36:1–36:32, 2012.

[14] B. Nuseibeh. Conflicting requirements: When the customer is not always right. *Requirements Engineering*, 1(1):70–71, 1996.

[15] A. V. Papadopoulos, M. Maggio, S. Negro, and A. Leva. General control-theoretical framework for online resource allocation in computing systems. *IET Control Theory & Applications*, 6(11):1594–1602, 2012.

[16] A. V. Papadopoulos, M. Maggio, F. Terraneo, and A. Leva. A dynamic modelling framework for control-based computing system design. *Mathematical and Computer Modelling of Dynamical Systems*, 21(3):251–271, 2015.

[17] R. Sebastiani and P. Trentin. Optimathsat: A tool for optimization modulo theories. In *In proc. Int. Conf. on Computer Aided verification, CAV'15*, 2015 (to appear).

[18] V. E. Souza, A. Lapouchnian, K. Angelopoulos, and J. Mylopoulos. Requirements-driven software evolution. *Comput. Sci.*, 28(4):311–329, Nov. 2013.

[19] G. van der Veen, J.-W. van Wingerden, M. Bergamasco, M. Lovera, and M. Verhaegen. Closed-loop subspace identification methods: an overview. *Control Theory Applications, IET*, 7(10):1339–1358, July 2013.

[20] M. Verhaegen and V. Verdult. *Filtering and System Identification: A Least Squares Approach*. Cambridge University Press, New York, NY, USA, 2012.

[21] Q. Zhang, Q. Zhu, M. Zhani, and R. Boutaba. Dynamic service placement in geographically distributed clouds. In *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*, pages 526–535, June 2012.

Quo Vadis Cyber-Physical Systems

Research Areas of Cyber-Physical Ecosystems

A Position Paper

Christian Bartelt
Clausthal University of Technology
Department of Informatics
38678 Clausthal-Zellerfeld, Germany
christian.bartelt@tu-clausthal.de

Andreas Rausch
Clausthal University of Technology
Department of Informatics
38678 Clausthal-Zellerfeld, Germany
andreas.rausch@tu-clausthal.de

Karina Rehfeldt
Clausthal University of Technology
Department of Informatics
38678 Clausthal-Zellerfeld, Germany
karina.rehfeldt@tu-clausthal.de

ABSTRACT

Many *technological innovations* from the research area of dynamic adaptive systems or IT ecosystems are already established in current software systems. Especially cyber-physical systems should benefit by this progress to provide smart applications in ambient environments of private and industrial space. But a proper and *methodical engineering* of cyber-physical ecosystems (CPES) is still an open and important issue. Traditional software and systems engineering facilities (system models, description languages, or process models) do not consider fundamental characteristics of these ecosystems as openness, uncertainty, or emergent constitution at runtime sufficiently. But especially these aspects let blur the line of system boundaries at design time. The diverse components of CPES have essential impacts on the engineering of CPES as well, concerning time synchronizing, execution control, and interaction structure. Self-balanced control in CPES promises new application possibilities, but also needs new engineering techniques concerning the overall engineering process, including requirements engineering and runtime verification. In this position paper we survey and summarize the dimensions of challenges in applying control theory for the engineering of cyber-physical ecosystems.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architecture

General Terms

Design, Reliability, Security, Languages, Theory

Keywords

Cyber-physical systems, software ecosystems, systems engineering, control theory, self-balanced control, system-of-systems

1. INTRODUCTION

The term cyber-physical system (CPS) is widely used nowadays. One of the most common definition of cyber-physical systems is a system which connects physical and virtual processes, where bidirectional information flows are significant[9]. The physical processes are controlled and monitored by computations. Traditionally, this kind of systems is engineered based on the automation system pyramid by the Totally Integrated Automation concept[13]. This classic architecture spans from field level to management level by a hierarchical control scheme. The automation system pyramid is inflexible and not suited for adaptive systems. Adaptivity in this context means, that a system is able to change its structure autonomously to adapt to a new environment or certain situation. By decomposing each layer of the automation system pyramid into components and allowing communication between arbitrary components, CPS become more flexible. As a result, the engineering of CPS deals with several challenges regarding the self-organization and adaptivity.

Openness yields benefits for CPS but also requires new research. In this case, openness refers to a system with high interoperability and open interfaces. We introduce the term cyber-physical ecosystems (CPES) for adaptive and open cyber-physical systems. These systems combine cyber-physical systems with an approach for interacting with other systems. The control of such highly dynamic systems have been researched in the field of software ecosystems for several years[6, 11, 12]. But the transfer of self-balanced control mechanisms to realize CPES raises new ambitious challenges in engineering. We aim to name some of these challenges in this position paper. To do so, we first inspect CPS and software ecosystems in three different engineering challenges areas: the overall approach, the design of components and the runtime operation. The overall approach for designing and running systems covers the whole development cycle as well as the runtime environment. The design of components concentrates on design time while the last area covers the runtime of systems. CPES combine concepts from CPS and software ecosystems. Therefore, we derive challenges for CPES by combining challenges for both system classes.

2. STATE OF THE ART

By allowing more flexibility in the architecture of CPS, new CPS applications will become possible. In this section, we first present engineering challenges of CPS and afterwards of software ecosystems.

2.1 Cyber-Physical Systems

A cyber-physical system is a system-of-systems. In a CPS, information systems cooperate with control systems. Information systems are socio-technical systems for information processing whereas control systems control physical processes. By cooperating, the control systems compensate the missing sensors and actuators of information systems. In turn, the information systems provide data analysis and storage for the control systems.

CTSE'15, August 31, 2015, Bergamo, Italy
ACM. 978-1-4503-3814-1/15/08...$15.00
http://dx.doi.org/10.1145/2804337.2804341

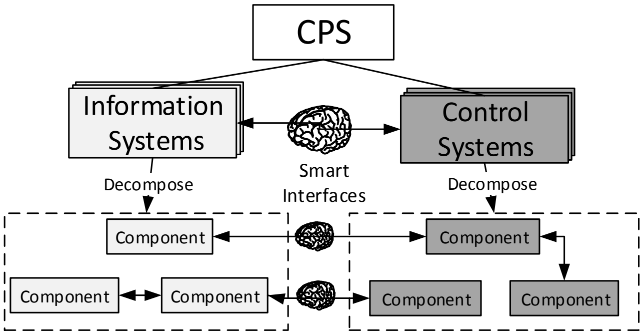

Figure 1: CPS (decomposed) with smart interfaces

There has to be an intelligent interface between the two kinds of systems (cf. Figure 1). On the one hand, control systems are often continuous and realtime-capable and interact with the physical world. On the other hand, information systems are discrete and have no natural concept of time but can operate on huge amounts of data. These different properties can only be joined by an intelligent interface.The special nature of CPS is adaptivity. Adaptivity in CPS is achieved by decomposing each information and control system in components and allowing data exchange between arbitrary components (cf. Figure 1). The interface between the components deals with the same different characteristics as before. But it can adapt to certain situations by composing the components in a different way. Those kinds of CPS will pose new challenges to science and research[2]:

How can an overall development and operation approach be established during the whole life-cycle regarding adaptivity of the systems, learning of functions, self-organization and more?

CPS consist of networked information systems and control systems. To manage complexity, information systems and control systems are decomposed in modular building blocks called components. Consequently, the intelligent interface between the information systems and the control systems are also decomposed to fine grained smart interfaces between the components of the information systems and the components of the control systems. Those local and small interfaces are smart enough to support adaptivity, learning of functions and self-organization.

How should design and development methods look like to consistently expand the concepts of system engineering in such a way that it can also be used for cyber-physical systems?

Heterogeneously networked structures like CPS require an integral systemic view and interdisciplinary cooperation between mechanical engineering, electrical engineering and computer science. Therefore an interdisciplinary multi-view and multi-level supporting modelling and development approach is required. It is necessary to prepare discipline-specific approaches for integration into CPS. Thus handling complexity and the realization of new functionalities through the adaptivity of the systems and the combination of functions will be at the forefront.

How should cyber-physical systems - that are more adaptive and open systems but still have high dependability requirements to be guaranteed - be deployed, operated, monitored and maintained?

CPS directly influence the physical world. Therefore an operation approach is required that is able to control and thereby prevent damage of the CPS and even more important its environment. The damage should not only be prevented in case of errors and failures but particularly in case of incorrect behavior and adaption of the CPS.

There are already different modelling approaches for CPS. Some of the challenges of modeling CPS are summarized in [4]. The work also lists some promising new approaches. The close integration of embedded systems and the physical world is often modelled with hybrid systems. In [8] hybrid automata are used for verification of cyber-physical systems. There are also approaches for component-based design of hybrid systems as well as methods for checking whether a hybrid systems satisfies a specification[3, 5].

2.2 Software Ecosystems

Complex software ecosystems are complex, compound system consisting of interacting individual adaptive systems, which are adaptive as a whole, based on engineered adaptability. The different life cycles of the individual adaptive systems must also be taken into consideration [12].

A complex software ecosystem (CSE) comprises of individual adaptive systems whose behavior and interactions change over time. These changes are usually not planned centrally, but arise from independent processes and decisions within and outside the CSE. For example, a slippery road warning system of a car requests for information about road conditions without any knowledge about relevant information providers in the ecosystem. The warning system could connect to totally different systems, like a weather forecast provider as well as an electronic accessible plan of the snow plowing service or even the ABS monitoring of cars in same area, to receive the relevant information.

In addition, CSE are mixed human-machine artifacts: human beings in the complex software ecosystem interact with the individual systems, and in this way they become an integral, active part of the CSE. Therefore, human requirements, goals, and behavior must be considered when designing a CSE, by modeling them as active system components. A number of ambitious challenges follow from the mentioned characteristics of software ecosystems. These can be divided in three areas – balancing evolution process between system (local) and ecosystem (global), distributed and independent engineering at design time, and technology support for system composition at runtime.

How can system development and system operation be integrated into a close linked and balanced approach for software ecosystem evolution?

The development and evolution of systems in software ecosystems is characterized by interplay of classical model-based engineering and constraint-based development to consider external restrictions from the ecosystem. As already mentioned, a CSE consists of a set of individual adaptive systems. To restrict the individual adaptive behavior of the adaptive systems, local constraints might be added and enforced locally under closed world assumption. In addition, institutional and improvement constraints from an ecosystem's communities or covering common ecosystem's objectives and guidelines are added—following an open-world semantic. All of these constraints can be used to validate the individual models of the adaptive systems but also to validate the union of models of all adaptive systems resp. the ecosystem's models. Therefore constraint validations of individual systems, but also of the interaction between these adaptive systems, can be identified during design time. However, as changes in these systems are not planned centrally, but arise from independent processes and decisions, adaptivity cannot be completely controlled during design time. Consequently, we also

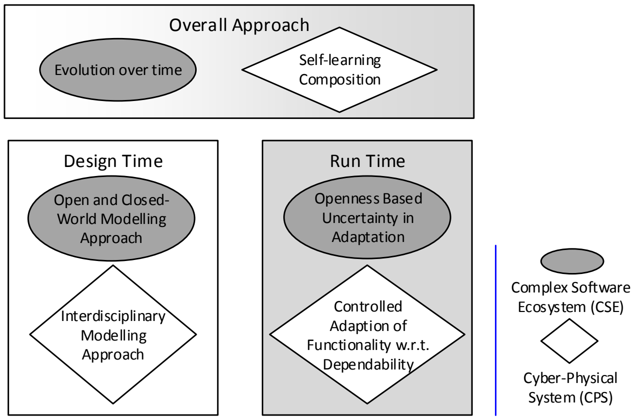
Overall Approach
Evolution over time
Self-learning Composition
Design Time
Open and Closed-World Modelling Approach
Interdisciplinary Modelling Approach
Run Time
Openness Based Uncertainty in Adaptation
Controlled Adaption of Functionality w.r.t. Dependability
Complex Software Ecosystem (CSE)
Cyber-Physical System (CPS)

Figure 2: Combined challenges in engineering of CPES

have to take the runtime into account. Therefore, the constraints provide a knowledge transfer between design time and run time. Constraints are additionally monitored and enforced during run time.

How should distributed engineering of the constituent systems with different life-cycles and not centrally managed and coordinated be organized during design time?

As consequence of openness of software ecosystems as well as uncertainty of their system environments, and the independency and distribution of development processes follow several challenges. For example, established design methods using software modeling under the closed-world assumption (CWA) with description techniques based on the open-world assumption (OWA) have to be combined. The application of OWA during design is necessary because the demand for the specification of system parts while only incomplete information about the whole system at runtime is available[11]. Especially software interfaces of independently developed components have to be connected semantically dependable. Because of the distributed development of system parts by independent actors a syntactic connection of interfaces is undependable in general[7]. Furthermore, adaptability demands have to be engineered at design time.

How can self-adaptation of the software ecosystem evolution be balanced during runtime?

The systems composition in CSE moves from design to runtime. This is a profound changed paradigm which requires answers to several challenges. One central question is, how can technology support the dynamical adaptivity resp. the emergent composition of systems to facilitate a self-balanced ecosystem at runtime? The adaptability of a software ecosystems needs a common technological infrastructure to manage the joining components and their connections as well as additional services [10]. The technology has to provide an autonomous mechanism which can balance the concurrent needs of systems in a CSE. Systems of a software ecosystem compete for common shared resources. For the balancing of the competitive demands of systems in software ecosystems an appropriate mechanism has to be implemented by the infrastructure [1]. Furthermore the semantic correctness of the autonomous system integration and system requirements has to be validated during operation and design time.

3. RESEARCH AREAS IN ENGINEERING OF CPES

CSE are complex adaptive systems of adaptive systems and human beings. Thereby software ecosystems are open systems with respect to the independent evolution of the constituent adaptive systems, the dynamic self-adaption mechanisms of the constituent adaptive systems themselves, and the active human beings within the CSE. Hence software ecosystems come with a high degree of uncertainty due to their openness and adaptivity. To manage uncertainty additional knowledge is elaborated and used during design time and run time. In addition an overall evolutionary development approach is provided by software ecosystems.

CPS merge together embedded software-intensive control systems and global networked internet-based information systems by modular, small, intelligent, and non-hierarchical interfaces between the components of the control system and the information system. Adaptivity in CPS is based on functional adaption of the control system with respect to high dependability issues. Interdisciplinary and holistic engineering approaches are applied to provide new functionalities through an intelligent new connection between existing control components and information components.

In CPES we integrate the openness and adaptivity of software ecosystems with the modular and intelligent component coupling of CPS. Thereby new, combined research areas arise. We deduce these research areas from the combination of the challenges in Section 2. The research areas cover A) the overall approach and evolution of the system, B) the design time development approach as well as C) the operation approach during runtime of the system (cf. Figure 2).

A) How should the controlled evolutionary development approach (CSE) be combined with the long-term self-adaption mechanism by the intelligent interfaces (CPS)?

In classical software engineering, the design starts with main requirements. These requirements are step-wise refined to hierarchical requirements. After design time the system is not intended to change. A new feature for the system will be a new project. This classical approach is not applicable for CPES.

In CPES integration and validation is done during runtime. Moreover the system is growing and changing dynamically over time. The system itself adapts to its environment. Once the system is no longer able to adapt itself, new data for machine learning based long-term self-optimization are provided by smart intelligent sensor data to fulfill the changing requirements and expectations after self-optimization. In addition new requirements might be derived for the system itself but also for components and physical processes realizing their own specific functionalities. These requirements may contradict each other – for example two machines with common resources which both aims at 100% occupancy rate. Therefore, global requirements exist as well. Each time a new component joins the system it has to be adjusted to fit to the global requirements.

Ensuing from this situation, technical challenges can be derived: The components and the host infrastructure of CPES need an advanced configuration service compared to CSE. This (distributed) service has to able to receive and value feedback from the physical environment. Further it has to balance competitive requirements (possibly by market mechanisms) and has to consider appropriate migration strategies.

B) How can an open and closed world modeling approach (CSE) be integrated with an interdisciplinary modeling approach (CPS)?

Traditional software development approaches offer various techniques to support software engineers. One of the most fundamental is the use of models and modeling. Depending on what is considered relevant to a system under development at any given point, various modeling concepts and notations may be used to highlight one or more particular perspectives or views of that system. It is often necessary to convert between different views at an equivalent level of abstraction facilitated by model transformation, e.g. between a structural view and a behavioral view.

CPES combine the virtual world represented by information systems and the physical world influenced by control systems. Consequently our modeling approach must be able to represent the various engineering disciplines for software to electrical up to mechanical engineering. Furthermore, as CPES are evolving in a not centrally planned and managed manner we have to use open and closed world models combining models describing the system under construction as well as overall ecosystem constraints that have to be guaranteed by all parts of the CPES in an interdisciplinary integrated approach.

The model-based design methods in the well-established engineering disciplines of physical systems (electrical/mechanical engineering) assume traditionally a closed world paradigm. For a consideration of openness in CPES, these system description languages have to be enhance by a support of not only inter-disciplinary modeling but also by additional expressions which allow to specify semantics based on the open-world assumption.

C) How can the openness based uncertainty in system adaption (CSE) be balanced with the need to control system adaption with respect to dependability issues (CPS)?

After a system has been designed, verified and developed, it will to be deployed and executed within the ecosystem. In order to be useful over time, systems must be able to adapt themselves to changing needs, goals, requirements or environmental conditions as autonomously as possible. Three levels of adaptability, namely: engineered adaptability, emergent adaptability and evolutionary adaptability have to be supported. During runtime, various aspects have to be considered in order to enable and control those kinds of adaptability. Therefore the MAPE-K loop is a typical well-known architectural blueprint for such a system.

As CPES manipulate and influence the physical world we have to guarantee the CPES does not damage or hurt its environment. CPS often realize safety-critical applications. Therefore, they underlie strict requirements like safety, security, privacy or realtime controlling. In classical applications the system behavior can be assured at design time. For CPES the existing approaches have to be enhanced to detect all possible dependability problems in advance and reorganize the system to prevent the environment from possible damage. Prediction techniques have to be elaborated to guide and guarantee the required dependability issues of the CPES during run time. Moreover the dependability issues might change over time and thus the guarantees have to be adapted dynamically during run time. To ensure the validity of dependability/safety conditions, the technical platform of CPES has to provide mechanisms to predict physical effects (simulation/test techniques). This is necessary for an estimation and valuation of the future behavior of systems before possibly irreversible or safety-critical effects are implemented in the physical environment. Considering that the available configuration mechanisms of CSE has to be enhance to ensure a safe operation of autonomous composed CPS.

4. CONCLUSION

In this paper we presented a new point of view on CPS. CPS will be more open and complex in the future. This will require transfer of self-balanced control mechanisms to CPES. We presented the current state of the art in both CSE and CPS. These different works may be considered when facing the challenges of CPES we presented.

5. REFERENCES

[1] Bartelt, C., Fischer, B. and Rausch, A. 2013. Towards a Decentralized Middleware for Composition of Resource-Limited Components to Realize Distributed Applications.

[2] Cyber-Physical Systems. Driving force for innovation in mobility, health, energy and production: http://www.acatech.de/de/publikationen/publikationssuche/detail/artikel/cyber-physical-systems-innovationsmotor-fuer-mobilitaet-gesundheit-energie-und-produktion.html. Accessed: 2015-06-15.

[3] Damm, W., Möhlmann, E. and Rakow, A. 2014. Component Based Design of Hybrid Systems: A Case Study on Concurrency and Coupling. Proceedings of the 17th International Conference on Hybrid Systems: Computation and Control (New York, NY, USA, 2014), 145–150.

[4] Derler, P., Lee, E.A. and Sangiovanni-vincentelli, A.L. 2011. Addressing Modeling Challenges in Cyber-Physical Systems.

[5] Henzinger, T.A. and Otop, J. 2014. Model Measuring for Hybrid Systems. Proceedings of the 17th International Conference on Hybrid Systems: Computation and Control (New York, NY, USA, 2014), 213–222.

[6] Herold, S., Klus, H., Niebuhr, D. and Rausch, A. 2008. Engineering of IT Ecosystems: Design of Ultra-large-scale Software-intensive Systems. Proceedings of the 2Nd International Workshop on Ultra-large-scale Software-intensive Systems (New York, NY, USA, 2008), 49–52.

[7] Kolatzki, S., Goltz, U., Hagner, M., Rausch, A. and Schindler, B. 2012. Automated Verification of Functional Interface Compatibility. 01 (2012).

[8] Krishna, S.N. and Trivedi, A. 2013. Hybrid Automata for Formal Modeling and Verification of Cyber-Physical Systems. Journal of the Indian Institute of Science. 93, 3 (Sep. 2013), 419–440.

[9] Lee, E.A. 2010. CPS Foundations. Proceedings of the 47th Design Automation Conference (New York, NY, USA, 2010), 737–742.

[10] Niebuhr, D., Klus, H., Anastasopoulos, M., Koch, J., Weiß, O. and Rausch, A. 2007. DAiSI -- Dynamic Adaptive System Infrastructure. Fraunhofer Institut für Experimentelles Software Engineering.

[11] Rausch, A., Bartelt, C., Herold, S., Klus, H. and Niebuhr, D. 2013. From Software Systems to Complex Software Ecosystems: Model- and Constraint-Based Engineering of Ecosystems. Perspectives on the Future of Software Engineering. J.M. Schmid, ed. Springer. 61–80.

[12] Rausch, A., Muller, J.P., Niebuhr, D., Herold, S. and Goltz, U. 2012. IT ecosystems: A new paradigm for engineering complex adaptive software systems. (Jun. 2012), 1–6.

[13] Totally Integrated Automation - Totally Integrated Automation - Siemens: http://www.industry.siemens.com/topics/global/en/tia/Pages/default.aspx. Accessed: 2015-06-08.

Robust Degradation and Enhancement of Robot Mission Behaviour in Unpredictable Environments[*]

Nicolas D'Ippolito[1], Victor Braberman[1], Daniel Sykes[2], Sebastián Uchitel[12]
[1] Departamento de Computación, FCEN, Universidad de Buenos Aires, Argentina
[2] Department of Computing, Imperial College London, UK

{ndippolito,vbraber}@dc.uba.ar, {daniel.sykes,suchitel}@imperial.ac.uk

ABSTRACT

Temporal logic based approaches that automatically generate controllers have been shown to be useful for mission level planning of motion, surveillance and navigation, among others. These approaches critically rely on the validity of the environment models used for synthesis. Yet simplifying assumptions are inevitable to reduce complexity and provide mission-level guarantees; no plan can guarantee results in a model of a world in which everything can go wrong. In this paper, we show how our approach, which reduces reliance on a single model by introducing a stack of models, can endow systems with incremental guarantees based on increasingly strengthened assumptions, supporting graceful degradation when the environment does not behave as expected, and progressive enhancement when it does.

Categories and Subject Descriptors

D.2 [**Software Engineering**]

General Terms

Design

Keywords

Self-adaptive Systems, Controller Synthesis

1. INTRODUCTION

Controller synthesis and planning approaches based on temporal logic have proven useful for generating discrete event-based robot behaviours from high-level specifications (e.g. [4, 30, 29]). Such approaches rely on finite-state models that purport to represent the operating environment and how the robot can interact with it. However, any such model is by definition an abstraction of the real environment and its dynamics, and any such model entails a risk that it is not a true representation of the environment as encountered at runtime. In some scenarios, this risk, when materialised, may lead to catastrophic failure of the mission.

One means to cope with this uncertainty [12] is to use machine learning techniques that revise (or indeed generate from scratch) the models on which synthesis relies so that, over a period of time, the models converge upon a "realistic" description of the environment [27, 11, 14]. One drawback of using such techniques is the computational cost of learning, and the delay before the mission can begin in earnest, which may be prohibitive in some domains (e.g. safety-critical systems). Another drawback is that the learned model may be of such complexity that synthesis becomes computationally infeasible, and in the worst case nothing can be guaranteed in a world where anything can go wrong. There is therefore a benefit in having an element of manual abstraction involved in synthesising robotic behaviours. To that end, we have proposed an approach [7] in which models at different levels of abstraction are used to synthesise a controller capable of gracefully degrading its guarantees when the runtime environment diverges from one of the more abstract models, and progressively enhancing its guarantees when the environment behaves as envisaged in the more idealised models.

Our approach uses a stack of models where higher models are more idealised and can be simulated by the lower models. A mission requirement is associated with each tier of the stack. Higher tiers allow to produce controllers guaranteeing stronger requirements, while lower tiers only allow for controllers with weaker requirements because of their more realistic description of the environment dynamics. Each tier of the stack can be regarded as an independent controller synthesis problem, but our approach combines the resulting controllers in such a way that a failure in a higher controller can be handled by a graceful degradation to the controller of a lower tier, resulting in a lower guaranteed 'service level'. Likewise, if the environment conforms to a higher tier, we may attempt to synthesise a controller for a higher tier and so enhance the guaranteed service level.

In this paper, we show how synthesised controller stacks can be used to provide robust behaviour for robot missions from high-level temporal logic specifications. We apply it to an existing case study involving a robot engaged in a surveillance mission [28] and show how, in addition to automatic synthesis for cyclic missions (i.e. missions in which the goals are achieved infinitely many times, our approach enables the robot to handle invalid environment models. In

[*]This work was partially supported by grants ERC PBM-FIMBSE, ANPCYT PICT 2012-0724, UBACYT W0813, ANPCYT PICT 2011-1774, UBACYT F075, CONICET PIP 11220110100596CO, MEALS 295261.

CTSE'15, August 31, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3814-1/15/08...$15.00
http://dx.doi.org/10.1145/2804337.2804342

other words, in this paper we report on an application of our previous technique to a known case study to evaluate its applicability and asses some of its properties.

In Section 3 we give an overview of our approach and the guarantees it makes. Section 4 describes our particular implementation of the general approach and how it achieves the general requirements. Sections 5 and 6 discuss how the approach has been applied to an existing surveillance case study. Finally, Section 7 comments on some of the related work, before Section 8 concludes.

2. BACKGROUND

We start with labelled transition systems a canonical representation of reactive components and systems.

Definition 2.1. (Labelled Transition Systems) *A* Labelled Transition System *(LTS) is $E = (S, A, \Delta, s_0)$, where S is a finite set of states, A is its* communicating alphabet*, $\Delta \subseteq (S \times A \times S)$ is a transition relation, and $s_0 \in S$ is the initial state. We denote $\Delta(s) = \{\ell \mid (s, \ell, s') \in \Delta\}$ and $\Delta(s, \ell) = \{s' \mid (s, \ell, s') \in \Delta_E\}$. A trace of E is $t{=}\ell_0, \ell_1, \ldots$, where for every $i \geq 0$ we have $(s_i, \ell_i, s_{i+1}) \in \Delta$, We denote the set of traces of E by $\mathrm{Tr}(E)$. We say that an LTS is deterministic if (s, ℓ, s') and (s, ℓ, s'') are in Δ implies $s' = s''$.*

Reactive systems are built as compositions of multiple reactive components. Such composition is formalised as follows:

Definition 2.2. (Parallel Composition) *Let $M = (S_M, A_M, \Delta_M, s_{M_0})$ and $E = (S_E, A_E, \Delta_E, s_{E_0})$ be LTSs. The* Parallel Composition *($\|$) is a symmetric operator such that $E\|M$ is the LTS $E\|M = (S_E \times S_M, A_E \cup A_M, \Delta, (s_{E_0}, s_{M_0}))$, where Δ is the smallest relation that satisfies the rules below, where $\ell \in A_E \cup A_M$:*

$$\frac{(s,\ell,s')\in\Delta_E}{((s,t),\ell,(s',t))\in\Delta}\ \ell \in A_E \setminus A_M \qquad \frac{(t,\ell,t')\in\Delta_M}{((s,t),\ell,(s,t'))\in\Delta}\ \ell \in A_M \setminus A_E$$

$$\frac{(s,\ell,s')\in\Delta_E,\ (t,\ell,t')\in\Delta_M}{((s,t),\ell,(s',t'))\in\Delta}\ \ell \in A_E \cap A_M$$

We restrict attention to states in $S_E \times S_M$ that are reachable from the initial state (s_{E_0}, s_{M_0}) using transitions in Δ.

There are various restrictions that can be imposed on the LTS to be composed using parallel composition. These restrictions vary in order to adequately capture different interaction models. We are interested in reasoning about what a controller can achieve in a possibly adversarial environment. Hence, the distinction between actions that are controlled or monitored by the controller is relevant. Thus we adopt the notion notion of legal LTS from *Interface Automata* [5] where a component may not block their environment from performing actions that they monitor.

Definition 2.3. (Legal LTS) *Given LTSs $M = (S_M, A, \Delta_M, s_{M_0})$ and $E = (S_E, A, \Delta_E, s_{E_0})$, where A is partitioned into actions controlled and monitored by M ($A = A_C \dot\cup A_M$), we say that M is a legal LTS for E if for all $(s_E, s_M) \in E\|M$ it holds that $\Delta_E(s_E) \cap A_C \supseteq \Delta_M(s_M) \cap A_C$ and also that $\Delta_E(s_E) \cap A_M \subseteq \Delta_M(s_M) \cap A_M$.*

Simulation relation between two LTSs is formally defined as follows.

Definition 2.4. (Simulation) *Let $\wp$ be the universe of all LTSs with communicating alphabet A. Given E and F in $\wp$, we say that E simulates F, written $E \geq F$, when (E, F) is contained in some* simulation relation *$R \subseteq \wp \times \wp$ such that for all $\ell \in A$ and $(E, F) \in R$ we have $E \xrightarrow{\ell} E'$ implies that there is F' such that $F \xrightarrow{\ell} F' \wedge \forall s_E \in \mathrm{init}(E') \cdot \exists s_F \in \mathrm{init}(F') \cdot (E', F') \in R$.*

We fix Fluent Linear Temporal Logic (FLTL) [15] as the language for describing properties. A *fluent Fl* is defined as $Fl = \langle I_{Fl}, T_{Fl}, Init_{Fl}\rangle$, where $I_{Fl} \subseteq A$ is the set of initiating actions, $T_{Fl} \subseteq A$ is the set of terminating actions and $I_{Fl} \cap T_{Fl} = \emptyset$. A fluent may be initially *true* or *false* as indicated by $Init_{Fl}$. Every action $\ell \in A$ induces a fluent, namely $fl_\ell = \langle \ell, A \setminus \{\ell\}, \mathit{false}\rangle$.

Let $\mathcal{F}$ be the set of all fluents over A. An FLTL formula is defined inductively using the standard Boolean connectives and temporal operators $\mathbf{X}$ (next), $\mathbf{U}$ (strong until) as follows:

$$\varphi ::= Fl \mid \neg\varphi \mid \varphi \vee \psi \mid \mathbf{X}\varphi \mid \varphi\mathbf{U}\psi$$

where $Fl \in \mathcal{F}$. Additionally, as it is usual, we define $\varphi \wedge \psi$ as $\neg\varphi \vee \neg\psi$, $\Diamond\varphi$ (eventually) as $\top\mathbf{U}\varphi$, $\Box\varphi$ (always) as $\neg\Diamond\neg\varphi$, and $\varphi\mathbf{W}\psi$ (weak until) as $(\varphi\mathbf{U}\psi) \vee \Box\varphi$.

Let Π be the set of infinite traces over A. The trace $\pi = \ell_0, \ell_1, \ldots$ satisfies a fluent Fl at position i, denoted $\pi, i \models Fl$, if and only if one of the following conditions holds:

- $Init_{Fl} \wedge (\forall j \in \mathbb{N} \cdot 0 \leq j \leq i \rightarrow \ell_j \notin T_{Fl})$
- $\exists j \in \mathbb{N} \cdot (j \leq i \wedge \ell_j \in I_{Fl}) \wedge (\forall k \in \mathbb{N} \cdot j < k \leq i \ \rightarrow \ell_k \notin T_{Fl})$

In other words, a fluent holds at position i if and only if it holds initially or some initiating action has occurred, but no terminating action has yet occurred.

For an infinite trace π, the satisfaction of a (composite) formula φ at position i, denoted $\pi, i \models \varphi$, is defined as follows:

$$\begin{array}{lcl}
\pi, i \models Fl & \triangleq & \pi, i \models Fl \\
\pi, i \models \neg\varphi & \triangleq & \neg(\pi, i \models \varphi) \\
\pi, i \models \varphi \vee \psi & \triangleq & (\pi, i \models \varphi) \vee (\pi, i \models \psi) \\
\pi, i \models \mathbf{X}\varphi & \triangleq & \pi, i+1 \models \varphi \\
\pi, i \models \varphi\mathbf{U}\psi & \triangleq & \exists j \geq i \cdot \pi, j \models \psi \wedge \\
 & & \forall\ i \leq k < j \cdot \pi, k \models \varphi
\end{array}$$

We say that φ holds in π, denoted $\pi \models \varphi$, if $\pi, 0 \models \varphi$. A formula $\varphi \in$ FLTL holds in an LTS E (denoted $E \models \varphi$) if it holds on every infinite trace produced by E.

An LTS control problems aims to find, given an LTS E modelling the environment to be controlled and a goal φ to be achieved, a *controller* M in the form of an LTS such that $E\|M$ does not restrict uncontrollable actions of E, does not have deadlocks and satisfies φ

Definition 2.5. (LTS Control [6]) *Given a domain model in the form of an LTS $E = (S, A, \Delta, s_0)$, a set of controllable actions $A_c \subseteq A$, and an FLTL formula φ, a solution for the LTS control problem $\mathcal{E} = \langle E, \varphi, A_c\rangle$ is an LTS $M = (S_M, A_M, \Delta_M, s_{0_M})$ such that M is a legal LTS for E, $E\|M$ is deadlock free, and every trace π in $E\|M$ is such that $\pi \models \varphi$.*

3. APPROACH

The central concept in our approach is that of the *control stack*, which has in each *tier* a controller synthesis problem for a particular mission requirement and environment model. Overall the control stack specifies the robot's mission.

The key requirements the approach imposes in order to guarantee graceful degradation and progressive enhancement are that (see Figure 1): (i) higher-level environment models must be simulated by lower-level environment models, capturing a notion of idealisation of higher-level models; (ii) higher-level controllers used to achieve enhanced functionality must be simulated by lower levels controllers, ensuring a consistent overall strategy; (iii) the runtime infrastructure must be capable of detecting when an inconsistency between an environment model (in any tier) and the runtime environment occurs; (iv) a sound automated replanning procedure for each tier that is expressive enough to deal with the system requirements for its tier must be provided, allowing progressive system enhancement after inconsistencies have been detected. Our implementation of the approach provides the runtime infrastructure (iii) and planning procedure (iv), guarantees controller simulation (ii), and checks that the models given in a control stack specification satisfy (i).

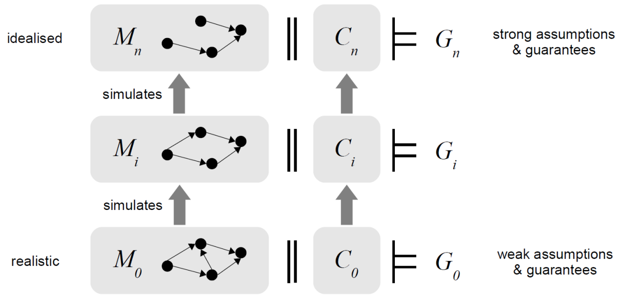

Figure 1: Multi-tier control problem

The environment models are expected to be ranked in terms of the degree of idealisation of the environment they represent. The environment model M_0 is the least idealised and require that environment models further up the hierarchy allow strictly less behaviour. This can be formally captured via a simulation relation [23], $M_i \geq M_j$ for $i < j$. We require environment models to have the same communicating alphabets partitioned identically into *controlled* and *monitored* actions. Controlled actions are those that the robot may choose to perform, while monitored actions are events that the robot observes in the environment. In summary, the less idealised the environment model is, the more behaviour (in terms of unexpected actions and non-determinism) may arise.

Each tier i has an associated requirement (G_i) to be achieved by the system assuming that the runtime environment conforms to the environment model for that tier (M_i).

Each tier introduces a control problem $\mathcal{E}_i = \langle M_i, G_i \rangle$. A solution to a control problem (a controller) is a deterministic LTS that, when composed with its environment, guarantees requirement G_i (i.e. $M_i \| C_i \models G_i$). The control stack introduces an additional constraint: each controller must be simulated by controllers in lower tiers ($C_i \geq C_j$ for $i \leq j$). Intuitively, this requires that a controller never do something that a lower-tier controller would not do, thus ensuring that if a controller must be stopped, because the assumptions for its tier are discovered not to hold, decisions made by it up to that point have been consistent with lower-tier controllers. This allows for graceful degradation, falling back to lower-tier controllers when needed. Section 4 describes how this constraint is satisfied.

Control stack synthesis is executed bottom-up through the tiers. The operation attempts to build a controller that solves the control problem in a tier while being simulated by the controller for the tier immediately below. We do not require that control problems for all tiers have solution. It is possible that the system starts in a degraded mode, with controllers solving problems up to level i. The system, as the current state evolves, may progressively enhance its behaviour by synthesising controllers for tiers beyond tier i.

After synthesis, the enactment procedure continuously monitors the environment and concurrently executes the stack of controllers giving priority to the controller of the uppermost enabled tier. It continuously updates the current state based on monitored actions and sensed state, disabling tiers at level i and above should an inconsistency be detected at tier i (Section 4 shows how this is achieved). At any point, to *progressively enhance functionality*, a replanning attempt may be made for the lowest disabled tier. Based on the current state of the enabled tier immediately below, the state of the disabled tier is automatically approximated and an attempt is made to build a controller that will work despite the uncertainty about the current state of the tier. This demands that the controller synthesis procedure be capable of solving problems exhibiting non-determinism. Should a controller exist, it is put into the controller hierarchy and the tier is enabled. The approach does not prescribe when replanning must be attempted. In principle this can be done at any time, however in practice replanning may be associated with a clock or with heuristics related to the problem domain. For a detailed explanation of the enactment procedure the reader is referred to [7].

4. IMPLEMENTATION

The implementation of our framework consists of two main components: a planner, which implements the controller synthesis algorithms, and an enactor, which handles runtime execution of the control stack.

4.1 Planner

The Modal Transition System Analyser (MTSA) [8], is a tool for developing and analysing compositional models of concurrent systems, using the Finite State Processes (FSP) process algebra. Importantly for our approach, MTSA implements controller synthesis algorithms for Generalised Reactivity(1) (GR(1)) goals, which cover an expressive subset of linear temporal logic including safety and liveness properties [8]. Our general approach is agnostic as regards the synthesis procedure, but GR(1) is expressive enough for many domains. We extended MTSA to support the specification and synthesis of complete control stacks A control stack `C` is specified in MTSA as follows:

```
controlstack ||C@{Controlled} {
  tier(ENV, REQ)
  ...
  }
```

where `Controlled` refers to a set of controlled actions, and where each tier consists of environment model `ENV` and mission requirement specification `REQ`. A control stack may consist of any number of tiers ordered such that the last tier has the most realistic environment model.

Environment models and requirements are defined using existing support in MTSA for process and property specification in FSP and FLTL (fluent linear temporal logic), and examples of them can be found in Section 5.

Synthesis of the control stack is achieved by solving the controller synthesis problem of each tier bottom-up from the lowest tier. If no solution is found for the problem in a particular tier, synthesis of the stack terminates at that tier. The procedure also includes a sanity check that the environments of tiers simulate the one immediately above.

Synthesis for a single tier i consists of the following steps:

1. Compose the tier's environment model E_i in parallel with the controller C_{i-1} generated by the tier below (if there is a tier below) to create E'_i. This ensures that the controller for tier i will be simulated by the controller of the tier below.

2. Solve the GR(1) controller synthesis problem for the tier's requirement on E'_i, to produce controller C_i.

3. *Complete* controller C_i to produce C'_i. The *completion* consists of considering the monitored actions enabled in each state of the controller, and adding transitions to a designated *exception state* for any monitored actions which are not enabled. These transitions capture behaviours of the environment that have not been anticipated in the present tier's environment model. If the runtime environment does not behave as the model describes, one of these transitions will be taken to the exception state. A single extra transition, which we call an *exception marker*, is added at the exception state which indicates to the enactor that a particular tier has been disabled. It is these transitions that enable the enactor to detect inconsistencies

The final control stack state machine CS is a parallel composition of the completed controllers C'_i, i.e. $CS = complete(C_1)\|...\|complete(C_n) = C'_1\|...\|C'_n$. This composition guarantees the requirements of every tier of the stack until the exception marker for tier i occurs, at which point it only guarantees the requirements of the tiers up to $i-1$.

4.2 Enactor

The enactor extends [3] to execute control stacks rather than individual controllers. It keeps track of the stack's current state, executing controlled actions (via domain-specific action implementations as in [3]) and responding to monitored environment events. When the current state is controlled, the enactor selects an enabled action at random. When the state is uncontrolled, the enactor waits to receive an environment event. In states where the only enabled action is an exception marker for some tier i, the enactor notes the degradation of the service to $i-1$ and reports this to the rest of the framework. In effect, this disables the controller for tier i. The planner may attempt at any point an enhancement by re-synthesising a controller for tier i (or above).

5. CASE STUDY

In this section we apply our approach to an autonomous robot given the mission of surveying a set of regions of a city. This case study is inspired by that given in [28]. The environment consists of five regions of interest, and when each region is visited the robot may receive a reward or incur some damage. In the original problem, the environment also contained a number of obstacles but the task of avoiding these is handled with lower-level control, and so we omit them from our discrete specification here.

Our overall mission goal is to have the robot repeatedly collect a specific reward (`reward[1]`), and collect the other rewards if possible. Unfortunately, it is not possible to guarantee achievement of this goal in the environment due to two sources of uncertainty. The first concerns the motion of the robot, which is not always reliable, and the second concerns the rewards and damage that the environment may provide in each region. In the worst case, the robot may move to the wrong region and receive unexpected rewards or damage. This would make our overall goal unachievable. However, we can decompose the mission into a series of requirements, the weakest of which can be satisfied in the most realistic environment in the lowest tier, and then introduce further tiers above for stronger requirements in more idealised environments.

The tier 1 (most realistic) environment model available to us is given in FSP syntax below.

```
SLIPPY_ROBOT = (arrive['r5]->ENV->ROBOT['r5]),
ROBOT[p:Locations] =
 (goto[q:Locations]->arrive[q]->ENV->ROBOT[q] |
 ...
 goto['r3]->arrive['r3]->ENV->ROBOT['r3] |
 goto['r3]->arrive['r4]->ENV->ROBOT['r4]).

ORIGINAL_MAP = MAP,
MAP = (
 arrive['r1]->(reward[1]->MAP | reward[2]->MAP) |
 ...
 arrive['r4]->(reward[2]->MAP | damage[2]->MAP) |
 arrive['r5] -> base -> MAP)+{ENV}.
||SLIPPY_DOMAIN = (ORIGINAL_MAP||SLIPPY_ROBOT).
```

The model states that the robot (which starts in `r5`) can perform a controlled `goto` action, which is followed by a monitored environment event `arrive`, which indicates arrival at the destination. This model displays a degree of non-determinism representing unreliable motion of the robot (it can equally represent unreliable sensing of location). In particular, there are two groups of adjacent regions. When the robot moves towards one such region, it may arrive in a different region that is adjacent to the destination. For example, moving to `r3` may lead to arrival in `r3` or `r4`.

As specified in the `MAP` process, after arrival, the environment can respond with one of several events representing rewards or damage. For instance, in region `r1`, the environment may provide `reward[1]` or `reward[2]`. Different regions provide different rewards, and region `r5` provides event `base` to represent the base location. The unpredictable motion in this environment model means that we cannot guarantee the strong properties that we are interested in. However, in the worst case we want the robot to ensure its physical safety and so our tier 1 requirement `AVOID_DAMAGE` is to avoid receiving `damage[2]`. This property is specified as follows:

```
fluent DAMAGE[i:Damages] = <damage[i], base>
ltl_property NO_DAMAGE2 = []!DAMAGE[2]
controllerSpec AVOID_DAMAGE = {
  safety = {NO_DAMAGE2}
  controllable = {CONT} }
```

The controller satisfying this requirement will never allow the robot to attempt a `goto['r3]` as it may arrive to `r4` which can result in damage.

If we assume reliable motion it is possible to guarantee stronger requirements. We specify such an assumption by removing the non-determinism from the `ROBOT` process:

```
ROBOT = (arrive['r5] -> ENV -> ROBOT['r5]),
ROBOT[p:Locations] =
(goto[q:Locations]->arrive[q]
             -> ENV -> ROBOT[q]).
||ORIGINAL_DOMAIN =
(ORIGINAL_MAP||ROBOT).
```

We are now able to introduce a more interesting mission requirement. We are particularly interested in having the robot collect `reward[1]`, and hence our tier 2 requirement `REWARD_LIVE_GOAL` is to collect `reward[1]` infinitely often (i.e. $\Box\Diamond\ reward_1$), and to visit the base infinitely often ($\Box\Diamond\ base$). It is specified as follows:

```
fluent REWARD[i:Rewards] = <reward[i], base>
fluent AT_BASE = <base, goto[Locations]>
assert REWARDS = (REWARD[1])
assert VISITBASE = AT_BASE
assert REWARDFAULTS = REWARD[2]
controllerSpec REWARD_LIVE_GOAL = {
  failure = {REWARDFAULTS}
  liveness = {REWARDS,VISITBASE}
  controllable = {CONT} }
```

The desired reward is only available in region `r1`, and the environment model states that, instead of `reward[1]`, the environment may, with some probability, give `reward[2]`. Provided that these hidden probabilities are non-zero, they can be abstracted with the assumption that arriving in `r1` infinitely often will yield `reward[1]` infinitely often. Encoding this kind of assumption in GR(1) has been demonstrated in [8] where probabilistic failures (i.e. `reward[2]`) are treated non-quantitatively.

It would now be possible to create a control stack consisting of two tiers using the above models and requirements. However, the synthesised controller only guarantees that *eventually* the desired reward will be received. In a practical setting with a robot's limited power supply, this is too weak a guarantee. Instead, we wish to have a bound on how long it will take to receive the reward. In order to do this we must strengthen our assumptions about the environment and create an idealised model of it, resulting in a third tier in our stack. In this case, we estimate that the environment has a low probability of repeatedly failing to give `reward[1]` (e.g. if the probability of `reward[1]` is 0.5 then the chance of failing in four attempts falls rapidly to 0.0625). We therefore introduce the assumption that the environment will comply within a small bound on the number of attempts, accepting that there is a small risk that this bound may be broken at runtime. In our idealised tier 3 environment model `BOUNDED_FAILURE_DOMAIN` we introduce an FSP process that restricts the previously given `ORIGINAL_MAP` such that the number of `reward` actions before the desired reward is bounded.

```
BOUNDED_FAILURES(Reward=1,Bound=2) = BF[0],
BF[i:0..Bound] = (reward[Reward] -> BF[0]
| when (i < Bound)
{{reward[Rewards]}\{reward[Reward]}} -> BF[i+1]).
||BOUNDED_FAILURE_DOMAIN =
 (ORIGINAL_MAP||BOUNDED_FAILURES(1,5)||ROBOT).
```

We are now in a position to specify the stronger requirement for tier 3. The `REWARD_BOUNDED_GOAL` states that the reward must be received within a count of 7 controlled actions.

```
fluent ENDED = <ended, reset>
fluent REWARD_BOUNDED[i:Rewards] =
     <reward[i], reset>
ltl_property BOUNDREWARDS =
     [](ENDED ->
      (REWARD_BOUNDED[1] && AT_BASE))
||BOUNDEDREWARDS(Bound=8) =
     (RUNNING || COUNT(Bound)
      || BOUNDREWARDS).
||BR = BOUNDEDREWARDS(7).
controllerSpec REWARD_BOUNDED_GOAL = {
  safety = {BR}
  controllable = {CONT} }
```

The tier 3 controller will ensure that `reward[1]` is received within at most 7 controlled actions (resetting the count when received), assuming that the runtime environment behaves like the tier 3 model. If, however, the runtime environment does not behave in this idealised manner, the control stack will ensure graceful degradation from tier 3 to tier 2. More specifically, if `reward[1]` is not received within the bound, an exception marker transition (inserted by the completion of the tier 3 controller) will be taken, and the subsequent behaviour of the control stack will no longer conform to the tier 3 controller. Instead it will guarantee only the requirements of tiers 1 and 2.

In the final, uppermost tier we have a mission requirement which relies on the most idealised model of the environment. We would like, if the environment turns out to be an ideal one, that the robot collect any other rewards available. Hence, our tier 4 requirement is for the robot to have received `reward[1]` and either of the other rewards infinitely often (i.e. $\Box\Diamond(reward_1 \wedge (reward_2 \vee reward_3))$):

```
assert ALL_REWARDS =
     (REWARD[1] && (REWARD[2] || REWARD[3]))
controllerSpec ALL_REWARD_LIVE_GOAL = {
  liveness = {ALL_REWARDS}
  controllable = {CONT} }
```

This requirement can only be achieved by assuming an environment which gives rewards deterministically:

```
PREDICTABLE_MAP = MAP,
MAP = (
 arrive['r1] -> (reward[1] -> MAP) |
 arrive['r2] -> (reward[2] -> MAP) |
 ...
 arrive['r5] -> base -> MAP)+{ENV}.
```

The specification of the mission control stack composed of the four tiers is as follows:

```
controlstack ||STACK@{CONT}= {
  tier(PREDICTABLE_DOMAIN,ALL_REWARD_LIVE_GOAL)
  tier(BOUNDED_FAILURE_DOMAIN,REWARD_BOUNDED_GOAL)
  tier(ORIGINAL_DOMAIN,REWARD_LIVE_GOAL)
  tier(SLIPPY_DOMAIN,AVOID_DAMAGE)
}
```

The resulting control stack state machine, of 2427 states, was synthesised in 2951ms on a laptop with an Intel Core i5 2.3GHz CPU and 4Gb memory.

5.1 Graceful Degradation

The four tiers of our control stack mean that the level of service can be degraded, in response to uncertainty in the environment, three times before failing completely (provided that the assumptions of the higher tiers are violated before those of lower tiers). A controller synthesised for a single model and single goal will fail completely the first time an unexpected event is encountered.

When the control stack is operating in tier 4, one of the possible traces leading to degradation is:

```
arrive['r5], base, goto['r1], count[0], arrive['r1],
reward[2], tier_disabled4
```

The `tier_disabled4` event is the exception marker used by the enactor to track the current service level. The exception occurs in this case because the tier 4 environment model does not allow `reward[2]` in region `r1`, and yet this is what happened in the runtime environment. After this exception, the control stack is operating in tier 3, achieving the tier 3 requirement. It may continue in this tier indefinitely, in the case where the runtime environment matches the tier 3 abstraction. On the other hand, one possible trace (continuing from the above trace) leading to further degradation is:

```
goto['r1], count[1], arrive['r1], reward[2],
...
goto['r1], count[5], arrive['r1], reward[2],
tier_disabled3
```

This exception occurs because the runtime environment has broken the bound given in the model, which states that region `r1` must provide a `reward[1]` within 5 attempts.

The control stack continues to operate in tier 2. Again, execution may continue from this point achieving the tier 2 requirement indefinitely. There is however the possibility of a further degradation as follows:

```
goto['r1], arrive['r3], tier_disabled2
```

This leaves the control stack operating in tier 1. A further sequence of events that lead to an exception is as follows:

```
reward[3], goto['r2], arrive['r2], damage[2],
tier_disabled1
```

Note that although we have presented the degradation from tier to tier, it is possible that the tier 1 assumptions are violated immediately, bypassing the intermediate tiers. After all tiers are disabled, the control stack cannot guarantee any goals until progressive enhancement takes place.

5.2 Progressive Enhancement

Suppose now that the stack is operating in tier 2, that is, the bound related to `reward[1]` has been broken, but the motion of the robot remains reliable. A progressive enhancement may now be considered in order to raise the service level back into tier 3. Suppose that the rewards given by the environment while in tier 2 have been observed, either automatically under a machine learning scheme or through manual intervention. Suppose further that it has been determined that, after the transient disturbance which caused the degradation from tier 3 to tier 2, the environment does in fact provide `reward[1]` within the required bound[1]. In such a situation, the initial state of the tier 3 model would be approximated. For instance, if the last executed action (in tier 2) was `goto['r1]` then the state of the tier 3 model must be one where `arrive['r1]` can occur. An attempt would then be made to synthesise a controller, and if successful the service level would be raised to tier 3.

6. EXPERIMENT

Figure 2: Nao executing mission

We have experimented with our synthesis and enactment infrastructure [3] in various robotic settings, including an AR Drone 2.0, a Katana robotic arm and a Nao H25 humanoid robot. A video of the latter executing a synthesised mission control stack similar to the one described above can be found at
`http://www.doc.ic.ac.uk/~das05/quadrotor3.avi`.

Controller enactment for these settings requires implementing each of the controlled actions in the control stack specification in terms of the existing behaviours provided by the robot's API. For instance, for the control of the Nao robot, the detection of various types of reward is achieved by recognising balls of different colours using the Nao's onboard camera. The balls are presented to the Nao upon arrival in each region. The location of the Nao within the environment (an office) is determined using trilateration with respect to a number of landmarks in positions known *a priori* (i.e. a structured environment). The landmarks themselves are recognised using the on-board camera. Similarly, rewards and locations can be recognised on the AR Drone using its front and bottom cameras respectively.

[1] In the case of manual intervention, it would be reasonable to amend tier 3 to match a different observed bound.

The synthesised mission control stack is executed by the enactor, which starts by assuming the runtime environment behaves like the model in the upper tier. Initially, in the video, we allow this assumption to hold by providing the Nao with the reward it is expecting. Later, we break the bound on the number of damage events expected in the uppermost tier, forcing the enactor to gracefully degrade the level of service. Execution continues seamlessly such that the Nao immediately seeks a repair, as demanded by the lower tier requirement.

The experiments demonstrate that our general approach can be deployed in a robotics setting on top of a high-level API that encapsulates the complexities of, for instance, control of the system dynamics that allows stable movement of the AR Drone or the localisation of the Nao robot. The resulting system can then ensure that mission-level guarantees can be gracefully and automatically degraded (or enhanced) when necessary to cope with unexpected mission-level events in the environment.

7. RELATED WORK

There has been an increasing interest in the development of robot planning motion techniques based on temporal logic (TL) specifications [30, 4, 29, 19, 4, 28, 21]. One of the main features such techniques provide is the ability to guarantee satisfaction of the goals if the environmental assumptions are met. This is especially relevant in robotics where strict safety conditions must be ensured by robot motion plans, which has been acknowledged by the community as a key problem to be tackled [13].

In addition to safety properties, TL-based approaches provide efficient algorithms for generating plans for liveness goals that allows a wide range of mission objectives to be specified such as surveillance and navigation, among many others. Techniques such as [29] and [6] automatically synthesise high-level motion plans from discrete-event descriptions of the environment and goals described as GR(1) [24] properties. In [28] goals are specified as co-safety formulas [20] to be satisfied by a non-deterministic model of the environment.

Other approaches [21, 22, 30, 4] consider settings where the environment is represented with stochastic transition systems such as Markov decision processes and the goals are expressed with temporal logic supporting probabilistic reasoning such as probabilistic computation tree logic (PCTL). The resulting plans guarantee the satisfaction of the goals up to a certain probability or expected reward.

However, all the aforementioned techniques have a single environment model with a fixed level of risk. Hence, if the real environment diverges from its model the plan would fail with unforeseen consequences. Thus, engineers must balance carefully the increased risk of introducing strong assumptions that allow achieving sophisticated goals against the robustness of having weak assumptions at the expense of only being able to achieve simpler goals.

Controller hierarchies have been studied (e.g [16]), however focus is on synthesis-time scalability rather than runtime robustness to invalid environment models.

Although we have implemented our approach in the MTSA toolset [9], this is not the first tool in implementing LTL-based controller synthesis. Tools such as Lily [17], Acacia+ [2] or Unbeast [10] implement techniques for synthesising controller from general LTL specifications. Such specifications are known to be 2EXPTIME Complete. Consequently, we restrict attention to GR(1) as it allows for tractable synthesis procedures. A number of tools supporting synthesis from GR(1) specifications have been developed [18, 25, 1, 26]. However, none of them work for event-based models which are central to our specification procedure.

8. CONCLUSIONS

In this paper we have presented an approach for robust high-level control synthesis for robot missions, and applied it in a various scenarios. In contrast to the ‘all or nothing’ approach of other work based on temporal logic, our approach allows a mission specification to include a range of requirements of different ‘strengths’ which entail different levels of risk when operating in the runtime environment. Our implementation ensures that when the stronger requirements of higher tiers cannot be met due to environmental uncertainty, the level of service degrades gracefully to a level at which requirements can be guaranteed. It then permits progressive enhancement at a later stage.

In future work we are interested in quantifying the level of risk associated with the tiers of our control stack, and combining the approach with techniques that can learn appropriate environment models for disabled tiers in the stack before progressive enhancement.

9. REFERENCES

[1] R. Bloem, A. Cimatti, K. Greimel, G. Hofferek, R. Könighofer, M. Roveri, V. Schuppan, and R. Seeber. Ratsy: a new requirements analysis tool with synthesis. In *Proceedings of the 22Nd International Conference on Computer Aided Verification*, CAV'10, pages 425–429, Berlin, Heidelberg, 2010. Springer-Verlag.

[2] A. Bohy, V. Bruyère, E. Filiot, N. Jin, and J.-F. Raskin. Acacia: a tool for ltl synthesis. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 652–657, Berlin, Heidelberg, 2012. Springer-Verlag.

[3] V. Braberman, N. D'Ippolito, N. Piterman, D. Sykes, and S. Uchitel. Controller synthesis: From modelling to enactment. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1347–1350. IEEE Press, 2013.

[4] I. Cizelj and C. Belta. Control of noisy differential-drive vehicles from time-bounded temporal logic specifications. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 2021–2026, May 2013.

[5] L. de Alfaro and T. A. Henzinger. Interface automata. In *ESEC / SIGSOFT FSE*, pages 109–120. ACM, 2001.

[6] N. D'Ippolito, V. Braberman, N. Piterman, and S. Uchitel. Synthesising non-anomalous event-based controllers for liveness goals. *ACM Tran. Softw. Eng. Methodol.*, 22, 2013.

[7] N. D'Ippolito, V. A. Braberman, J. Kramer, J. Magee, D. Sykes, and S. Uchitel. Hope for the best, prepare for the worst: multi-tier control for adaptive systems. In *ICSE*, pages 688–699, 2014.

[8] N. D'Ippolito, V. A. Braberman, N. Piterman, and S. Uchitel. Synthesis of live behaviour models for fallible domains. In R. N. Taylor, H. Gall, and N. Medvidovic, editors, *ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 211–220. ACM, 2011.

[9] N. D'Ippolito, D. Fischbein, M. Chechik, and S. Uchitel. Mtsa: The modal transition system analyser. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 475–476, Washington, DC, USA, 2008. IEEE Computer Society.

[10] R. Ehlers. Symbolic bounded synthesis. In *Proceedings of the 22Nd International Conference on Computer Aided Verification*, CAV'10, pages 365–379, Berlin, Heidelberg, 2010. Springer-Verlag.

[11] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Model evolution by run-time parameter adaptation. In *ICSE 2009*, pages 111–121. IEEE, 2009.

[12] N. Esfahani and S. Malek. Uncertainty in self-adaptive software systems. In R. de Lemos, H. Giese, H. A. Müller, and M. Shaw, editors, *Software Engineering for Self-Adaptive Systems*, volume 7475 of *Lecture Notes in Computer Science*, pages 214–238. Springer, 2010.

[13] T. Fraichard and J. J. K. Jr. Guaranteeing motion safety for robots. *Auton. Robots*, 32(3):173–175, 2012.

[14] C. Ghezzi, M. Pezzè, M. Sama, and G. Tamburrelli. Mining behavior models from user-intensive web applications. In *ICSE*, pages 277–287, 2014.

[15] D. Giannakopoulou and J. Magee. Fluent model checking for event-based systems. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-11, pages 257–266, New York, NY, USA, 2003. ACM.

[16] R. C. Hill, J. E. R. Cury, M. H. de Queiroz, D. M. Tilbury, and S. Lafortune. Multi-level hierarchical interface-based supervisory control. *Automatica*, 46(7):1152–1164, July 2010.

[17] B. Jobstmann and R. Bloem. Optimizations for ltl synthesis. In *Proceedings of the Formal Methods in Computer Aided Design*, FMCAD '06, pages 117–124, Washington, DC, USA, 2006. IEEE Computer Society.

[18] B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem. Anzu: A tool for property synthesis. In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, pages 258–262, Berlin, Heidelberg, 2007. Springer-Verlag.

[19] H. Kress-Gazit, G. Fainekos, and G. Pappas. Temporal-logic-based reactive mission and motion planning. *Robotics, IEEE Transactions on*, 25(6):1370–1381, Dec 2009.

[20] O. Kupferman and M. Y. Vardi. Model checking of safety properties. *Form. Methods Syst. Des.*, 19(3):291–314, Oct. 2001.

[21] M. Lahijanian, J. Wasniewski, S. Andersson, and C. Belta. Motion planning and control from temporal logic specifications with probabilistic satisfaction guarantees. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 3227–3232, May 2010.

[22] A. Medina Ayala, S. Andersson, and C. Belta. Temporal logic control in dynamic environments with probabilistic satisfaction guarantees. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 3108–3113, Sept 2011.

[23] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[24] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive (1) designs. *Lecture notes in computer science*, 3855:364–380, 2006.

[25] A. Pnueli, Y. Sa'ar, and L. D. Zuck. Jtlv: A framework for developing verification algorithms. In *Proceedings of the 22Nd International Conference on Computer Aided Verification*, CAV'10, pages 171–174, Berlin, Heidelberg, 2010. Springer-Verlag.

[26] V. Raman and H. Kress-Gazit. Synthesis for multi-robot controllers with interleaved motion. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, 2014.

[27] D. Sykes, D. Corapi, J. Magee, J. Kramer, A. Russo, and K. Inoue. Learning revised models for planning in adaptive systems. In *Proceedings of ICSE*, 2013.

[28] A. Ulusoy, M. Marrazzo, K. Oikonomopoulos, R. Hunter, and C. Belta. Temporal logic control for an autonomous quadrotor in a nondeterministic environment. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 331–336, May 2013.

[29] E. Wolff, U. Topcu, and R. Murray. Efficient reactive controller synthesis for a fragment of linear temporal logic. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 5033–5040, May 2013.

[30] C. Yoo, R. Fitch, and S. Sukkarieh. Provably-correct stochastic motion planning with safety constraints. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 981–986, May 2013.

Control Theory Meets Software Engineering: The Holonic Perspective

Luca Pazzi
DIEF - University of Modena and Reggio Emilia
Via Pietro Vivarelli 10
Modena, Italy
luca.pazzi@unimore.it

ABSTRACT

One of the main challenges towards a software-based theory of control consists in finding an effective method for decomposing monolithic event-based interactive applications into modules. The task is challenging since this requires in turn to decompose both the invariants to be maintained as well as the main control loop. We present a formalisms for gathering portion of behaviour by special units, called holons, which are both parts and wholes and which can be arranged into part-whole taxonomies. Each holon hosts a state machine and embodies different invariants which give semantics to its states. Control is achieved by both taking autonomously internal actions by the state machine in order to maintain such state invariants, as well as by having the the state machine move from one invariant to another by actions driven by external events. Such an approach requires to introduce non trivial solutions in order to allow communication among such modules, mainly by implementing control loops among couple of holons. The proposed model consists essentially in shaping each module in order to be both a controller and a controllable entity. Each module may control a definite number of modules and is controlled by a single module. Control is exercised by discrete events which travel through a communication medium. Control actions as well as feedback events travel thus from a module to the another, thus achieving local control loops which, taken globally, decompose the main control loop.

Categories and Subject Descriptors

H.1 [**MODELS AND PRINCIPLES**]: General

Keywords

Holons, Part-Whole Statecharts, Compositional verification

1. INTRODUCTION

Time-dependent real-time software systems deal with fast changing environments, to which they adapt in order to maintain a specific and often complex operation task. Changing environments are heavily dynamical, and their changing behaviour is often revealed by events which reach the software system [12]. Software systems react to external incoming events by producing, in turn, other outgoing events towards the environment, thus closing the loop and becoming *de facto* control systems (Figure 1).

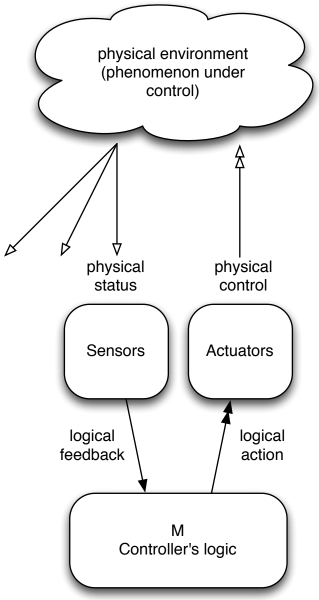

Figure 1: Asymmetry in the control process at first glance. A generic controllable phenomenon does not know its controller M, while the controller is designed specifically for dealing with the phenomenon through actuators and sensors.

Software systems are required to exhibit however other features, mainly effectiveness, robustness, and dependability: when dealing closely with real world, such features have to ensure, in first place, liveness and safety properties. Albeit a control software may be even conceived as a monolithic block modelling a single global behaviour, a really effective software development methodology requires to modularise the overall control software in order to defeat the overall complexity. This requires in turn a coherent criterion for decomposing, as well as composing, the control loop. Though control engineers practiced such a decomposition for

CTSE'15, August 31, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3814-1/15/08...$15.00
http://dx.doi.org/10.1145/2804337.2804343

many years, a deeper understanding of the basic software engineering notion of module structure and connectivity would improve both control theory and software engineering. If a modelling paradigm is more efficient then it is presumably more natural, i.e., ontologically sound. By the current view, systems are seen "as transmitting and transforming signals from the input channel to the output channel, and interconnections are viewed as pathways through which outputs of one system are imposed as inputs to another system" [13]. Such a view of systems seen as signal transformers is naively accepted by both software and control engineering communities, but reveals its limitation especially when modelling interconnected systems under a software engineering perspective, typically raising problems in terms of understandability, reusability and maintainability. Even worse, once systems are interconnected by direct links, model checking becomes infeasible due to the exponentially growing complexity of the resulting system. A paradigm shift is therefore needed in order to ensure effective modularity. The paper describes an ongoing research [6][8] in modular decompositions of behaviour of reactive and autonomic systems [7]. In particular this paper will focus on the hierarchic control of distributed invariants and can be seen a continuation of [6]. We believe that the application to control theory of the holonic approach may shed new light and bring further ideas to the original research in software engineering.

As in [9] we are interested in the basic control problem of ensuring that a logical predicate remains invariantly true when it is initially satisfied. By the supervisory control approach [10] the problem bears to state space explosion that is exponential in the number of system components. It becomes thus necessary to explore modular system architectures with the property that the overall complexity can be appropriately defeated. We devise a state-based tree-like structure of modules as in [5]: our composition model and general motivations are similar also to those in [2], but we believe our model is conceptually cleaner and simpler.

Our approach is mainly focused on restricting communication among modules and on labelling states by invariant state propositions as described in detail in [8]. State transitions laid among states have to comply with both arrival and starting state propositions. In other words, each module is "correct by construction" in the sense that it is guaranteed that when control is in a state a given proposition on the states of its components is always satisfied. More complex predicates can be existentially or universally verified by exploring the state diagrams within the modules.

We start by examining (Section 2) how events reveal structural aspects of the domain, and as such may be used in order to shape modelling constructs among which they flow. Events are not only the basic elements of communication in control, they reveal different aspects of entities participating in the scenario being controlled. The way events flow to and from them may in fact help conceiving a method for decomposing control around the participating entities and for shaping control structures. Such control structures may on their turn be centred around behavioural constructs, finite state automata, which indeed prescribe control by parsing and generating event flows. Additionally, since the time of the Fusion methodology [1], finite state automata provide a very effective tool for specifying software behaviour in the development process [11] and notably they can be easily transformed into code or directly executed by a suitable interpreter. Holon inspired control modules are therefore shaped around such behavioural constructs. They provide a connection framework in which events flow from one state machine to another by following a decomposition hierarchy. Section 3 introduces finally holons as control units by a working example adapted from [6] used in order to show the feasibility of the approach.

2. FROM EVENTS TO MODULES

Events taken separately are not meaningful: their sequences instead reveal complex phenomena happening in the real world. Such sequences are part of a language built upon them denoting indeed real-world phenomena. For example it can be easily observed that "*door-open*, *door-close*, *light-green*, *engine-on*" is a string of events belonging to the language describing the interaction of an automated car and a traffic light. Such a car is additionally provided with doors for embarking and disembarking passengers along a track: additionally, it stops and restarts its engine according to signals from traffic lights on its route. Doors are automatically opened when the car has to embark/disembark passengers and closed when it restarts. Doors are locked when the car moves, but the locking mechanism may be bypassed and the doors may be opened at any time, in which case the car has to stop immediately. By the schema of Figure 2 it may be hypothesised that a controller takes care of the mutual interaction of the doors and of the engine.

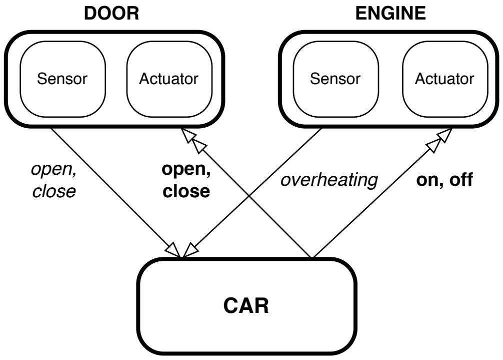

Figure 2: Module Car parses and generates the joint language which denotes the synchronization of its controlled modules. Events may be distinguished between input events (bold) and output events (italic) and travel, respectively, along double and single arrow lines.

Entities belonging to the environment produce a language built from a vocabulary of words, i.e. strings of events coming from their respective event alphabet. Modelling some sort of interaction means restricting the full set of sequences that can be built from the alphabet of events of the participating entities. Consider for example the interaction of the doors of the automated car and its engine. Once the engine and the doors do not interact, that is when they are taken physically apart, any sequence of events from their joint vocabulary may be observed. We call it the *free* alphabet of the doors and the engine, for example "*door-open*, *engine-off*" and "*door-open*, *engine-on*" are both words belonging to their joint free vocabulary. In order to model a useful behaviour, the second word has to be banned, since a simple safety constraint does not allow the vehicle to move with doors open.

In order to shape a useful joint language, that is a useful behaviour, we therefore insert a module which is able to observe and to prescribe events to both the involved entities. Module Car of Figure 2, for example, is the place where any aspect regarding the mutual interaction of the car's components, doors and engine, should be modelled. It may be observed that there are now two distinct loops, each carrying information from module Car to and from module Door and Engine. An event can be either directed from the controller towards a component (double arrow lines), in this case driving its actuator, or being emitted by the sensor of the component towards the controller (single arrow lines). In the former case the event denotes an action that the component has to undertake, in the latter the event denotes a spontaneous action that already happened within the component.

Module Car may be conceived as hosting an automaton which acts both as parser and generator of the joint language of the two components. The parsing and generating mechanism may be thought as being implemented by transitions of such automaton which are triggered by events belonging to the sublanguage being parsed while, at the same time, forwarding events belonging to the sublanguage being generated (Figure 3).

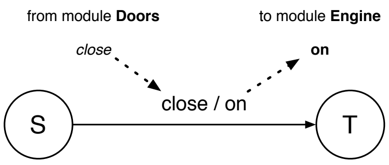

Figure 3: Module Car parses and generates the joint language of modules Doors and Engine by a traditional Statecharts diagram, whose transitions are labelled by events belonging to the modules in the form of triggers and forwarded events

2.1 From Modules to Holons

The car *taken as a whole* interacts with the traffic lights signals, producing its own set of events, for example *start*, *stop* and *fail* which in turn form meaningful words once interleaved with the alphabet of the traffic light, for example "*light-green*, *car-start*" and "*light-red*, *car-stop*".

Such events denote the *external behaviour* of the car, that is the behaviour of the car *taken as a whole* observable from outside. Conversely, the joint behaviour of the components of the car (the doors and the engine in the example) will be referred to as the *internal behaviour* of the car. Both behaviours have to agree, and therefore the related languages have to be parsed and generated accordingly. It is therefore necessary to endow the automaton within the module of additional modelling capabilities, in order to have the two behaviours match in a meaningful way. In other words, the languages of the car components have to agree with the language of the car taken as a whole.

This requires a more elaborated syntax and semantics than traditional Statecharts. Part-Whole Statecharts [8] allow to label a state transition with specific constructs in order to account for *internal* and *external* events. Two different transition typologies result, as shown in Figure 4. Synchronisation amongst the internal and the external language is achieved by allowing both internal and external events to be present in the same state transition with specific operational meanings, which underly two different reactive mechanisms. For example, the transition of Figure 4 (a) models an externally triggered behaviour. The car, seen as a whole, is required to start by receiving event go from an external module, and the doors and the engine are as a result required of being, respectively, closed and turned on. Figure 4 (b) models instead an internally triggered behaviour. In that case, the happening of an event within a component (the opening of a door while moving) requires the engine of being stopped and the car, seen as a whole, to emit an undirected fail event towards an unspecified client module.

The need to take into account, at the same time and by a single behavioural construct, an internal and an external language, requires to introduce a new modular construct, called *holon*, presented in the next Section.

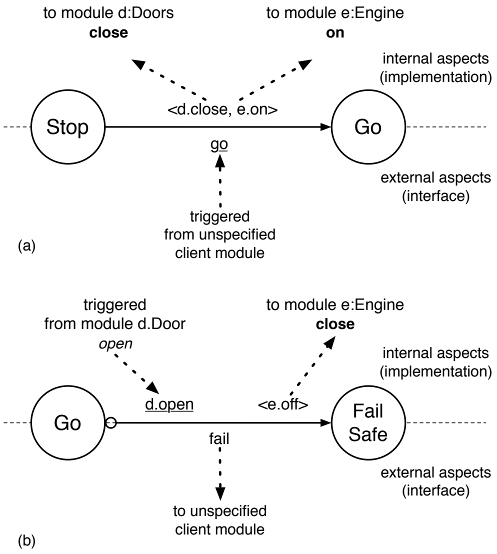

Figure 4: Part-Whole Statecharts is a state-based formalism which is able, amongst other features, to integrate internal and external behavioural aspects. Two main reactive patterns are feasible, that is externally (a) and internally (b) triggered transitions. Events are syntactically distinguished into directed and undirected ones.

3. THE HOLONIC FRAMEWORK

In the preceding sections we argued for constructs which exhibit a "double behavioural nature", that is they should be able to account both for an internal language as well as an external language. What we need are therefore modular units which are able to conform to such a double nature. Each module needs therefore *four* ports, two on the internal and two on the external behavioural side. Such ports are meant to be behavioural connection points to other modules. Modules should communicate one with the other by connecting the internal ports of the module acting as component to the external ports of the module acting as compound entity. Each module should finally expose only its external features, hiding the internal ones.

Holons, by Arthur Koestler [3][4], have a double, "Janus face", nature, thus being able to host both an implementation and an interface as in Figure 5-(a), referred to as internal and external aspects in the previous sections. The interface allows to view and use the module as "part", the implementation allows instead to view the module as "whole", that to coordinate a number of other holons as parts.

The interest of the holonic approach in the context of a novel software-engineering based theory of control consists indeed in the feasibility of composing holons through partial control loops. Each holon playing the role of whole coordinates n different holons playing the role of parts through n control loops. Holons within a holarchy cooperate in order to achieve a global task by exchanging control events of different typologies which travel along the lattice of control loops established among them.

The ontological rationale behind such a choice is that, as shown in Figure 1, a monolithic control software (the controller) acts upon the environment, aimed at changing its current set of properties, called collectively state. As the state of the environment changes, events are generated (for example temperature changes) and broadcast towards the controller. Vice versa, the controller prescribes directed actions to actuators, for example turning a furnace on in order to raise the temperature of the room. In other words, this marks an asymmetry since the controller knows the controlled entity, but not vice versa, in the sense that trivially the temperature of the room only implicitly and indirectly acts on the temperature controller. The proposed framework mimics such an ontological property of controlled processes by having controllers the capability to observe controlled entities, not vice versa.

This paves the road for an effective software development method, where modular reusability is of primary concern and underlies the general principle of *asymmetry* in control. Each modular artifact is in fact totally reusable since it has to be designed in full generality without having to know the module to which it has to be composed.

3.1 Invariant-Based Holonic Control

Monolithic control applications fulfill different tasks, each possibly consisting in maintaining different logical invariants.

Holons have an asymmetrical and complementary nature by which they can be composed into *holarchies*, as in Figure 5-(b). Root modules can be seen as bare implementations, since they do not need to be further composed in the context of a control application. Leaf modules are simple interfaces, since they are the logical view of sensors and actuators. A holarchy can be thus defined as a set of holon modules connected through control loops by which events flow upwards and downwards. Events can be classified into different typologies depending on the direction in which they travel and to the target to which they are directed. The framework presented is aimed at maintaining equivalently control of such state invariants dispersed amongst separate component holons within a hierarchy. Within a specific module of the holarchy such invariants become state invariants of the PW-Statechart governing it. Internal and external aspects of the PW-Statechart hosted by a single holon match the communication ports, by following the schema depicted in Figure 5-(a).

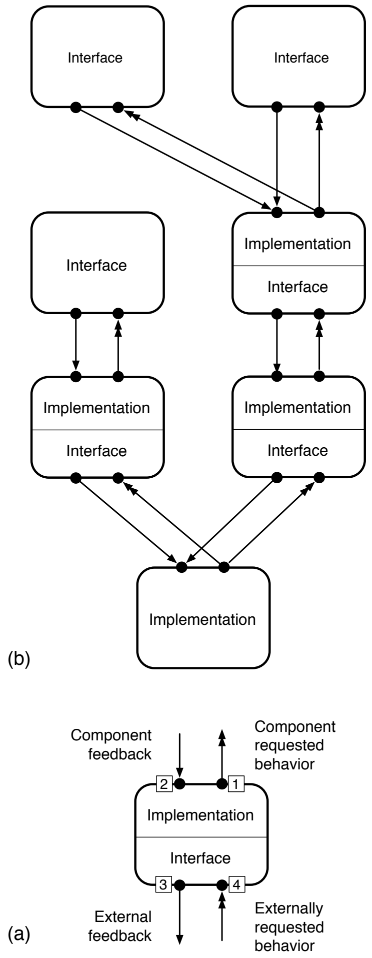

Figure 5: (a) Holon modules present four logical ports and are split into an implementation and an interface part; (b) By connecting iteratively port 2 with port 3 and port 1 with port 4 it is possible to obtain hierarchical structures called "holarchies", consisting of nodes connected by partial control loops.

Koestler conceived indeed holons as self-regulatory units [4]. Each holon node is designed for maintaining a limited set of invariants which are not visible to its components. Modules Doors and Engine, in the example, do not *know* that their current joint state must fulfill a limited set of state invariants. Such invariants are encoded in the holon which has them as components. Only module Car indeed *knows* for example that the opening of the door requires the engine to stop.

A holarchy may be seen therefore as a directed acyclic graph of self-regulatory nodes, each node maintaining its own invariants by perturbing its components and being perturbed by other nodes having it as component. A state invariant can be seen indeed as a "configuration" of components.

Module Car in Figure 6 has three invariants associated to its states; state Stop for example is such that when the control is in the state then doors are opened and the engine is stopped (C_1), when in state Go doors are closed and the

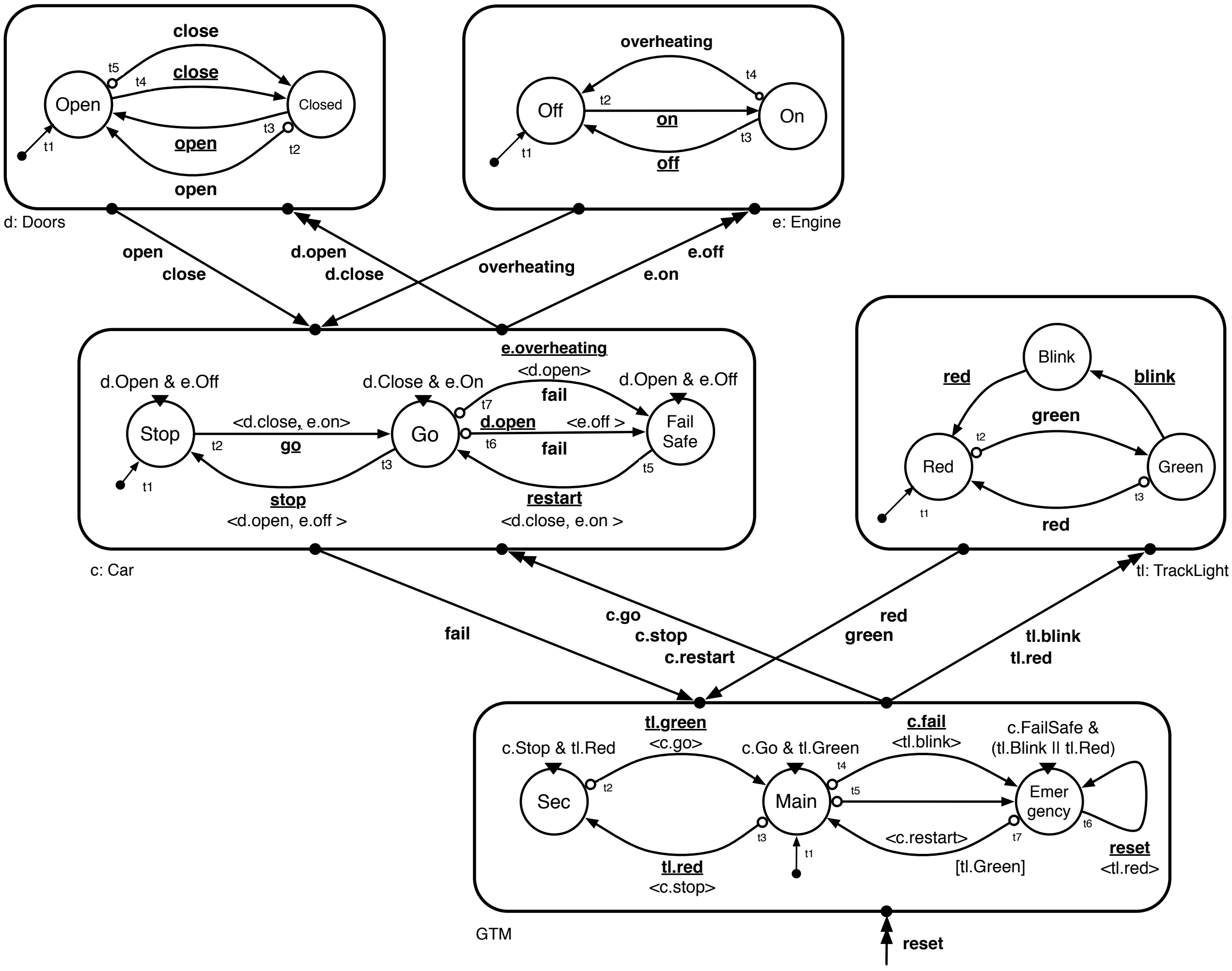

Figure 6: The holarchy implemented by PW-Statecharts modelling the behaviour of the holon Car by its two component holons (Doors and Engine) and of a GTM (Global Track Monitor) holon having Car and TrackLight as components (adapted from [6]).

engine is running (C_2). Finally, when in state FailSafe doors are opened and the engine is stopped (C_3). Observe that the same state invariant may be associated to different states, for example Stop and FailSafe share C_1 and C_3. The holon Car moves along the three state invariants above. Each state invariant may be invalidated by both an external as well as an internal stimulus. External stimuli change the current state and the associated invariant (Figure 7-(a)). Internal stimuli come from autonomous state changes happening within components. Invariants need then to be restored by sending commands to components (Figure 7-(b)).

3.2 Partitioning Control Invariants

A holonic application maintains global state invariants by the recursive composition of control over partial invariants. Each partial invariant is such that it logically constrains the states belonging to the component holons. For example, when the control is in state Main in module GTM (Global Track Monitor) of Figure 6, then invariant condition $C_2^{\mathsf{GTM}} = \mathsf{c.Go} \wedge \mathsf{tl.Green}$ holds, meaning that module car is moving and traffic light is green.

Control consists in comparing, at each clock step, whether condition C_2^{GTM} holds compared to the global state of its components. In this way, only components at the immediate upper level have to be checked in order to verify whether the desired invariant holds. In case a difference is noticed between C_2^{GTM} and the current state of the components, for example by the traffic light changing to red, either

1. current state invariant has to be restored by the global track monitor GTM by sending a command to its *immediate upper level components* Car and TrackLight; this is not possible since the GTM would request the light to switch back to green but the requested traffic light behaviour is modelled through autonomous *non triggerable* transition t_3; or

2. the GTM module has to switch to a different state invariant which complies with the current global state of the *immediate upper level components*, in the example by moving from state Main to state Sec through transition t_3 in module GTM.

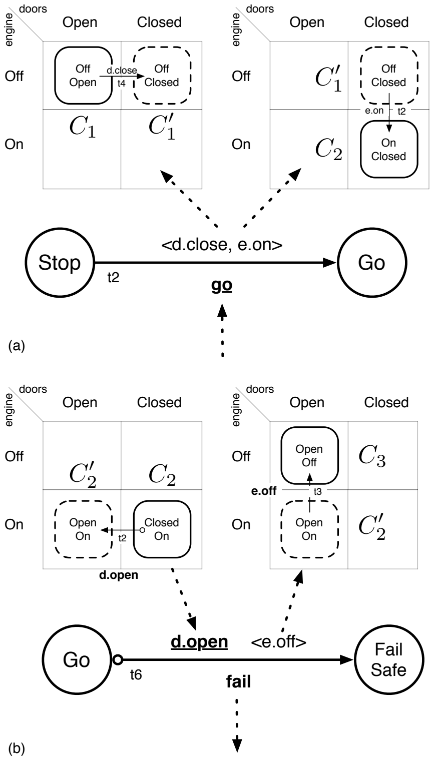

Figure 7: (a) An external stimulus (event go) make the holon change its current state from $C_1 = \mathsf{d.Open} \wedge \mathsf{e.Off}$ to $C_2 = \mathsf{d.Closed} \wedge \mathsf{e.On}$ by forwarding additional event stimuli towards its components, which modify their configuration. (b) An internal stimulus (a door is opened manually by event $\mathsf{d.open}$) makes invariant $C_2 = \mathsf{d.Closed} \wedge \mathsf{e.On}$ invalid and triggers a change of state to the state having associated condition $C_3 = \mathsf{d.Open} \wedge \mathsf{e.Off}$, which becomes the new state invariant. The holon then emits event fail towards the external composition context.

In both cases the only current global state which has to be looked up or modified is the one resulting form the current state of the components at the immediate upper level. Control is thus exercised only by establishing a control loop consisting of a *single level* of composition. By such schema of control, commands travel upward while feedback travels downward and is used to establish, incrementally, the current state of the components. Feedback from components may implicitly trigger state transition, that is, feedback consists in information regarding their current state, not the transitions which have to be triggered in the client holon. For example in Figure 6 a change of state in the doors is revealed by feeding back the event d.open which denotes indeed the change of state which happened. Holon Car reacts to such an event by triggering a state transition since the current state invariant no longer holds (in fact, doors must be closed when car is running). Other client holons (that is holons *using* the doors holon as component) may not need to react to doors opening since they implement a different application policy (for example doors may be both opened or closed while the car is stopped). In other words, the same change of state in a component is dealt with in different ways, not foreseeable at design time.

Observe finally that since:

$\mathsf{c.Go} \implies (\mathsf{d.Close} \wedge \mathsf{e.On})$

We have that

$\mathsf{c.Go} \wedge \mathsf{tl.Green} \implies \mathsf{d.Close} \wedge \mathsf{e.On} \wedge \mathsf{tl.Green}$

In other words each state invariant denotes one or more configuration of leaf holons, as depicted in Figure 8. For example C_3^{GTM} is equal to $C_3^{\mathsf{Car}} \wedge (\mathsf{tl.Red} \vee \mathsf{tl.Blink})$ which in turn is equal to $\mathsf{d.Open} \wedge \mathsf{e.Off} \wedge (\mathsf{tl.Red} \vee \mathsf{tl.Blink})$

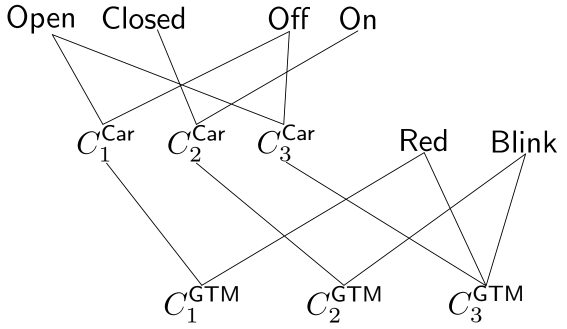

Figure 8: Each invariant in the Car as well as the GTM (Global Track Module) denotes one or more configurations of leaf (i.e. basic, unstructured) holons.

3.3 Discussion and Further Research

Further research is needed in order to answer some noticeable questions. For example: does the holonic perspective provide some guidance for decomposing the control loop as well as the overall behaviour? Which are the general requirements imposed by the holonic perspective? Which are its limits?

Holons do not provide any criterion for decomposing the control loop, rather they support the decomposition or partitioning of control by simply hosting portions of behaviour which is strictly necessary in order to implement the reactive behaviour among a finite and limite number of components. Events are generated as part of such a working behaviour and are therefore dependent from it. In other words there is no criteria for decomposing the main loop, each more complex behavioural level is simply built upon previous less complex levels. Each level emits events towards both more and less complex layers. Event loops simply connect the different layers and give rise to unforeseeable patterns of behaviour, as in Figure 9. Further research is needed in order to understand whether non terminating sequences of events may be occasionally generated.

The overall behaviour of a monolithic control application is devoted at maintaining a number of invariants. Each holon acts as a self regulatory node in order to maintain only some invariants. Once a mutual dependence is detected among two holon entities, a third holon having such entities as "parts" is required in order to host the mutual behaviour and the mutual invariants. Invariants within a holon hence constrain only to its component parts. In the example of Figure 6 holon subsystem Car is devoted at maintaining state invariants which deal with the mutual interaction of its component holons, i.e. doors and engine. Invariants are modelled through state propositions. Each state proposition is a log-

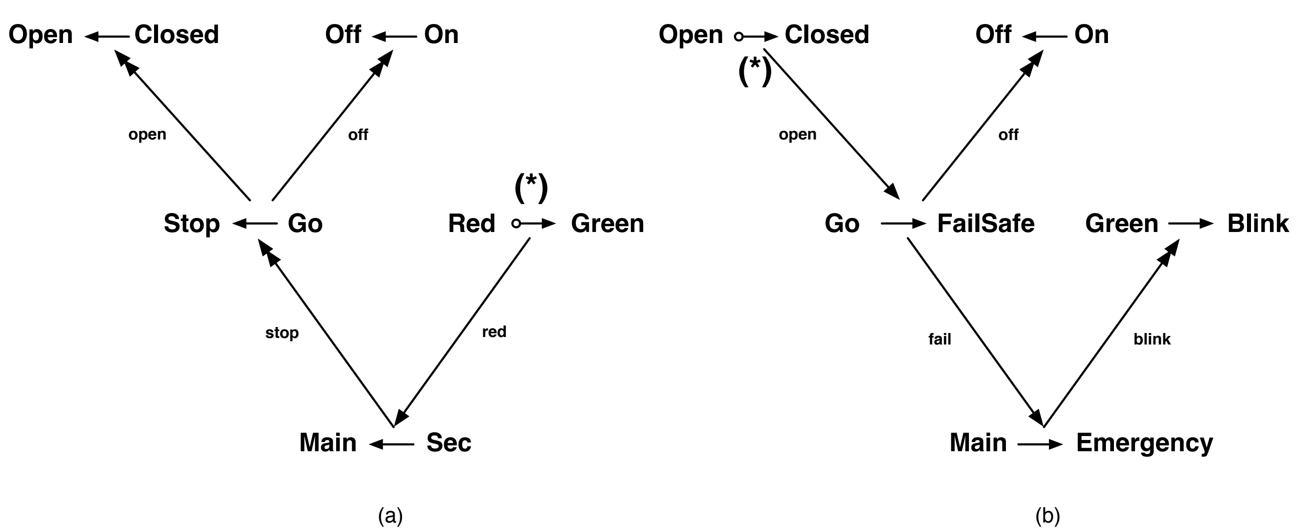

Figure 9: Two execution traces from the example of Figure 6. Observe that, by the proposed approach, events propagate both upwards and downwards. Propagation origin is marked by (*).

ical formula which denotes the current state of the current upper less complex level. At the same time each state of such level is in turn constrained by another formula. As shown in Figure 8 global invariants are maintained by maintaining simpler invariants at each level. Further research is finally needed in order to understand whether any system is decomposable through a holarchy.

4. CONCLUSIONS

Distinguishing between internal and external events requires to adopt new behavioural constructs. Such constructs may in turn be hosted by a suitable module, which provides communication facilities by which modules are connected one with the other. Internal aspects of the module implement the activity of the system by coordinating its components, external aspects model instead the activity of the system seen as a whole. Connecting internal aspects of a module to external aspects of a component module allows to implement a partitioned form of control by establishing control loops which decompose the main loop. Each control loop is simply devoted at controlling its immediate composition level, that is at controlling holons which in turn controls their own component holons, and so on.

One of more invariants coexist within a holon state-based behaviour. Such invariants are state propositions associated to the states of the state machine. Once such propositions are invalidated, regulating events are sent to its components in order to restore them. In case no restoring action is possible, the state machine performs a transition to another state whose invariant is compatible with the internal changes, making them self-regulatory units as in Koestler's vision. Modules can be moreover verified directly at design time by visiting their state invariants along their finite and limited state diagram and composed at any time within a holarchy without having to check them again.

The more appealing benefit of adopting the holonic perspective is therefore related to the capability of reducing the overall complexity by modular units, together with the feasibility of composing off-the-shelf verified modules.

5. REFERENCES

[1] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: the Fusion Method.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[2] G. Delaval, S. Mak-Karé Gueye, E. Rutten, and N. De Palma. Modular Coordination of Multiple Autonomic Managers. In *17th International ACM Sigsoft Symposium on Component-Based Software Engineering (CBSE 2014)*, page 291, Lille, France, June 2014.

[3] A. Koestler. Beyond atomism and holism - the concept of the holon. *Perspectives in Biology and Medicine*, 13(2):131–154, 1970.

[4] A. Koestler and J. R. Smythies. *Beyond reductionism; new perspectives in the life sciences.* Macmillan New York, 1970.

[5] C. Ma and W. Wonham. Nonblocking supervisory control of state tree structures. *Automatic Control, IEEE Transactions on*, 51(5):782–793, May 2006.

[6] L. Pazzi. Modeling systemic behavior by state-based holonic modular units. In J. Dingel, W. Schulte, I. Ramos, S. AbrahŃo, and E. Insfran, editors, *Model-Driven Engineering Languages and Systems*, volume 8767 of *Lecture Notes in Computer Science*, pages 99–115. Springer International Publishing, 2014.

[7] L. Pazzi and M. Pradelli. Part-whole hierarchical modularization of fault-tolerant and goal-based autonomic systems. In *Dependable Control of Discrete Systems, 2009. DCDS '09. 2nd IFAC Symposium on*, pages 175–180, 2009.

[8] L. Pazzi and M. Pradelli. Modularity and part-whole compositionality for computing the state semantics of statecharts. In *Application of Concurrency to System Design (ACSD), 2012 12th International Conference on*, pages 193 –203, june 2012.

[9] P. J. Ramadge and W. M. Wonham. Modular feedback logic for discrete event systems. *SIAM Journal on*

Control and Optimization, 25(5):1202–1218, 1987.

[10] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal on Control and Optimization*, 25(1):206–230, 1987.

[11] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1998.

[12] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):14:1–14:42, May 2009.

[13] J. C. Willems. The behavioral approach to open and interconnected systems. *Control Systems Magazine*, pages 46–99, 2007.

Author Index