# Requirements-Driven Qualitative Adaptation

Vítor E. Silva Souza, Alexei Lapouchnian, and John Mylopoulos

Department of Inf. Engineering and Computer Science, University of Trento, Italy
{vitorsouza,lapouchnian,jm}@disi.unitn.it

**Abstract.** Coping with run-time uncertainty pose an ever-present threat to the fulfillment of requirements for most software systems (embedded, robotic, socio-technical, etc.). This is particularly true for large-scale, cooperative information systems. Adaptation mechanisms constitute a general solution to this problem, consisting of a feedback loop that monitors the environment and compensates for deviating system behavior. In our research, we apply a requirements engineering perspective to the problem of designing adaptive systems, focusing on developing a qualitative software-centric, feedback loop mechanism as the architecture that operationalizes adaptivity. In this paper, we propose a framework that provides qualitative adaptation to target systems based on information from their requirements models. The key characteristc of this framework is extensibility, allowing for it to cope with qualitative information about the impact of control (input) variables on indicators (output variables) in different levels of precision. Our proposal is evaluated with a variant of the London Ambulance System case study.

**Keywords:** requirements, goal models, adaptive systems, feedback loops, qualitative reasoning.

## 1 Introduction

For software systems, as for humans and organizations alike, uncertainty is a given: at any time, the system is uncertain about all the details of its environment, or what might happen next. To cope with it, biological and social agents are capable of adapting their behavior and their objectives. Consistently with this, adaptation for software systems has become a focus of much research, addressing questions such as "How do we design adaptive systems?", "What runtime support is needed?", "How do we ensure that they have desirable properties, such as stability and quick convergence to an optimal behavior?"

We are interested in developing a set of design principles for adaptive software systems. We define adaptation as the process of the system switching from one behavior to another in order to continue to fulfill its requirements. Thus in adaptation, requirements remain unchanged and an adaptation strategy consists of choosing a suitable change of behavior to restore requirements fulfillment. Our proposed framework assumes that requirements should be at the very center of an adaptation mechanism, determining what constitutes normal behavior, what is to be monitored and what are possible compensations in case of deviations.

R. Meersman et al. (Eds.): OTM 2012, Part I, LNCS 7565, pp. 342–361, 2012.

In earlier work, we have characterized a class of requirements, called *Awareness Requirements* (*AwReqs*) that determine what needs to be monitored by an adaptive system [25]. In addition, we extended goal models (which represent system requirements, as proposed in [15]) by including control-theoretic information concerning control variables and indicators, along with qualitative differential relations that specify the impact of the former on the latter [23].

The main objective of this paper is to "close the loop" by proposing a framework within which a failure of a monitored *AwReq* leads to a new behavior that consists of selecting a new variant of the system's goal model, and/or new values for its control variables. The proposed mechanism is inspired by control theoretic concepts, notably the PID controller [13], recast in qualitative terms and using goal models to define both the desired output and the space of possible behaviors for getting it. Its key features are the use of requirements models for run-time adaptation and being highly extensible, allowing for different adaptation algorithms to be used depending on the availability and precision of information. To validate our proposal, we have implemented our framework and simulated our adaptation algorithms using different scenarios.

The rest of the paper is organized as follows: Section 2 summarizes our previous research, which serves as the baseline for this work; Section 3 presents the main contribution of this paper: an extensible framework for qualitative adaptation called *Qualia*; Section 4 describes how the framework was implemented and evaluated using simulations; Section 5 compares our approach with related work; Section 6 discusses challenges in handling multiple concurrent failures, introducing some of our on-going and future work; finally, Section 7 concludes.

## 2 System Identification for Adaptive Systems

In our previous work [23,25], we have applied a requirements engineering perspective to the problem of designing adaptive systems, focusing on developing a qualitative software-centric, feedback loop mechanism as the architecture that operationalizes adaptivity. Feedback loops introduce functionality to a system proper, providing monitoring of specified indicators and making the system aware of its own failures (i.e., aware of when not fulfilling its mandate). In these cases, a possible adaptation solution is to change the value of one or more system parameters which are known to have a positive effect on the necessary indicators.

In Control Theory, quantifying the effects of control input on measured output is a process known as *system identification* [13]. In some cases (e.g., a thermostat), and given the necessary resources, it is possible to represent the equations that govern the dynamic behavior of a system from first principles (e.g., quantitative relations between the amount of gas injected in the furnace and the change in temperature produced by it). For most adaptive information systems, however, such models are overly complex or even impossible to obtain. For this reason, in [23], we have proposed a systematic system identification method for adaptive software systems based on qualitative reasoning.

Our proposal is based on Goal-oriented Requirements Engineering (GORE), which is founded on the premise that requirements are stakeholder *goals* to be

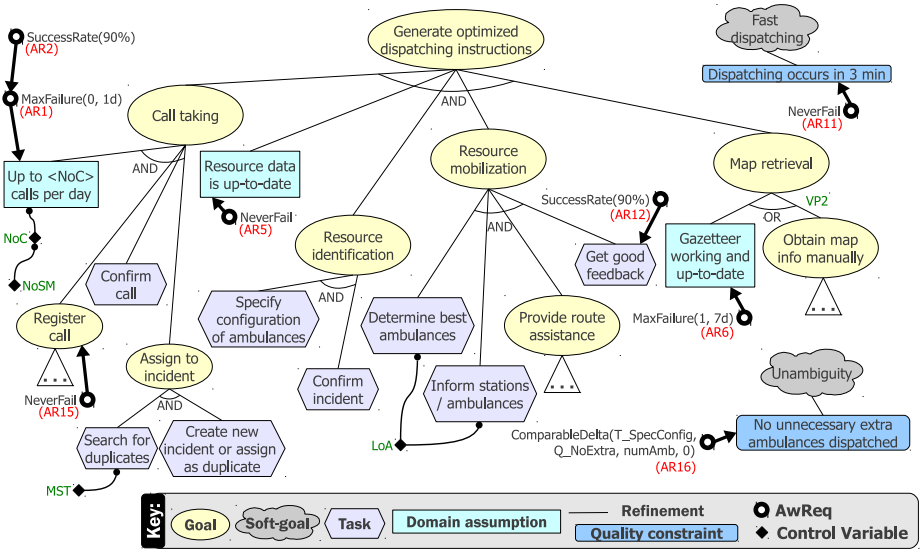


**Fig. 1.** Part of the goal model for the A-CAD system [22] after system identification

fulfilled by the system-to-be along with other actors. Goals are elicited from stakeholders and are analyzed by asking "why" and "how" questions [6]. Such analysis leads to goal models which are partially ordered graphs with stakeholder requirements as roots and more refined goals lower down, following obvious AND/OR Boolean semantics for goal satisfaction. *Goals* are refined until they reach a level of granularity where there are *tasks* an actor (human or system) can perform. On the other hand, *softgoals* are special types of goals that do not have clear-cut satisfaction criteria, and thus refined to measurable *quality constraints* for satisfaction. Finally, *domain assumptions* indicate states of the world that we assume to be true in order for the system to work. All of these elements are part of the ontology for requirements proposed by Jureta et al. [15].

An example of a goal model representing system requirements can be seen in Figure 1, which shows parts of the goal model of an *Adaptive Computer-aided Ambulance Dispatch system* (A-CAD), used as a running example throughout this paper. In the figure, triangles with points of ellipsis represent goal subtrees that are not relevant for the explanations contained herein and, thus, were removed to make the diagram simpler to read. The interested reader can refer to [22] for complete models and descriptions of the A-CAD.

Other than the aforementioned goal model elements, the diagram also shows some of the **indicators** and system **parameters** identified for the A-CAD. In our research, we use *Awareness Requirements* (*AwReqs*) [25] to define indicators of requirements convergence. *AwReqs* represent undesirable situations to which stakeholders would like the system to adapt, in case they happen (e.g., failure of critical requirements). Figure 1 shows eight of the sixteen *AwReqs* identified for the A-CAD, e.g., quality constraint *Dispatch occurs in 3 min* should never fail (*AR11*), *AwReq AR1* should have 90% success rate (*AR2*), etc.

Although *AwReqs* are not performance measurements per se, they are defined in terms of these measures (in the above examples, dispatch time and success rate), setting targets for requirement satisfaction. Currently, our framework uses strictly *AwReqs* as indicators and, therefore, in this papers the terms *indicator* and *AwReq* will be used interchangeably. The approach presented here could, however, be adapted to other kinds of indicators, as long as they are monitored and the system is made aware of their failures.

Parameters can be of two flavors. *Variation points* consist of OR-refinements which are already present in high variability systems (i.e., systems that offer different means of satisfying certain goals) and merely need to be labeled. E.g., the value of `VP2` specifies if the system should assume the *Gazetteer is working and up-to-date* or if staff members should *Obtain map info manually*.

*Control variables* are abstractions over large/repetitive variation points and are represented by black diamonds attached to the elements of the model to which they refer. For instance, `LoA` represents the *Level of Automation* of tasks *Determine best ambulances* and *Inform stations/ambulances*, abstracting over the (repetitive) OR-refinements that would have to be added to them in order to represent such variability. `LoA` is an example of an enumerated variable (possible values are *manual*, *semi-automatic* and *automatic*), whereas `MST` (*Minimum Search Time*), `NoSM` (*Number of Staff Members working*) and `NoC` (*Number of Calls*, which is dependent on `NoSM`) are instead numeric.

Having identified the indicators to monitor and the parameters that can be tuned at runtime, we can finally model the effect changes in the latter have on the former in a qualitative way, which is done by means of differential relations. Considering indicator *AR11* as example, the A-CAD specification contains the following relations (and subsequent descriptions):

$$\Delta\,(AR11/NoSM)\,[0, MaxSM] > 0 \tag{1}$$

$$\Delta\,(AR11/LoA) > 0 \tag{2}$$

$$\Delta\,(AR11/MST)\,[0, 180] < 0 \tag{3}$$

$$\Delta\,(AR11/VP2) < 0 \tag{4}$$

(1) More staff members increases the chance of satisfying *AR11*;
(2) The higher the level of automation, the faster the dispatching;
(3) Increasing the minimum search time makes dispatching take longer;
(4) Obtaining maps manually contributes negatively to a fast dispatching.

As the examples above show, the syntax $\Delta\,(i/p) > 0$ represents the fact that if parameter $p$ is increased, so is indicator $i$. This syntax borrows from Calculus the concept of differential equations: if $i = f(p)$, a positive differential $f' > 0$ means that the greater the value of $p$, the greater the value of $i$. Negative differentials are analogous. Note that for variation points the convention is that they "increase" from left to right. See [23] for further details.

Moreover, equations (1) and (3) exemplify the specification of boundaries for the specified effect, the former using a variable that represents the maximum number of staff members the ambulance service's facilities can hold (to be specified later), the latter using numerical boundaries directly.

Finally, relations referring to the same indicator can be refined to specify: (a) if a change in one parameter has greater effect than another; and (b) if changing more than one parameter at the same time has cumulative effect on the indicator (which is assumed to be the default behavior). In our running example, an order has been established among the effects that different parameters have towards *AR11*, as shown in Equation (5). Absolute values are used in order to properly compare positive and negative effects.

$$\begin{aligned} |\Delta(AR11/VP2)| > |\Delta(AR11/LoA)| > \\ |\Delta(AR11/MST)| > |\Delta(AR11/NoSM)| \end{aligned} \tag{5}$$

In the field of Qualitative Reasoning, there is a spectrum of choices of qualitative representation languages, each of them providing a different level of *precision* (sometimes referred to as *resolution*) [10]. Some examples of qualitative quantity representation languages are [10]:

- *Status abstraction*: represents a quantity by whether or not it is normal;
- *Sign algebra*: represents parameters according to the sign of their underlying continuous parameter — positive (+), negative (−) or zero (0). It is the weakest form of representation that supports some kind of reasoning;
- *Quantity space*: represents continuous values through sets of ordinal relations, providing variable precision as new points of comparison are added;
- *Intervals*: similar to quantity space representation, consists of a variable-precision representation that uses comparison points but also includes more complete information about their ordinal relationship;
- *Order of magnitude*: stratify values according to some notion of scale, such as hyper-real numbers, numerical thresholds or logarithmic scales.

The proposed representation for the information elicited through system identification allows analysts to start with a very low level of precision (e.g., $\Delta(I/P) > 0$, similar to sign algebra) and evolve this specification when more information becomes available (e.g., $\Delta(I/P)[a,b] > 0$, using *landmarks* as boundaries of intervals). Such evolution can happen either horizontally (more information at the same level of precision) or vertically (increasing the precision of a specific information, e.g., $\Delta(I/P) = 2$, meaning $I = 2 \times P$, quantitative precision). The framework proposed in this paper can accommodate different levels of precision, enabling more elaborate adaptation algorithms when more precise information is available. The process and algorithms for refining precision of differential relations among indicators and control variables as the system operates remains an open problem on our to-do list.

## 3 A Framework for Qualitative Adaptation

As we have just seen, system identification adds to a requirements model qualitative information on how changes in system parameters affect indicators that are deemed important by the stakeholders. With this information, it is already

possible to propose an adaptation algorithm for when *AwReqs* fail at runtime, for instance: (1) find all parameters that affect the failed *AwReq* positively; (2) calculate the one(s) with the least negative impact on other indicators; (3) return a new system configuration changing the value of this/these parameter(s).

In this paper, we address two particular limitations of our current approach:

- There are still a few pieces of information missing regarding the requirements for adaptation. E.g., considering the algorithm proposed in the previous paragraph, the following questions (among others) are still unanswered: how many parameters should be changed and by how much? When calculating negative impact to other indicators, should priorities (e.g., [17], § 3.3) among them be considered? What if the *AwReq* fails again, should the previous adaptation attempts be taken into account when deciding a new one?
- Moreover, the adaptation algorithm exemplified above is just one of many possible algorithms that can be used given the available qualitative information about the system's dynamic behavior. Among the many possible algorithms, the choice of which to use should belong to the stakeholders and domain experts and, thus, be part of the system requirements specification. The adaptation framework should be able to accommodate this.

Therefore, in this paper we propose a framework to operationalize adaptation at runtime based on this qualitative information. We call this framework *Qualia* (**Quali**tative **a**daptation). When made aware of a failure in an indicator, *Qualia* adapts the system by conducting eight activities, as shown in Figure 2 and described below (the numbers below match the ones in the figure):

1. One or more parameters modeled during system identification are chosen;
2. Based on the relation of this/these parameter(s) with the failed indicator, *Qualia* decides by how much it/they should be changed;
3. The chosen parameter(s) are then incremented (consider decrements as negative increments for simplicity) by the calculated value(s);
4. The framework waits for the change to produce any effect on the indicator;
5. *Qualia* evaluates the indicator again after the waiting time;
6. In each cycle, *Qualia* may learn from the outcome of this change, possibly evolving the adaptation mechanism and updating the model;
7. Finally, it decides whether the current indicator evaluation is satisfactory and either concludes the process or starts over;
8. If it decides to start over, it reassesses the way adaptation was conducted in the previous cycles, possibly adapting itself for the following cycle.

To accommodate the different levels of precision, we propose an extensible framework by defining an interface for each activity in the process of Figure 2 and providing default implementations that assume only the minimum amount of information is available. Then, we allow designers to create and plug-in new ***procedures*** into *Qualia*, possibly requiring more information about the system in order to be applicable. We use the term ***adaptation algorithm*** to refer to the set of procedures chosen to support the adaptation process. In the requirements specification, analysts should indicate which *adaptation algorithms* to use in response to each indicator failure.

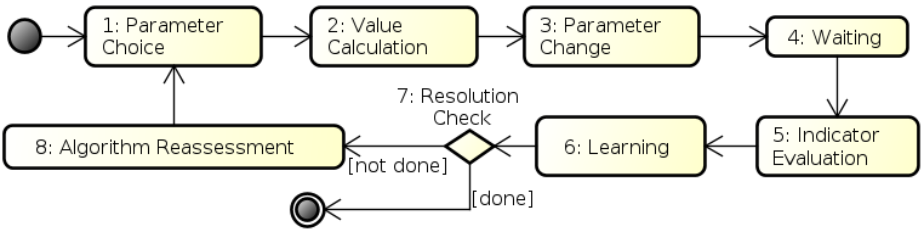


**Fig. 2.** The adaptation process followed by the *Qualia* framework

In the following sub-sections, we present three *adaptation algorithms*: the Default Algorithm (§ 3.1), the Oscillation Algorithm (§ 3.2) and the PID-based Algorithm (§ 3.3). To illustrate how *Qualia* can accommodate different levels of precision, we also propose different procedures (§ 3.4) for the first step of its process (*Parameter Choice*). An important remark here is that we do not make any claim on which adaptation algorithm is better suited for any particular context, but instead we just illustrate how this framework can be extended as needed. The choice of algorithm to use is the responsibility of the analysts.

### 3.1 The Default Algorithm

As mentioned earlier, when adapting the system, *Qualia* executes the algorithm that has been associated with the failure at hand by stakeholders or domain experts. When a particular algorithm is not specified, *Qualia* executes the *Default Algorithm*, which requires minimum information from the requirements models:

- **Indicators:** *Qualia* has to be notified of indicator failure, hence the model should specify what are the relevant indicators in a way such that another component of the feedback loop is able to monitor them. For this purpose, we use *AwReqs* (cf. Section 2) and its monitoring infrastructure [25];
- **Parameters:** to adapt to an indicator failure, there should be at least one related parameter. Section 2 also described how this information is specified through differential equations;
- **Unit of increment:** each numeric parameter must specify its unit of increment, because *Qualia* will not be able to guess it.

The *unit of increment* is important for the comparison among indicator/parameter relations. E.g., the comparison $|\Delta(AR11/MST)| > |\Delta(AR11/NoSM)|$ presented earlier as part of Equation (5), should be complemented by $U_{NoSM} = 1$ and $U_{MST} = 10\,seconds$, meaning that changing `MST` *by 10s* improves *AR11* more than changing `NoSM` *by 1 staff member*. Moreover, enumerated parameters must be ordered (cf. [23]) and their unit of increment defaults to choosing the next value in the order.

The *Default Algorithm* is composed of eight default procedures, one for each activity of the process depicted earlier in Figure 2 (again, the numbers below match the numbers in the figure):

1. *Random Parameter Choice*: picks one parameter randomly from the set of parameters related to the failed indicator, considering those which can still be incremented by at least one unit (i.e., are within their boundaries).
2. *Simple Value Calculation*: decides the increment value for the chosen parameter, by multiplying the value of the parameter's unit of increment $U$ by the indicator's increment coefficient $K$, returning the value $V = K \times U$.

   The increment coefficient is an optional parameter (with default value $K = 1$) that can be associated to each indicator in the specification to determine how critical it is to adapt to their failures. Higher values of $K$ will produce more significant changes, but the requirements engineer should be aware of the risks of overshooting. Note also that parameters should never exceed their boundaries.
3. *Simple Parameter Change*: changes the chosen parameter by the calculated value, at the *class* level.

   The *class/instance* terminology is inherited from our previous work [25]: changes at the *class* level will affect the system "from now on", whereas changes at the *instance* level only affect the current execution of the system.
4. *Simple Waiting*: waits until the next time the indicator is evaluated by the monitoring component of the feedback loop.
5. *Boolean Indicator Evaluation*: verifies if, after executing the first four steps of the process, the next time the indicator succeeded.
6. *No Learning*: in the *Default Algorithm*, learning is skipped.
7. *Simple Resolution Check*: stops the process if the outcome of the indicator evaluation (step 5) was positive, otherwise it iterates.
8. *No Algorithm Reassessment*: the *Default Algorithm* does not reassess or adapts itself, but always executes the same procedures in every iteration.

Let us illustrate the above algorithm using the A-CAD. Imagine that for a given emergency call received, an ambulance was not dispatched within three minutes, breaking indicator (*AwReq*) *AR11* (quality constraint *Dispatching occurs in 3 min* should never fail). Available parameters to improve this indicator are `NoSM`, `LoA`, `MST` and `VP2` (assuming all are within boundaries). For this example, consider that the *Random Parameter Choice* procedure chose `MST`.

Imagine further that the specification says that $K_{AR11} = 2$ and we know that $U_{MST} = 10s$ and, moreover, Equation (3) says that `MST` contributes negatively to *AR11*. Therefore, the *Simple Value Calculation* procedure decides to decrease `MST` by $V = 2 \times 10s = 20s$ and, as a consequence, the *Simple Parameter Change* procedure does so at the *class* level, i.e., for all dispatches following the one that did not satisfy *AR11*, until further notice.

Since *AR11* is evaluated at every dispatch, the next dispatch will resume the process (*Simple Waiting* procedure) and the *Boolean Indicator Evaluation* procedure will check if, after `MST` was reduced by 20s, the next dispatch took less than 3 minutes to complete. If the 20s reduction was effective, then the *Simple Resolution Check* procedure will stop the process; otherwise it will repeat the same procedures as above.

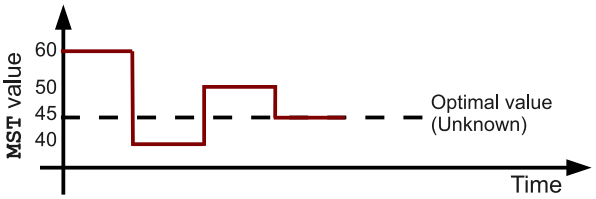


**Fig. 3.** A scenario of use of the *Oscillation Algorithm* in the A-CAD

As mentioned earlier, the requirements engineer should include in the requirements specification which algorithm — i.e., which set of procedures — should be used for each system failure. The *Default Adaptation Algorithm* can be represented by the empty set $\varnothing$, meaning that all the default procedures described above will be used. Other algorithms, as will be described next, are represented by naming the non-default procedures that compose them: the specified procedures replace their default counterparts (the one with the same interface), keeping the default ones that have not been replaced.

### 3.2 The Oscillation Algorithm

One of the desired characteristics of control systems is to avoid overshooting its control inputs. For instance, if an ambulance dispatch takes $3min10s$, we decide to reduce `MST` from $60s$ to $0s$ and the next dispatch takes only $2min10s$, we have overshot `MST`'s decrement by $50s$. Granted, this overshoot could be corrected whenever some other indicator (e.g., *AR16*, which controls if unnecessary ambulances are sent to incident sites) fails and `MST` is chosen to be incremented. Still, a good adaptation algorithm tries to avoid overshooting in the first place and, in what follows, we present one such algorithm.

The *Oscillation Algorithm* works as depicted in Figure 3: back to the *AR11* / `MST` scenario, imagine that given the current circumstances, the optimal[1] value for `MST` is $45s$. The controller obviously does not know it, so when *AR11* fails, it decreases `MST` to $40s$, which actually solves the problem. However, instead of stopping here, the algorithm **assumes to have overshot the change**, and thus starts incrementing the same parameter in the opposite direction, using **half of the previous increment value**. When `MST` is set to $50s$, *AR11* fails again, which makes the controller switch increment direction and halve the increment value one more time. This process goes on until one of the following conditions:

– The parameter is incremented to a value that it has already assumed before, which means that we should be very close to the optimal value. E.g., if we

[1] Here, we consider "optimal" the smallest change that fixes the problem, because we assume every adaptation brings negative side effects to other indicators. If this is not the case, one could just set the parameter to its maximum (minimum) value from the start and no adaptation is necessary.

continue the oscillations shown in Figure 3, `MST` will assume values $47s$, $46s$, $45s$ and then stop;
- The algorithm has already performed the *maximum number of oscillations*, which is an optional attribute that can be assigned to a specific *AwReq* or to the entire goal model. Here, we consider each inversion of increment direction to be an oscillation (three, in the figure);
- The increment value is halved to an amount that is lower than the *minimum change value* of the parameter at hand (optional). For instance, Figure 3 represents the case in which this value is $5s$. Note that, for integer variables such as `MST`, 1 is the lowest possible value.

In order to tune this algorithm, the framework also allows for the specification of parameters' *halving factors* different from the default value of 0.5. When oscillating, the increment value will be multiplied by the specified factor. The table below summarizes the *Oscillation Algorithm*:

| **Oscillation Algorithm** | |
|---|---|
| **Specification** | {*Oscillation Parameter Choice*, *Oscillation Value Calculation*, *Oscillation Resolution Check*} |
| **Properties** | – *Maximum number of oscillations* (optional);<br>– *Minimum change value* (optional);<br>– *Halving factors* (default = 0.5). |

The *Oscillation Resolution Check* procedure assumes to have overshot when the problem is fixed and begins the oscillations, whereas the *Oscillation Value Calculation* procedure is responsible for determining when the value should be increased or decreased and when it should be halved. The *Oscillation Parameter Choice* procedure replaces the default, random one by choosing the parameter randomly at first, but then maintaining the choice until the end of the oscillations. Later, in Section 3.4, other parameter choice procedures will be illustrated, some of which could also be used here.

### 3.3 The PID-Based Algorithm

As mentioned in Section 1, our framework's controller is inspired by control-theoretic concepts, notably the Proportional-Integral-Differential (PID) controller. This controller is widely used in the process control industry and provides an efficient algorithm (described in most books on Control Theory, e.g., [13], Chapter 9) to keep a single output of the target system as close as possible to the specified, single reference input.

The question that arises then is the following: given its proven efficacy, would it be possible to use the actual PID algorithm in our models? First, since the PID algorithm works with *single input/single output* (SISO) and information systems usually have *multiple inputs/multiple outputs* (MIMO), this algorithm would work well only when the analyst can identify, for a given indicator, one single parameter whose changes have a significant effect in the indicator's outcome. Moreover, since this algorithm requires a numeric value for the control error

and *AwReqs* (our indicators) are somewhat of a Boolean nature ($success = true|false$), we need a way to extract a numeric value from them.

As detailed in [25], *AwReqs* can be divided in three categories: *Delta AwReqs* impose constraints over properties of the domain (e.g., "number of ambulances at the incident should not be greater than the number specified"), *Aggregate AwReqs* determine requirements' success rates ("75% of the ambulances should arrive within 8 minutes"), and *Trend AwReqs* impose constraints over aggregated success rates over time ("success rate of *Get good feedback* should not decrease two weeks in a row"). *Qualia* will extract numeric control errors from these types of *AwReqs* as follows:

- *Delta AwReqs* : if the property is numeric, calculate the difference between desired and monitored values. In the above example, they are the specified number and the actual number of ambulances at the incident;
- *Aggregate AwReqs* : calculate the difference between the desired and actual success rates. Note that *AwReqs* of the form "*R* should never fail" can be translated into "*R* should have 100% success rate";
- *Trend AwReqs* : calculate the difference between the last two measured success rates. In the above example, if the rate decreases in 7% in the first week and then again by 4% in the second, the control error is 4%.

If the *AwReq* in question follows one of these patterns, the *PID Algorithm* can be used. As the table below indicates, the algorithm affects *Qualia*'s procedures for value calculation, indicator evaluation and resolution check.

| **PID-based Algorithm** | |
|---|---|
| **Specification** | {*PID Value Calculation*, *PID Indicator Evaluation*, *PID Resolution Check*} |
| **Properties** | None. |

### 3.4 Other Procedures

In the beginning of Section 3, we have stated that *Qualia* supports different levels of precision by allowing for new procedures to be implemented and plugged in to the framework for each of the eight activities in its adaptation process (Figure 2). To illustrate how our proposed framework can be extended, we focus here on the *Parameter Choice* activity and describe new procedures that execute it differently from the default one, especially in the presence of more precise information in the specification:

- *Shuffle Parameter Choice*: with the same amount of information used by the *Random Parameter Choice* procedure, this procedure randomly puts the system parameters in order during the first cycle and picks the next one using this pre-defined sequence when switching parameters is required.

A new property — *repeat policy* — determines when the parameter choice procedure should repeat the last used value or switch to a different one. Its default value is *repeat while incrementable*, but it can be set to *repeat M times*, where $M$ is also configurable.

- *Ordered Effect Parameter Choice*: if differential relations regarding the indicator in question have been refined to provide comparison of their effect (as explained in Section 2), this procedure orders the parameters according to their effect on the indicator and uses them in this order.

  Other than the *repeat policy* property, an *order* property is also relevant to this procedure, specifying if relations should be placed in *ascending* or *descending* order of effect (depending if stakeholders would like to start with the parameters that have the greatest or the smallest effect on the indicator). Moreover, if the set of relations concerning an indicator is only partially ordered, the *remaining parameters* property specifies if the non-ordered relations should be excluded from the list or shuffled at the end of it.

- *Ordered Side Effect Parameter Choice*: in case priorities among indicators are given (using, e.g., [17]), this procedure orders the parameters according to the priority of the indicators to which the parameter change would contribute negatively. It is particularly suitable for lower-priority indicators that can, in general, be sacrificed to maintain high-priority ones.

  The *side effect calculation* property specifies if the *average* of the priorities of the indicators that suffer side effects should be calculated or if only the *highest* priority should be considered. The *remaining parameters* and *order* properties are also relevant here.

- *Ordered Maturation Time Parameter Choice*: domain experts can specify an optional attribute to differential relations called *maturation time*, which indicates how long it takes for the changes in the related parameter to take effect in the related indicator. Take, for instance, the scenario described in Section 3.1, and say *Qualia* has chosen `NoSM` instead of `MST` to adapt for the failure in *AR11*. Hiring and training a new staff takes a few days and, thus, the framework should wait for this specified time before continuing. Hence, this procedure will order the relations by their *maturation times*. As with the other ordered parameter choice procedures, the *order* parameter is also relevant here. Notice that relations' *maturation time* attribute also affects the *Default Waiting* procedure, illustrated earlier.

Finally, all of the procedures presented above can be further customized by the *number of parameters* property, which defaults to 1, but can be set to any positive integer, or even *all* parameters, mimicking the behavior of a *multiple input, single output* (MISO) control system. As demonstrated throughout this section, our proposed framework can be extended as needed by requirements engineers, depending on stakeholder requirements.

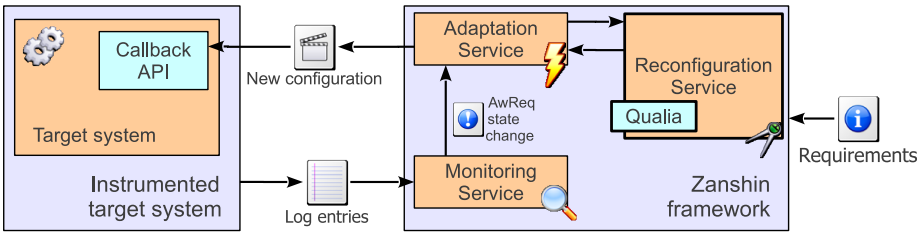


**Fig. 4.** Overview of the *Zanshin* framework and the addition of *Qualia*

## 4 Implementation and Evaluation

To evaluate *Qualia*, the framework described in Section 3, we have implemented it as a component of the *Zanshin* framework. Proposed in [24] (and available at `http://github.com/vitorsouza/Zanshin`), *Zanshin* applies an Event-Condition-Action (ECA)-based process to adapt to *AwReq* failures by effecting changes in other requirements in the model. Figure 4 shows an overview of the framework, highlighting with thicker borders the components added by this paper.

The monitoring infrastructure of our previous work [25] has been used to identify *AwReq* (indicator) failures from the *log entries* of the *instrumented target system*. The *Monitoring Service* will then notify *Zanshin*'s *Adaptation Service* about *AwReq* state changes (e.g., *AR11* has succeeded, *AR11* has failed, etc.). In some cases, based on the system *requirements*, this component might conclude that reconfiguration should be used, and asks the *Reconfiguration Service* for one of its registered *reconfiguration strategies*. *Qualia* is registered as a strategy, but *Zanshin* allows for other reconfiguration frameworks to be plugged-in (e.g., some existing frameworks are described as related work in Section 5). After the selected *reconfiguration strategy* produces a new configuration, the *Adaptation Service* sends it to the *target system* through a *callback API*.

The framework was implemented as a set of OSGi bundles and its requirements meta-models were specified using the Eclipse Modeling Framework (EMF), as shown in Figure 5. Because of space constraints, the meta-model for requirements specifications in *Zanshin* will not be reported here, but the reader can refer to [24] for its description. Figure 5 shows four elements from A-CAD's goal model, which were depicted earlier in Figure 1: root goal *Generate optimized dispatching instructions*, softgoal *Fast dispatching*, its quality constraint (QC) *Dispatching occurs in 3 min* and *AwReq AR11*, which targets that QC.

In the `<strategies>` tag, we can see that *Qualia* has been selected as *reconfiguration strategy* for failures of *AR11*. Further below, the `<configuration>` tag specifies parameter `MST` as a numeric control variable (`ncv`), with $U_{MST}$ set to `10` and initial value `60`. Finally, the `<relations>` tag represents the differential relation shown back in Equation (3): $\Delta(AR11/MST)[0, 180] < 0$.

Based on experimental evaluation methods of Design Science [14], we developed simulations to mimic the behavior of the A-CAD in different possible runtime scenarios, in order to evaluate the framework's response to system failures.

```
<?xml version="1.0" encoding="UTF-8"?>
<acad:AcadGoalModel ...>
 <rootGoal xsi:type="acad:G_GenDispatch">
  ...
  <children xsi:type="acad:S_FastDispatch"/> <!--7-->
  ...
  <children xsi:type="acad:Q_Dispatch" softgoal="//@rootGoal/@children.7"/>
       <!--12-->
  ...
  <children xsi:type="acad:AR11" target="//@rootGoal/@children.12"
      incrementCoefficient="2">
   <condition xsi:type="model:ReconfigurationResolutionCondition"/>
   <strategies xsi:type="model:ReconfigurationStrategy" algorithmId="qualia
       ">
    <condition xsi:type="model:ReconfigurationApplicabilityCondition"/>
   </strategies>
  </children> <!--26-->
 </rootGoal>
 <configuration>
  <parameters xsi:type="acad:CV_MST" type="ncv" unit="10" value="60" metric
      ="integer"/>
 </configuration>
 <relations indicator="//@rootGoal/@children.26" parameter="//
     @configuration/@parameters.0" lowerBound="0" upperBound="180"
     operator="ft" />
</acad:AcadGoalModel>
```

**Fig. 5.** Part of the A-CAD requirements specified as an EMF model

The simulations send logging messages to the *Monitoring Service*, equivalent to the ones that would have been sent by a real system, indicating a failure. For instance, one of the implemented simulations produces log entries that indicate that *Dispatching occurs in 3 min* was not satisfied, which triggers a failure of *AR11*. Based on the EMF model of Figure 5, *Zanshin* activates *Qualia*, which executes its *Default Algorithm*, described and illustrated in Section 3.1.

The result of this particular simulation is shown in Figure 6. In this output, `S` represents the simulation (i.e., the *target system*), `Z` is *Zanshin* and `Q` is for *Qualia*. Figure 5 shows that *Qualia* selected `MST` and reduced its value to 40*s*, but another failure in *AR11* followed, and therefore the parameter was again reduced to 20*s*, which solved the problem.

Another simulation uses a randomly generated goal model with different number of parameters (from 100 to 1000, scaling up by 100 elements each time), all of them related to a failing *AwReq*. *Zanshin* and *Qualia* were timed in ten sequential executions of this simulation and average times for each number of parameters, as shown in Table 1, indicate linear scalability. In effect, by analyzing *Qualia*'s default algorithm, one can conclude that its complexity is $O(N \times R)$, where $N$ is the *number of parameters* to choose and $R$ is the number of differential relations in the model. With proper data structures, however, this complexity can be further reduced. In [24], we showed that *Zanshin* also scales linearly to goal models of increasing number of elements.

*Qualia* and *Zanshin* are part of a broader research proposal for the design of adaptive systems using a control theoretic perspective founded on requirements. Further evaluation efforts are in our future research plans, including experiments with actual running systems, user surveys to evaluate our methods and modeling language, then finally full-fledged case studies with partners in industry.

```
S: A dispatch took more than 3 minutes!
Z: State change: AR11 (ref. Q_Dispatch) -> failed
Z: (S1) Created new session for AR11
Z: (S1) Selected strategy: ReconfigurationStrategy
Z: (S1) Exec. ReconfigurationStrategy(qualia; class)
Q: Parameters chosen: [CV_MST]
Q: To inc/decrement in the chosen parameters: [20]
S: Instruction received: apply-config()
S: Parameter CV_MST should be set to 40
Z: (S1) The problem has not yet been solved...
-----------------------------------------------------
S: A dispatch took more than 3 minutes!
Z: State change: AR11 (ref. Q_Dispatch) -> failed
Z: (S1) ...
Q: Parameters chosen: [CV_MST]
Q: To inc/decrement in the chosen parameters: [20]
S: Instruction received: apply-config()
S: Parameter CV_MST should be set to 20
-----------------------------------------------------
S: A dispatch took less than 3 minutes.
Z: State change: AR11 (ref. Q_Dispatch) -> succeeded
Z: (S1) Problem solved. Session will be terminated.
```

**Fig. 6.** Result of the A-CAD simulation in which *AR11* fails

**Table 1.** Average time (in milliseconds) for executions of *Qualia* and *Zanshin*

| Elements | *Qualia* | *Zanshin* |
|---|---|---|
| 100 | 40.4 | 1,187.5 |
| 200 | 1,064.4 | 4,416.3 |
| 300 | 2,098.5 | 10,122.3 |
| 400 | 3,132.2 | 11,851.1 |
| 500 | 4,164.8 | 13,097.7 |

| Elements | *Qualia* | *Zanshin* |
|---|---|---|
| 600 | 5,212.6 | 14,323.4 |
| 700 | 6,244.4 | 15,568.3 |
| 800 | 7,283.3 | 18,811.1 |
| 900 | 8,314.6 | 20,621.6 |
| 1000 | 9,169.0 | 26,135.4 |

## 5 Related Work

In the field of requirements-driven adaptation two well-known proposals are the RELAX framework [27] and FLAGS [1], the former based on structured natural language whereas the latter uses goal models. Both of them use fuzzy logic in order to transform "crisp" (invariant) requirements into "relaxed" ones in order to capture uncertainty. Additionally, in FLAGS, adaptive goals define countermeasures to be executed when goals are not attained, using ECA rules. The GAAM approach [21] models quantifiable properties of the system as attributes, while specifying the order of preference of adaptation actions towards goals in a preference matrix, and the desired levels of attributes of each goal in an aspiration level matrix.

Several approaches in the literature propose adaptation through *reconfiguration*, i.e., switching the system's behavior by finding a new configuration for its parameters. Wang & Mylopoulos [26] propose algorithms that suggest a new configuration without components that have been diagnosed as responsible for a failure; Nakagawa et al. [19] developed a compiler that generates architectural configurations by performing conflict analysis on goal models; Fu et al. [11] use reconfiguration to repair systems based on an elaborate state-machine diagram

that represents the life-cycle of goal instances at runtime; Peng et al. [20] assign preference rankings to softgoals and determine the best configuration using a SAT solver; Khan et al. [16] apply Case-Based Reasoning to find the best configuration; Dalpiaz et al. [5] propose an algorithm that finds all valid variants to satisfy a goal and compares them based on their compensation/cancelation cost and benefit (e.g., contribution to softgoals).

Like us, Filieri et al. [8] have also applied control theory to the problem of designing adaptive systems with a requirements perspective, focusing on adapting to failures in reliability and modeling requirements using Discrete Time Markov Chains (DTMCs). There, transitions are labeled with control variables, whose values can be set by a controller that decides the system's settings in order to keep satisfying the requirements. Well established control theoretic tools are used to design such controller and the authors claim the approach can be extended to deal with failures of different nature. An extension [9] proposes a more efficient solution for dynamic binding of components and an auto-tuning procedure.

Our work is also related to design-time trade-off approaches, considering that they could be tailored for the type of reasoning needed for run-time adaptation. For instance, Heaven & Letier [12] use stochastic simulation in order to generate quality values which are used to compute objective functions over a goal model, simulating design decisions in order to compare and optimize them.

Compared to the above approaches, the novelty in our proposal is the use of qualitative information about requirements, allowing analysts to start with the minimum information at hand and add more as further details about the system become available. In many cases, quantitative approaches might be difficult or even impossible to apply accurately and reliably due to the relativity of numerical values, incorrect mathematical judgment, non-linearity of value functions, etc. [7]. Furthermore, we advocate for expressive, but simple requirements models, believing that heavy formalisms such as linear temporal logic, fuzzy logic and DTMCs can, in some cases, place unnecessary burden on developers.

Qualitative reasoning has also been used by others to analyze system requirements in a similar fashion to what we propose. Menzies & Richardson [18] propose a matrix that depicts the contribution of process actions to interesting indicators (positive, negative, unknown or none) and use stochastic simulation to analyze this matrix and decide the best choice of actions, considering stakeholder-assigned utility values for each indicator. The proposed matrix conveys the same kind of information as our differential equations, albeit our models have considerably more expressive power. Elahi & Yu [7] also focus on requirements trade-offs at design-time, making pair-wise comparison of alternatives with respect to goals that were selected as indicators. We propose a more concise and expressive means to represent such comparisons, namely differential equations. Furthermore, both approaches focus on design-time decisions whereas our proposal targets run-time adaptation.

Finally, the use of control-theoretic concepts in our research (advocated by recent survey/roadmap papers such as [3,4]) comes from the fact that, in order to be adaptive, systems need to implement some kind of monitor-adapt feedback

loop. Given our Requirements Engineering perspective, our approach makes explicit in the models both requirements for monitoring (*indicators*/*AwReqs*) and adaptation (the chosen *adaptation algorithms*), allowing developers to design adaptive systems all the way from requirements to implementation.

## 6 Discussion and Future Work

The models proposed in this approach are a first step towards a comprehensive method for the specification of adaptation requirements based on GORE and qualitative reasoning techniques. Moreover, the *Qualia* framework offers a prototype for the operationalization of such requirements at runtime, alleviating developers of most of the effort of implementing the features of a feedback loop. Nonetheless, there is still a lot of work to be done, especially if we intend to apply this research in practice, on real software development projects.

One assumption that might threaten the applicability of our proposal in more complex systems is that of variable independence. Our proposed language (cf. § 2) represents how changes in single *parameters* affect single *indicators*, whereas in complex, adaptive systems, parameters (or indicators) cannot be assumed to be independent of one another. Nonetheless, this simplification is not accidental. State-of-the-art methods for modeling and controlling *multiple inputs/multiple outputs* (MIMO) control systems — such as state/output feedback and Linear Quadratic Regulator (see [28], § 3.4) — can be very complex and many software projects may not dispose of the necessary (human/time) resources to produce models with such degree of formality. As mentioned in the previous section, our approach is intended to be less heavy-handed in the formalism, while at the same time allowing analysts to model the requirements for the system's adaptation based on a feedback loop architecture.

Another considerable limitation of our current approach is the fact that its adaptation process responds to failures of single indicators (*AwReqs*) and does not consider the scenario in which multiple indicators fail concurrently and one failed indicator's adaptation might have an influence in another's. Procedures like *Ordered Side Effect Parameter Choice*, together with the specification of indicators' priorities (e.g., [17]), can help in avoiding undesirable situations such as focusing on less-critical failures or even deadlocks, but more direct consideration of concurrent failures is necessary to guarantee some level of consistency.

Therefore, we are currently working on extending *Qualia* by including a *priority queue* that would make the framework deal with more important failures first (in case, e.g., large *maturation times* create long-running adaptation cycles); the introduction of *locks* (as in database transaction processing) that would prevent certain parameters form being changed because they affect indicators that have been locked; and the ability of dealing with *multiple failures in a single adaptation loop*. The latter would require new procedure implementations, especially for the activities of *Parameter Choice* (e.g., choose parameters that do not have negative effects on all failed indicators), *Value Calculation* (e.g., considering multiple *increment coefficients*), *Waiting* (e.g., consider the *maturation time* of all failed indicators) and, obviously, *Indicator Evaluation*.

On the methodology side, improvements such as the ellaboration of a graphical representation in the goal model might make the adaptation specifications easier to read; pre-defined policies can abstract the choice of adaptation algorithm and its many attribute values in mnemonics such as "aggressive", "conservative", etc.; moreover, a CASE tool would also greatly help analysts in following our proposed approach.

Finally, more experiments, especially with real systems, would help us examine the kinds of adaptation scenarios that are possible and, thus, propose sensible implementations for the *Algorithm Reassessment* and *Learning* activities, which have received little attention so far. These would involve a repository of past experiences, which would record failures, what was done to adapt and the outcome of the adaptation. Then, on-line or off-line learning procedures could query this repository in order to evolve the specification in general.

## 7 Conclusions

In this paper, we have proposed a framework within which a failure of requirements leads to a new behavior obtained by selecting a new variant of the system's goal model, and/or new values for its control variables. The proposed controller is inspired by control theoretic concepts, notably the PID controller, recast in qualitative terms and using goal models to define the desired output and the space of possible behaviors for obtaining it. To validate our work, we have implemented our framework and simulated its algorithms using different scenarios.

Our proposal is founded on the thesis that requirements should be at the very center of any adaptation mechanism, determining what constitutes normal behavior, what is to be monitored and what are possible compensations in cases of deviations. Following Berry et al.'s *envelope of adaptability* [2], systems are only able to adapt to "the extent to which the adaptation analyst can anticipate the domain changes to be detected and the adaptations to be performed".

Moreover, by separating the standard, "normal behavior" from the requirements for monitoring and adaptation, our approach provides abstractions that can facilitate modeling and communication of requirements for systems that are supposed to have several adaptation capabilities. As with any abstraction in Software Engineer, our proposals should be applied when the benefits of having these concepts in the models outweigh the cost of using the approach.

As the discussions illustrated earlier, the work presented here is the first step towards a full qualitative adaptation framework that can operationalize most stakeholder requirements for adaptation using a generic feedback loop.

**Acknowledgments.** We are grateful to our Trento colleagues for their feedback to this work, which has been supported by the ERC advanced grant 267856 "Lucretius: Foundations for Software Evolution" (unfolding during the period of April 2011 – March 2016) — `http://www.lucretius.eu`.

## References


1. Baresi, L., Pasquale, L., Spoletini, P.: Fuzzy Goals for Requirements-driven Adaptation. In: Proc. of the 18th IEEE International Requirements Engineering Conference, pp. 125–134. IEEE (2010)
2. Berry, D.M., Cheng, B.H.C., Zhang, J.: The Four Levels of Requirements Engineering for and in Dynamic Adaptive Systems. In: Proc. of the 11th International Workshop on Requirements Engineering: Foundation for Software Quality, pp. 95–100 (2005)
3. Brun, Y., Di Marzo Serugendo, G., Gacek, C., Giese, H., Kienle, H., Litoiu, M., Müller, H., Pezzè, M., Shaw, M.: Engineering Self-Adaptive Systems through Feedback Loops. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 48–70. Springer, Heidelberg (2009)
4. Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Di Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H.M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H.A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J.: Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 1–26. Springer, Heidelberg (2009)
5. Dalpiaz, F., Giorgini, P., Mylopoulos, J.: Adaptive socio-technical systems: a requirements-based approach. In: Requirements Engineering, pp. 1–24 (2012)
6. Dardenne, A., van Lamsweerde, A., Fickas, S.: Goal-directed Requirements Acquisition. Science of Computer Programming 20(1-2), 3–50 (1993)
7. Elahi, G., Yu, E.S.K.: Requirements Trade-offs Analysis in the Absence of Quantitative Measures: A Heuristic Method. In: Proc. of the 2011 ACM Symposium on Applied Computing, pp. 651–658. ACM (2011)
8. Filieri, A., Ghezzi, C., Leva, A., Maggio, M.: Self-Adaptive Software Meets Control Theory: A Preliminary Approach Supporting Reliability Requirements. In: Proc. of the 26th IEEE/ACM International Conference on Automated Software Engineering, pp. 283–292. IEEE (2011)
9. Filieri, A., Ghezzi, C., Leva, A., Maggio, M.: Reliability-driven dynamic binding via feedback control. In: Private Communication (2012)
10. Forbus, K.D.: Qualitative Reasoning. In: Computer Science Handbook, 2nd edn., ch. 62. Chapman and Hall/CRC (2004)
11. Fu, L., Peng, X., Yu, Y., Zhao, W.: Stateful Requirements Monitoring for Self-Repairing of Software Systems. Tech. rep., FDSE-TR201101, Fudan University, China (2010), `http://www.se.fudan.sh.cn/paper/techreport/1.pdf`
12. Heaven, W., Letier, E.: Simulating and Optimising Design Decisions in Quantitative Goal Models. In: Proc. of the 19th IEEE International Requirements Engineering Conference, pp. 79–88. IEEE (2011)
13. Hellerstein, J.L., Diao, Y., Parekh, S., Tilbury, D.M.: Feedback Control of Computing Systems, 1st edn. Wiley (2004)
14. Hevner, A.R., March, S.T., Park, J., Ram, S.: Design Science in Information Systems Research. MIS Quarterly 28(1), 75–105 (2004)
15. Jureta, I., Mylopoulos, J., Faulkner, S.: Revisiting the Core Ontology and Problem in Requirements Engineering. In: Proc. of the 16th IEEE International Requirements Engineering Conference, pp. 71–80. IEEE (2008)

16. Khan, M.J., Awais, M.M., Shamail, S.: Enabling Self-Configuration in Autonomic Systems using Case-Based Reasoning with Improved Efficiency. In: Proc. of the 4th International Conference on Autonomic and Autonomous Systems, pp. 112–117. IEEE (2008)
17. Liaskos, S., McIlraith, S., Sohrabi, S., Mylopoulos, J.: Representing and reasoning about preferences in requirements engineering. Requirements Engineering 16(3), 227–249 (2011)
18. Menzies, T., Richardson, J.: Qualitative Modeling for Requirements Engineering. In: Proc. of the 30th Annual IEEE/NASA Software Engineering Workshop, pp. 11–20. IEEE (2006)
19. Nakagawa, H., Ohsuga, A., Honiden, S.: gocc: A Configuration Compiler for Self-adaptive Systems Using Goal-oriented Requirements Description. In: Proc. of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, pp. 40–49. ACM (2011)
20. Peng, X., Chen, B., Yu, Y., Zhao, W.: Self-Tuning of Software Systems through Goal-based Feedback Loop Control. In: Proc. of the 18th IEEE International Requirements Engineering Conference, pp. 104–107. IEEE (2010)
21. Salehie, M., Tahvildari, L.: Towards a Goal-Driven Approach to Action Selection in Self-Adaptive Software. Software: Practice and Experience 42(2), 211–233 (2012)
22. Silva Souza, V.E.: An Experiment on the Development of an Adaptive System based on the LAS-CAD. Tech. rep., University of Trento (2012), http://disi.unitn.it/~vitorsouza/a-cad/
23. Silva Souza, V.E., Lapouchnian, A., Mylopoulos, J.: System Identification for Adaptive Software Systems: A Requirements Engineering Perspective. In: Jeusfeld, M., Delcambre, L., Ling, T.-W. (eds.) ER 2011. LNCS, vol. 6998, pp. 346–361. Springer, Heidelberg (2011)
24. Silva Souza, V.E., Lapouchnian, A., Mylopoulos, J.: (Requirement) Evolution Requirements for Adaptive Systems. In: Proc. of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, pp. 155–164. IEEE (2012)
25. Silva Souza, V.E., Lapouchnian, A., Robinson, W.N., Mylopoulos, J.: Awareness Requirements for Adaptive Systems. In: Proc. of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, pp. 60–69. ACM (2011)
26. Wang, Y., Mylopoulos, J.: Self-Repair through Reconfiguration: A Requirements Engineering Approach. In: Proc. of the 2009 IEEE/ACM International Conference on Automated Software Engineering, pp. 257–268. IEEE (2009)
27. Whittle, J., Sawyer, P., Bencomo, N., Cheng, B.H.C., Bruel, J.M.: RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems. In: Proc. of the 17th IEEE International Requirements Engineering Conference, pp. 79–88. IEEE (2009)
28. Zhu, X., Uysal, M., Wang, Z., Singhal, S., Merchant, A., Padala, P., Shin, K.: What Does Control Theory Bring to Systems Research? ACM SIGOPS Operating Systems Review 43(1), 62 (2009)