2010 18th IEEE International Requirements Engineering Conference

Requirements-Aware Systems

A research agenda for RE for self-adaptive systems

Pete Sawyer, Nelly Bencomo, Jon Whittle
Computing Department
Lancaster University
Lancaster, UK
e-mail: [sawyer, nelly, whittle]@comp.lancs.ac.uk

Emmanuel Letier, Anthony Finkelstein
Department of Computer Science
University College London
London, UK
e-mail: [e.letier , a.finkelstein]@cs.ucl.ac.uk

Abstract— **Requirements are sensitive to the context in which the system-to-be must operate. Where such context is well-understood and is static or evolves slowly, existing RE techniques can be made to work well. Increasingly, however, development projects are being challenged to build systems to operate in contexts that are volatile over short periods in ways that are imperfectly understood. Such systems need to be able to adapt to new environmental contexts dynamically, but the contextual uncertainty that demands this self-adaptive ability makes it hard to formulate, validate and manage their requirements. Different contexts may demand different requirements trade-offs. Unanticipated contexts may even lead to entirely new requirements. To help counter this uncertainty, we argue that requirements for self-adaptive systems should be run-time entities that can be reasoned over in order to understand the extent to which they are being satisfied and to support adaptation decisions that can take advantage of the systems' self-adaptive machinery. We take our inspiration from the fact that explicit, abstract representations of software architectures used to be considered design-time-only entities but computational reflection showed that architectural concerns could be represented at run-time too, helping systems to dynamically reconfigure themselves according to changing context. We propose to use analogous mechanisms to achieve requirements reflection. In this paper we discuss the ideas that support requirements reflection as a means to articulate some of the outstanding research challenges.**

Keywords- Requirements, reflection, run-time, self-adaptive systems

I. INTRODUCTION

At the heart of orthodox requirements engineering (RE) is the need to understand the problem domain in order to formulate the system-to-be's requirements model, comprising goals, domain assumptions and requirements. Implicit in this is the assumption that the environmental context is reasonably static and can be understood sufficiently well to permit the requirements model for a workable solution to be formulated with confidence. In practice, environmental contexts are seldom static over long periods, and their sheer scale sometimes inhibits understanding. Nevertheless, RE offers a range of techniques capable of mitigating or avoiding these problems provided change happens slowly enough to allow developers to evaluate the implications and take appropriate action.

Increasingly, however, *self-adaptive systems* [3] are being commissioned for problem contexts that are subject to change over short periods and in ways that are poorly understood. In part, this is because the machinery of self-adaptation has improved, providing a means for systems to respond at run-time to changing context. For example, adaptive middleware systems allow software components providing different functionality or quality of service to be substituted at run-time. Such *compositional adaptivity* [40] has made it technically and economically feasible to, for example, engineer networked systems [8] capable of autonomous changes in topology, load, tasks, and physical and logical network characteristics. Thus smart routers [39] are able to optimize their behaviour to prevailing network conditions. Complementing the bottom-up driver for self-adaptive systems provided by improved software technology, is a problem-driven motivation driven by a range of pressing real-world problems such as disaster planning and smart energy management. The common factor in each of these problem domains is the potential for rapidly-changing, hard-to-understand environmental contexts. It seems likely, therefore, that self-adaptivity, once considered the preserve of robotics research, will become an increasingly required system property.

However, for this to become true, it is crucial that it is possible to discover, reason about and manage requirements for systems that, at run-time, will encounter environmental contexts about which significant uncertainty exists at design-time. One key contribution to the achievement of this has been work on requirements monitoring, pioneered by the seminal work of Fickas and Feather [11].

Requirements monitoring is necessary because deviations between the system's run-time behaviour and the requirements model may trigger the need for a system modification [9]. Such deviation needs to be correlated with the state of the environment so that the reasons can be diagnosed and appropriate adaptation performed. This is what Salifu et al. call monitoring and switching in context [56]. Where systems have the need to adapt dynamically in order to maintain satisfaction of their goals, requirements engineering ceases to be a purely static, off-line activity, but

1090-705X/10 $26.00 © 2010 IEEE
DOI 10.1109/RE.2010.21

a run-time one too; an idea first proposed by Berry et al. [57].

Run-time reasoning over requirements is necessary where design-time decisions about the requirements are made on incomplete and uncertain knowledge about the application domain and the stakeholders' goals. There are clear benefits to being able to revise these decisions at run-time when more information can be acquired by observing the system in use. Our argument in this paper, which is an expansion of the initial ideas presented in our short ICSE 2010 paper [53], is that to support run-time RE, requirements for self-adaptive systems need to be run-time entities that can be reasoned over at run-time.

The essence of our idea is that a self-adaptive system should be *requirements-aware*. A requirements-aware system should be able to introspect about its requirements in the same way that reflective middleware-based systems [13] build on Smith's [58] original notion of computational reflection to permit introspection about their architectural configuration [2]. Indeed, since reflective middleware provides an elegant mechanism for the achievement of compositional adaptation, requirements reflection would represent a logical extension of the same principle. Implicit in the ability for a system to introspect on its requirements model is the representation of that model at run-time. Recently, several authors have made important steps in the direction of requirements-aware systems. Sutcliffe et al. [54] with their PC-RE method allows requirements to change over time in the face of contextual uncertainty. Epifani et al. [49] propose a run-time methodology to use a feedback control loop between models of non-functional properties and implementations. At run-time, the executing system provides information as feedback that is used to update a model to increase its correspondence with reality. Analysis of the updated model at run-time makes it possible to detect if a desired property (such as performance and reliability) is violated, causing automatic reconfigurations or recovery actions aimed at guaranteeing the desired goals. Where our ideas differ from the above is our explicit use of computational reflection to serve as the primary means to achieve requirements-awareness. This means that there exists a run-time representation of the requirements models that is causally connected to the executing system.

We identify five key challenges that need to be addressed for requirements to become useful run-time entities and for self-adaptive systems capable of operating resiliently in volatile and poorly-understood environments to become a reality. These challenges represent a research agenda that we believe to be necessary in order for RE to remain relevant for an important and radically different new class of systems.

The rest of the paper is structured as follows. In Section II we elaborate on the argument above to motivate the need for requirements-aware systems, concluding by enumerating five key challenges. Sections III, IV, V, VI and VII provide our rationale for each challenge and pointers to ways in which we believe progress can be made. Section VIII concludes the paper.

II. Background and Motivation

There are two primary drivers for self-adaptive systems; improved capability, which is making self-adaptive systems easier to implement, and the emergence of problems for which self-adaptation offers the most viable solution.

Improved capability means that programmers now have access to programming frameworks and run-time infrastructures that support self-adaptation. These include adaptive architectures [14] such as OpenCom [5] Rainbow [41] and the work of Oreizy et al. [42]. OpenCom, for example, provides a component model and a set of open component libraries. A programmer can define the architecture and compose it using a set of substitutable components from the component libraries. Policy rules define the circumstance under which components can be substituted, while the architecture is constrained to ensure that only valid component configurations are created. Using this model, adaptive applications such as GridStix can be constructed. GridStix [43] is a sensor grid that adapts dynamically as the river it monitors changes state. Hence, for example, the system can switch between components that implement Bluetooth, IEEE 802.11b or GPRS communications technologies according to the demands imposed by battery health, water depth and resilience.

While the machinery of self-adaptation has improved, self-adaptation has emerged as a design strategy to mitigate maintenance costs in systems where factors such as complexity, mission-criticality or remoteness make off-line adaptation impractical. Such systems range from enterprise systems where scale and complexity are the main drivers for self-adaptation, to embedded systems where remoteness and inaccessibility drive the choice of design strategy. Where these drivers are combined with uncertainty about the environmental context, self-adaptation may offer the only feasible solution.

Self-adaptation presupposes that a system is able to sense its environment, detect changes and react accordingly. If the environment is well-enough understood the appropriate action to take can be determined because the relationship between environment and system behaviour can be determined at design-time. Where the environment is poorly understood, however, that relationship cannot be known with certainty and so the decision of how to react is hard to make. To remedy this, it is insufficient to merely sense the environment. It is necessary to monitor to discover when the system behaviour deviates from its requirements model. Monitoring requirements' satisfaction is hard. The requirements themselves may be imprecise "softgoals" or non-functional requirements, and they may not be measurable directly. For example, while individual web-service response times may be measurable, the end-to-end quality of service of a composed web service subject to network latency and uneven user demand may be much harder to determine on-the-fly. The relationship between what can be monitored at run-time and what the monitored data indicates about requirements' satisfaction may be itself to uncertainty. Salifu et al., for example, identify sets of monitorable *context variables* at design-time with the

expectation that the set may need to be revised or extended as greater domain understanding is gained through operation.

Despite these difficulties, significant progress has been made on requirements monitoring [11][19][51][7][21][24]. However, on the closely-related issue of how to take corrective action to reconcile system behaviour with the requirements model when monitoring detects deviancy, research is still in its infancy. Some progress has been made in the domain of web services [49], where infrastructure standards help service discovery and dynamic (re-)binding. But even a well-defined web service infrastructure where service specifications can be queried on-line doesn't help with reasoning over the requirements model for a composed service. For example, switching to a functionally equivalent web service as an alternative to one whose response time has become unacceptable may impact on other requirements such as those relating to cost or security. Such issues can be resolved off-line, particularly if the monitoring data helps resolve some of the environmental uncertainty, but only because the developers have access to the full requirements models over which they can reason and reach informed resolution decisions.

To be able to fully exploit requirements monitoring and realize our vision of a requirements-aware system it will be necessary to hold the requirements models in memory in a form that permits the running system itself to evaluate goal satisfaction in real-time and to propagate the effects of (e.g.) falsified domain assumptions. Introspection on requirements models needs to be coupled with identification of alternative solution strategies. The system therefore needs to be aware of the capabilities of its own adaptive machinery and this awareness needs to be coupled with the requirements models. When the system behaviour deviates from the requirements models and triggers a search for suitable corrective action, the range of available adaptations (perhaps in the form of component substitutions) can be evaluated and their different trade-offs balanced to find the most suitable of the available solutions. Once identified, the adaptation can be enacted and monitored to evaluate its effect on system behaviour.

To make this work, the requirements model must be able to tolerate uncertainty. Of course, the monitoring, reasoning and adaptive capabilities of the system help tolerate uncertainty, but the requirements models also need an explicit representation of where uncertainty exists to know which requirements can be traded off in favour of critical requirements, and under what circumstances. In all of this, there will be conflicts as every adaptive strategy may involve a different set of trade-offs. The adaptive reasoning mechanism needs to be capable of dealing with these conflicts, and of reasoning with imperfect knowledge.

Finally, self-adaptive capabilities do not mean that traditional, off-line adaptive or corrective maintenance will not be necessary. Developers will therefore need to be able to analyse the system's performance and in particular trace triggering events, the consequential adaptations and the reasoning that selected them. Users too, may need access to explanations for system behaviour in order to build trust in them.

There are therefore complementary and inter-linked areas needing research to realize requirements-aware systems:

1. Run-time representations of requirements
2. Evolution of the requirements model and its synchronization with the architecture
3. Dealing with uncertainty
4. Multi-objective decision-making
5. Self-explanation

Research challenges for each of these are developed in more detail over the next five sections.

III. RUN-TIME REPRESENTATIONS OF REQUIREMENTS

Architectural reflection [5][18] offers a pointer to how requirements may become run-time artifacts. Architectural reflection allows introspection of the underlying component-based structures. An architecture meta-model can be used to get the current architecture information to determine the next valid step in the execution of the system. Specifically, the architecture meta-model provides access to the component graph where components are nodes and bindings are arcs. Inspection is achieved by traversing the graph, and adaptation/extension is realized by inserting or removing nodes or arcs. Such extensions and changes will be reflected on the systems during run-time. Crucially, this meta-model supports reasoning about the architecture of the system. We argue that the same principles can be applied to allow introspection and reasoning based on (meta-) models of requirements at run-time. The mechanisms for achieving this are explored in the next section. In this section, we focus on run-time requirements representations in a form suitable for introspection and adaptation. Introspection would offer the ability of a run-time requirements entity to reveal information about itself and hence allow the system to reason about its requirements.

RE is concerned with the identification of the goals to be achieved by the system, the operationalization of such goals as specifications of services and their constrains, and the assignment of responsibilities for services among agents [22] (i.e. human, physical, and software components) forming the system. Goals can be operationalized in many different ways and goal modeling allows us to explore the choices, detect conflicts between requirements and select the preferred choice by the assessment of the effects on the system and its context [16][24]. The selection of an appropriate set of choices is essential to the success of a system. However, inherent uncertainty about the environment and behavior may make it impossible to anticipate all the exceptional circumstances. In contrast to assumptions made during the specification of the system, the conditions of execution may change unexpectedly manifesting unforeseen obstacles [55]. As a result, the selection of the right set of choices, in the case of self-adaptive systems, may need to be delayed until run-time when the system can reason to make choices informed by concrete data sensed from the environment [11].

Dynamic assessments and reasoning about requirements imply a run-time representation of system requirements (i.e. its run-time requirements model [1]) that is rich enough to support the wide range of run-time analyses concerning stakeholders' goals, software functional and non-functional

requirements, alternative choices, domain assumptions, scenarios, risks, obstacles, and conflicts. Such run-time representation will drive the way a system can reason and assess requirements during run-time and crucially will underpin the four challenges described in the following sections. To support such dynamic assessment of requirements, language features found in goal-oriented requirements modeling languages KAOS [22] and i* [26] hold particular promise. KAOS is particularly useful here as it integrates the intentional, structural, functional, and behavioral aspects of a system, and offers formal semantics that would allow automated reasoning over goals.

One way to achieve a run-time representation of requirements would be to base it on goal-based RE, and, particularly, to provide language support for representing, traversing and manipulating instances of a metamodel for goal modeling, for example based on the KAOS meta-model [6]. The meta-model could be provided as a set of built-in constructs to a programming language, or alternatively be provided in the form of (e.g.) a library. Crucial, the meta-model must provide a way to represent and maintain relationships between requirements and agents and the inter-relationships between requirements, to dynamically reassign the goals to different agents or to move to alternative goals in the goal tree. In other words and in contrast to previous work [6], we envision that this representation must take place in such a way that is not only readily understandable by humans but also easily manipulable by the system itself. This will allow the persons responsible for maintaining software to query the software (as opposed to externally stored documentation) to determine requirements-relevant information, such as: What are the sub-goals of a goal? Which agents are responsible for achieving the goal? What assumptions are associated with a goal? In some cases, the software itself would also be able to use this information to guide its own adaptation. The fact that humans would be able to query the requirements model and its relation to the run-time behavior may be more important than just letting the software do so. The benefits of being able to easily maintain and retrieve up-to-date requirements models go beyond self-adaptation.

To reason about requirements relationships we also need to model the inter-requirement relationships. The right associations between such requirements models and the implementation artefacts should also be taken into account to maintain the run-time traceability needed. Therefore, the run-time representation should enable the definition of run-time-traceable dependencies of requirements specifications. Hence, tracing becomes a run-time activity.

IV. Evolution of the Requirements Model and its Synchronization with the Architecture

Requirements reflection enables self-adaptive systems to revise and re-evaluate design-time decisions at run-time when more information can be acquired about these by observing their own behaviour. We therefore see two research issues here. One is the evolution of the requirements models themselves and the maintenance of consistency between the different views during this evolution. In order to do this it is necessary to specify how the system's requirements can evolve dynamically and to specify the abstract adaptation thresholds that allow for uncertainty and unanticipated environmental conditions [52][4]. Unfortunately, to our knowledge none of the existing techniques deal with this degree of evolution, incomplete information, or uncertainty.

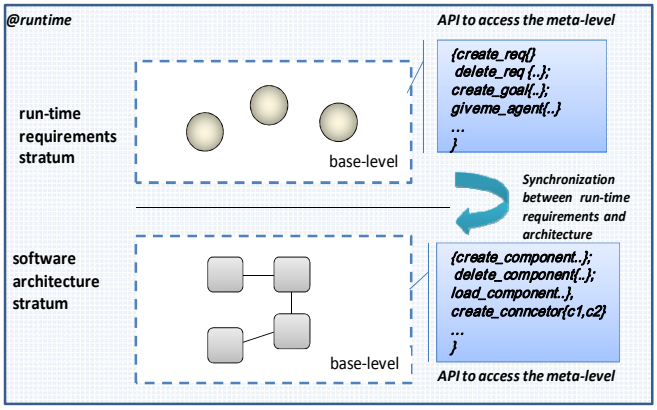

Figure 1. Synchronization between run-time requirements and the architecture

The second research issue is the need to maintain the synchronization of the run-time requirements model and the software architecture as either the requirements are changed from above or the architecture is changed from below. Current work on computational reflection offers a potential way to structure the run-time relationship between the requirements model and the architecture. Traditionally, reflective architectures are organized into two causally-connected layers [5]– the base layer, which consists of the actual running architecture – and the meta-layer, which consists of meta-objects, accessible through a meta-object protocol (MOP), for dynamically manipulating the running architecture. We envision a similar strategy for achieving requirements reflection: a base layer consisting of run-time requirements objects (i.e. the requirements models) and a meta-layer allowing dynamic access and manipulation of requirements objects (i.e. stakeholders' goals, goal refinements, alternative choices, domain assumptions, etc.). This way of structuring requirements reflection therefore leads to two strata – one for requirements and one for architecture – each encompassing a causally-connected base and meta-layer. As in the case of the traditional architecture meta-model (which offers operations over components and connectors), we can define primitives for the goal-based requirements meta-model that allows the meta-level to modify the base-level for the case of the requirements stratum. These primitives might include *add_requirement, delete_requirement, replace_requirement, add_goal, delete_goal, replace_goal, obtain_agent_from_goal, assign_agent_to_goal*. A library of requirements model transformation operators, in the spirit of [44], would then be defined on top of these primitive operations. The rich catalogue of model transformation patterns for goal

refinement, conflict resolution and obstacle resolution associated with the KAOS language [22] may provide the basis for defining this library. It would also be complemented with operators for resolving inconsistencies between multiple views in the spirit of Xlinkit [45] or techniques for automatically fixing inconsistencies in UML models [46]. Figure 1 summarizes the proposed structure.

The structures in Figure 1 would require coordination between the upper requirements stratum and the lower architecture stratum. As a simple example, if a goal is changed in the upper stratum, then the running system may identify a set of components in the architecture to replace. Put more simply, changes in the software architecture should be monitored to ensure that the requirements are not broken; and changes to the requirements at run-time should be reflected in the running system through dynamic generation of changes to the software architecture. For this to be possible there needs to be a tight semantic integration between requirements and architecture models. While there are many similarities between requirements engineering models and architecture description languages, subtle semantic differences between existing languages make the relation between the two models complex [47]. Integration between requirements and architecture is already an urgent area for research. It is particularly important for requirements-aware systems that progress is made.

V. DEALING WITH UNCERTAINTY

A key challenge when developing self-adaptive systems is how to tame uncertainty about the environment. It is important to understand that uncertainty and change are related, but distinct concepts. An environment that changes, but for which the nature of those changes is known, can be handled using standard development techniques such as defining adaptation trigger conditions. In such cases, requirements awareness is not strictly necessary. More interesting, however, are cases where the environment changes in ways that cannot be predicted. In this situation, it is not adequate to define adaptation triggers because the correct triggering conditions cannot be anticipated at design-time. An alternative solution is therefore required: either one that learns new triggering conditions at run-time, or, as proposed in this paper, a higher-level adaptation mechanism in which requirements themselves are represented at run-time, monitored, and traded-off against each other, if necessary, when unexpected contextual changes take place.

Handling uncertainty depends of course on how uncertain a phenomenon is. Although uncertainty can be measured on a variety of different scales, at different levels of granularity, a useful 4-level classification is provided by Courtney [27]:

- Level one: general confidence about the shape of the future, but some key variables do not have precise values. It is possible to make some estimates and to establish a range of possible outcomes.
- Level two: there are a variety of possible future scenarios but it is possible to list them and they are mutually exclusive and exhaustive.
- Level three: it is feasible to construct future scenarios but these are mere possibilities and are unlikely to be exhaustive.
- Level four: it is not even possible to frame possible future scenarios. Any scenario is likely to be just a wish list and may have little bearing on reality.

Although taken from business theory, this classification can be applied to self-adaptive systems. Level one corresponds to the fact that change is known to happen, but that there is little or no uncertainty about the change. Standard trigger conditions could be used to implement solutions in this case. Level two is a little more complex because it requires designers to take into account the full range of possible scenarios. A possible development solution in this case is provided by techniques such as dynamic software product lines that enumerate all alternative environmental conditions and design adaptations accordingly. Level four is beyond the scope of this paper since the uncertainty level is too high to expect any kind of self-adaptation. However, level three is where we feel requirements awareness can be effective.

In the short term, we believe that RE should consider a move away from binary satisfaction conditions for requirements to more nuanced notions of requirements conformance. As an example why this is necessary, consider a self-adaptive system with two overarching requirements: to perform a given task well and to perform it efficiently. A naïve approach to requirements monitoring might attach thresholds as to what constitutes "well" and "efficiently". The problem with this is that a slight change in efficiency or "wellness" will trigger an adaptation when it may in fact not be necessary.

A second consideration is that, for self-adaptive systems, not all requirements have equal standing. If the environment changes unexpectedly, for instance, it may be wise temporarily not to satisfy a non-critical requirement if it means that a critical requirement will continue to be satisfied.

To address such issues, we call for research into how existing requirements languages and methodologies can be extended so that self-adaptive systems have run-time flexibility to temporarily ignore some requirements in favour of others – that is, we envisage run-time trade-offs of requirements being made as the environment changes.

As a first step, we have developed the RELAX requirements language for adaptive systems [25][4][48]. RELAX defines a vocabulary for specifying varying levels of uncertainty in natural language requirements and has a formal semantics defined in terms of fuzzy branching temporal logic. This allows a requirements engineer to specify ideal cases but leaves a self-adaptive system the flexibility to trade-off requirements at run-time as environmental conditions change – i.e., certain requirements can be temporarily RELAX-ed.

As a very simple example, consider a protocol that synchronizes various computing devices in a smart office environment. One requirement for such a system might be:

The synchronization process SHALL be initiated when the device owner enters the room and at 30 minute intervals thereafter.

RELAX provides a process that assists a requirements engineer to make a decision whether a requirement should be RELAX-ed. In this case, s/he might decide that the hard thresholds are not crucial and RELAX the requirement to:

The synchronization process SHALL be initiated AS EARLY AS POSSIBLE AFTER the device enters the room and AS CLOSE AS POSSIBLE TO 30 minute intervals thereafter.

Given a set of RELAX-ed requirements, they can be traded-off at run-time. For example, critical requirements would not be RELAX-ed, whereas less critical ones would be; in this case, the self-adaptive system can autonomously decide to temporarily not fully satisfy such requirements. RELAX provides a set of well-defined operators (e.g., AS EARLY AS POSSIBLE, AFTER above) which can be used to construct flexible requirements in a well-defined way. It also offers a way to model the key properties of the environment that will affect adaptation. Although there is a formal semantics, in terms of fuzzy logic, for RELAX, there is not as yet an implementation that actually monitors RELAX-ed requirements at run-time. This therefore is a clear avenue for immediate research.

Fuzzy logic is not the only formalism that could be used to reason about uncertainty in the environment, of course. Numerous mathematical and logical frameworks exist for reasoning about uncertainty [12]. For example, probabilistic model checkers have been used to specify and analyse properties of probabilistic transition systems [15] and Bayesian networks enable reasoning over probabilistic causal models [10]. However, only limited attention has been shown so far to the treatment of uncertainty in requirements engineering models. Our ongoing work has the objective to develop extensions to goal-oriented requirements modeling languages to support modeling and reasoning about uncertainty in design-time and run-time models.

In the longer term, self-adaptive systems, and RE in particular, needs a theory of uncertainty. Given such a theory, requirements for self-adaptive systems could be related to the uncertainty in the environment and could be monitored or adapted according to that uncertainty. Other fields of study offer possible starting points for such a theory – for example, risk analysis for possible security issues in software-intensive systems [28], risk assessments in engineering disciplines [29], the economics of uncertainty [30], and uncertainty in management theory [27], as well as well-known mathematical models of uncertainty such as Bayesian networks. All of these fields have developed theories for dealing with uncertainty in their respective domains. An interesting longer-term research question is to distill some of this thinking and incorporate it into requirements engineering for self-adaptive systems.

VI. MULTI-OBJECTIVE DECISION-MAKING

Because of the nature of conflicting requirements, run-time resolutions of uncertainty inherently involve multi-objective decision making. In software engineering, multi-objective decision making techniques most often rely on constructing a utility function, defined as the weighted sum of the different objectives. However, this approach suffers from a number of drawbacks. Firstly, it is well known that correctly identifying the weight of each goal is a major difficulty. Secondly, the approach hides conflicts between multiple goals under a single aggregate objective function rather than truly exposing the conflicts and reasoning about them.

We argue, in contrast, that users must be involved in the decision making process in an interactive fashion. Consider a smart energy management system based on mobile energy monitors placed within a household that can sense both energy usage and household activity. The system can adjust heating levels in different rooms according to the presence or absence of inhabitants or, it might override pre-set heating controls to respond to sudden weather changes. The system thus needs to be adaptive to respond to contextual changes and changes in people's behaviour. The system's decision-making must of necessity be multi-objective with comfort of the users, economy and the needs of individuals within a household potentially in tension. At run-time we need to understand the current behaviour of the system and cope with future behaviour. Such an approach provides more flexibility than predefined utility functions as it would allow the relative importance of goals to be discovered and modified at run-time. By engaging users in the decision making process, it would also increase their trust and understanding of the system's adaptive behaviour. The core technical challenge here is to integrate and adapt existing interactive multi-criteria decision approaches to the problem of making run-time decisions about alternatives in goal models. We envisage a mathematical framework that supports decision making about *requirements alternatives*; the parameters used in the decision model should be *measurable* so that they can be related to the data collected during system monitoring; and the *computational complexity* of the decision model should be such that it can be evaluated efficiently at run-time. Such a framework can build on existing outranking and interactive approaches to multi-criteria decision-making [20] [50] as well as on research on evaluating alternatives [17] and dealing with conflicts in goal models [23].

VII. SELF-EXPLANATION

A well-known problem with self-adaptive systems is that users may not understand or trust them. Such a lack of intelligibility can mean that users may cease to use a self-adaptive system [35]. One well-studied approach to addressing this problem is to provide human-readable explanations of adaptive behaviour. The intuition is that if a user can query the system's decisions, s/he is less likely to abandon it and, indeed, may accept the system's choices over

his/her own, which may not be based on a full understanding of the system and the context.

The provision of understandable explanations of system behaviour has been studied in a number of research fields: principally, in knowledge-based systems [34][37], intelligent tutoring systems [31], context-aware systems [36][33], and debugging [38]. Depending on the application, a range of information and a range of interaction styles can be supported in explanations. Lim et al [36] provide a useful classification of explanation types as follows:

- **What:** What did the system do?
- **Why**: Why did the system do it?
- **Why not:** Why did the system not do something else?
- **What if:** What would the system do in a hypothetical scenario?
- **How to:** How can the system be forced to do something, given the current context?

Interestingly, Lim et al report on a controlled experiment comparing the effectiveness of the last four of these explanation types on user understanding and trust of the system. Perhaps surprisingly, although **why** and **why not** explanations show a significant improvement in user trust and understanding, as compared to no explanation, **what if** and **how to** provide no improvement in their study. This may suggest that, when designing self-explanation capabilities, designers should focus resources on **why** and **why not**.

The level of complexity of providing explanations depends crucially on the technology used to implement adaptation. Basic context-aware systems, for example, use rules to identify a situation and change behaviour accordingly. Much of the work in intelligibility of context-aware systems does not consider interactions between rules and, in such cases, providing explanations is fairly straightforward since a rule *Context → Change*, can be handled by keeping track of which rule triggered a change and then presenting, in a user-friendly way, the contextual condition which enabled the rule. Chains of rules can be handled using backward chaining or abduction.

At the other extreme, self-adaptive systems based on machine learning or AI techniques (e.g., Support Vector Machines or Neural Networks) are black boxes that are not inherently interpretable. Although there has been some work on extracting rules from such technical solutions (e.g., [32]), which can then be used to derive explanations, explaining such systems will remain a major challenge for the foreseeable future.

In the context of requirements reflection, as described in this paper, the technique for self-adaptation is based on a run-time goal model and qualitative and quantitative reasoning about how the organisation of the goal model changes over time. The tree-based structure of goal models means that existing techniques for explanations in context-aware systems based on decision trees (e.g., [36]) are immediately applicable. However, there are additional challenges which will have to be addressed, namely:

1. *How to explain the interaction of multiple conflicting goals and their corresponding adaptations?* The majority of existing work has tackled only one adaptation rule at a time and has not considered how to explain, in an understandable way, the interplay between multiple rules, e.g., why one requirement has been traded-off against another.
2. *How to explain the link between requirements and architecture?* Although some end-users may not wish to see explanations in terms of architectural elements, for more advanced users (e.g., developers or extenders of a self-adaptive system), it will be important to understand how a change in the goal model is reflected in a change in the run-time architecture model. Explaining such links requires synchronization between goals and architecture, as discussed in Section III, and requires that this synchronization is adequately conveyed to the user.
3. *How to explain fuzzy behaviour?* Soft computing is likely to be a key component of many self-adaptive systems and, as noted above, the complexity of the explanation mechanism depends heavily on the complexity of the adaptation implementation technique. In our own case, as highlighted in Section IV, one of the techniques we are investigating is fuzzy temporal logic. This clearly presents challenges in terms of how to interpret and present decisions based on fuzzy boundary conditions – e.g., will the user understand why a key adaptation decision was taken based on reaching a "rough" threshold?

An additional research challenge worth exploring is in how best to present explanations, which essentially consist of a trace of system behaviour (in our case, a sequence of operations applied to the run-time requirements and/or architecture models), in an intuitive way, which is easy and quick for users to grasp. The best method of presentation, of course, depends on the type of user, but one potential method from software engineering is the use of scenarios as a presentation technique. Scenarios are well-known in the requirements engineering community as a representation that is easy to understand by a variety of different stakeholder groups, and we therefore posit that they could be effectively used within a self-explaining adaptive system.

VIII. CONCLUSIONS

We have argued that self-adaptive systems should be requirements-aware systems. Our motivation for advocating requirements-awareness is that self-adaptive systems are increasingly being tasked with operating in volatile and poorly-understood environments. At design-time, sufficient uncertainty about the environment exists that requirements engineers can only hypothesize about the states and events that the system may encounter at run-time. Because so much is based on conjecture, the systems need the ability to self-adapt to cope with unforeseen or partly-understood events if it is to be adequately resilient to unanticipated environmental contexts.

A number of advances have been made in RE that support this vision; notably work on requirement monitoring and more recent work that builds upon monitoring to

maintain run-time requirements representations (e.g. [11][19][48][53]). However, our manifesto is framed in terms of requirements reflection; that we need to extend the principles that allow advanced reflective systems and reflective middleware platforms to introspect about their architecture, upwards into the requirements level. This will provide the means to observe how the system's behaviour matches that predicted in its requirements models. This in turn requires that the requirements models cease to be strictly off-line, passive entities and become run-time objects that can be queried and manipulated to (e.g.) re-assign goal satisfaction responsibility between different agents as the needs of the fluctuating environmental context dictate.

Synchronization of the system's architecture and requirements models is implicit in requirements reflection since different architectural configurations often imply different trade-offs, particularly in terms of softgoal satisfaction. Such trade-offs often necessitate the resolution of conflicting goals. Uncertainty and the scale of the possible solution space preclude enumeration and resolution of such conflicts at design-time, so the necessary resolution reasoning needs to occur at run-time. Underpinning these principles is a need to be able to express the uncertainty that exists in terms of what it is that is uncertain and the boundaries of what is acceptable in terms of goal satisficement when unanticipated events occur and conflicting goals need to be traded-off. Finally, a self-adaptive system is likely to exhibit emergent behaviour. Developers need to be able to trace the origin of this behaviour and users need to be gain confidence in the system. These both require an ability for the system to account for its behaviour in some appropriate form.

The machinery for self-adaptation already exists and is increasingly being deployed in systems with limited context-awareness and reconfigurability capabilities. RE has a critical role to play if future self-adaptive systems are to reach the point where they can be developed and deployed with confidence in their ability to survive and deliver their intended service. The research agenda set out in this paper will, we believe, help achieve this.

ACKNOWLEDGEMENT

We are grateful to Gordon Blair and Paul Grace for their enlightening discussions about the ideas presented in this paper, and to Dan Berry for his helpful suggestions.

REFERENCES

[1] Blair, G., Bencomo, N., and France, R. 2009. "Models@ run.time", IEEE Computer, 42(10), pp. 22–27, 2009.

[2] Capra, L., Blair, G., Mascolo, C., Emmerich, W., and Grace, P. "Exploiting reflection in mobile computing middleware", ACM SIGMOBILE Mobile Computing and Communications Review, 6(4), pp. 34–44, 2002.

[3] Cheng, B., Giese, H., Inverardi, P., Magee, J., and de Lemos, R. "Software engineering for self-adaptive systems: A research road map", In Software Engineering for Self-Adaptive Systems. Springer. LNCS, 5525, 2008.

[4] Cheng, B., Sawyer, P., Bencomo, N., and Whittle, J. "A Goal-Based Modeling Approach to Develop Requirements of an Adaptive System with Environmental Uncertainty", Proc. 12th ACM/IEEE International on Model Driven Engineering Languages and Systems (MODELS 2009), pp. 468-483, 2009.

[5] Coulson, G., Blair, G., Grace, P., Joolia, A., Lee, K., Ueyama, J., and Sivaharan, T. "A generic component model for building systems software", ACM Trans. on Comp. Syst, 26 (1), pp. 1-42, 2008.

[6] Dardenne, A., van Lamsweerde, A., and Fickas, S. "Goal-directed requirements acquisition", Sci. Comput. Program., 20(1-2), pp. 3–50, 1993.

[7] Dingwall-Smith, A. Run-Time Monitoring of Goal-Oriented Requirements Specifications, PhD thesis, UCL, UK, 2006.

[8] Dobson, S., Denazis, S., Fernandez, A., Gaiti, D., Gelenbe, E., Massacci, F., Nixon, P., Saffre, F., Schmidt, N., and Zambonelli, F. "A survey of autonomic communications", ACM Trans. on Auton. and Adapt. Systems, 1(2), pp. 223–259, 2006.

[9] Feather, M., Fickas, S., van Lamsweerde, A., and Ponsard, C. "Reconciling system requirements and run-time behavior", Proc. 9th international Workshop on Software Specification and Design, IEEE Computer Society, Washington, DC, 1998.

[10] Fenton, N., and Neil, M. "Making decisions: using bayesian nets and mcda", Knowl.-Based Syst., 14(7), pp. 307–325, 2001.

[11] Fickas, S. and Feather, M. "Requirements monitoring in dynamic environments", Proc. 2nd IEEE International Symposium on Requirements Engineering (RE'95), 1995.

[12] Halpern, J. Reasoning about Uncertainty. The MIT Press, October 2003.

[13] Kon, F., Costa, F., Blair, G., and Campbell, R. "The case for reflective middleware", Comm. of the ACM, 45(6), pp. 33–38, 2002.

[14] Kramer, J and Magee, J. "Self-managed systems: an architectural challenge", Proc. Future of Software Engineering (FOSE '07), IEEE Computer Society, pp. 259-268, 2007.

[15] Kwiatkowska, M., Norman, G., and Parker, D. "Probabilistic symbolic model checking with prism: A hybrid approach", International Journal on Software Tools for Technology Transfer (STTT), 6(2), pp. 128-142, 2002.

[16] Lapouchnian, A., Liaskos, S., Mylopoulos, J., and Yu, Y. "Towards requirements-driven autonomic systems design", Proc. 2005 Workshop on Design and Evolution of Autonomic Application Software (DEAS'05), 2005.

[17] Letier, E. and van Lamsweerde, A. "Reasoning about partial goal satisfaction for requirements and design engineering", SIGSOFT Softw. Eng. Notes, 29(6), Nov. 2004, pp. 53-62, 2004.

[18] Maes, P. Computional reflection. PhD thesis, Vrije Universiteit, 1987.

[19] Robinson, W. "A requirements monitoring framework for enterprise systems", Requir. Eng., 11(1), pp. 17–41,, 2006.

[20] Roy, B. Multicriteria Methodology for Decision Aiding. Kluwer Academic, Dordrecht, 1996.

[21] Spanoudakis, G. and Mahbub, K. "Requirements monitoring for service-based systems: Towards a framework based on event calculus", Proc. 19th International Conference on Automated Software Engineering (ASE'04), pp. 379–384, 2004.

[22] van Lamsweerde, A. Requirements Engineering: From System Goals to UML Models to Software Specifications. John Wiley & Sons, 2009.

[23] van Lamsweerde, A., Darimont, R., and Letier, E. "Managing conflicts in goal-driven requirements engineering", IEEE Trans. Sof. Eng., 24(11), pp. 908–926, 1998.

[24] Wang, Y., McIlraith, S., Yu, Y., and Mylopoulos, J. "Monitoring and diagnosing software requirements', Autom. Softw. Eng., 16(1), pp. 3–35, 2009.

[25] Whittle, J., Sawyer, P., Bencomo, N., Cheng, B., and Bruel, J-M. "Relax: Incorporating uncertainty into the specification of self-adaptive systems", Proc. 17th IEEE International Conference on Requirements Engineering (RE'09), Atlanta, Georgia, August 2009, pp. 79-88, 2009.

[26] Yu, E. “Towards modeling and reasoning support for early-phase requirements engineering” 2007. Proc. 3rd IEEE international Symposium on Requirements Engineering (RE’07), pp. 05–08, 2007.

[27] Courtney, H. 20/20 Foresight: Crafting Strategy in an Uncertain World, Harvard Business School Press, 2001.

[28] Peltier, T., Information Security Risk Analysis, Auerbach Publications, 2010.

[29] Stewart, M. and Melchers, R. Probabilistic Risk Assessment of Engineering Systems, Springer, 1997.

[30] Gollier, C. The Economics of Risk and Time, MIT Press, 2001.

[31] Anderson, J., Corbett, A., Koedinger, K., and Pelletier, R. “Cognitive Tutors: Lessons Learned, Journal of the Learning Sciences”, 4(2), pp. 167-207, 2005.

[32] Andrews, R., Diederich, J., and Tickle, A. “A Survey and Critique of Techniques for Extracting Rules from Trained Artificial Neural Networks”, Knowledge-Based Systems, 8, pp. 373-389, 1995.

[33] Bellotti, V., and Edwards, W. “Intelligibility and Accountability: Human Considerations in Context-Aware Systems”, Human-Computer Interaction, 16(2-4), pp. 193-212, 2001.

[34] Gregor, S. and Benbasat, I. “Explanations from Intelligent Systems: Theoretical Foundations and Implications for Practice”, MIS Quarterly, 23(4), pp. 497-530, 1999.

[35] Muir, B., “Trust in Automation: Part I, Theoretical Issues in the Study of Trust and Human Intervention in Automated Systems”, Ergonomics, 37(11), pp.1905-1922, 1994.

[36] Lim, B., Dey, A., and Avrahami, D. “ Why and Why Not Explanations Improve the Intelligibility of Context-Aware Intelligent Systems”, Proc. 27th international Conference on Human Factors in Computing Systems (CHI '09), 2009.

[37] Whittle, J., van Baalen, J., Schumann, J., Robinson, P., Pressburger, T., Penix, J., Oh, P., Lowry, M., and Brat, G. “Amphion/NAV: Deductive synthesis of state estimation software”, Proc. 16th IEEE International Conference on Automated Software Engineering (ASE 2001), pp. 395–399, 2001.

[38] Subrahmaniyan, N., Kissinger, C., Rector, K., Inman, D., Kaplan, J., Beckwith, L., Burnett, M., “Explaining Debugging Strategies to End-User Programmers”, Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC’07), pp. 127-136, 2007.

[39] Schantz, R., Loyall, J., Rodrigues, C., Schmidt, D., Krishnamurthy, Y., and Pyarali, I. ”Flexible and adaptive QoS control for distributed real-time and embedded middleware”, Proc. ACM/IFIP/USENIX 2003 international Conference on Middleware, pp. 374-393, 2003.

[40] McKinley P., Sadjadi, S., Kasten, E., and Cheng, B. “Composing Adaptive Software”, IEEE Computer, 37 (7), pp. 56-64, 2004.

[41] Garlan, D., Cheng, S-W., Huang, A-C., Schmerl, B., and Steenkiste, P. “Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure”, IEEE Computer, 37(10), pp. 46-54, 2004.

[42] Oreizy, P., Medvidovic, N., and Taylor, N. “Architecture-Based Runtime Software Evolution”, Proc. 20th international Conference on Software Engineering (ICSE’98), pp. 177-186, 1998.

[43] Hughes, D., Greenwood, P., Coulson, G., and Blair, G. “GridStix: Supporting Flood Prediction using Embedded Hardware and Next Generation Grid Middleware“, Proc. 2006 international Symposium on on World of Wireless, Mobile and Multimedia Networks, pp. 621-626, 2006.

[44] Johnson, W. and Feather, M. “Building an evolution transformation library”, Proc. 12th international Conference on Software Engineering (ICSE’90), pp. 238-248, 1990.

[45] Nentwich, C., Emmerich, W., and Finkelstein, A. ”Consistency management with repair actions“, Proc 25th international Conference on Software Engineering (ICSE’03), pp. 455-464, 2003.

[46] Egyed, A. ”Fixing Inconsistencies in UML Design Models“, Proc. 29th international Conference on Software Engineering (ICSE’07), pp. 292-301, 2007.

[47] Letier, E., Kramer, J., Magee, J., and Uchitel, S. ”Deriving event-based transition systems from goal-oriented requirements models“, Automated Software Eng. 15(2), pp. 175-206, 2008.

[48] Whittle, J., Sawyer, P., Bencomo, N., Cheng, B., Bruel, J-M. “RELAX: A Language to Address Uncertainty in Self-Adaptive Systems Requirements”, Requir. Eng., 15(2), 2010, pp. 177-196, 2010.

[49] Epifani, I., Ghezzi, C., Mirandola, R., and Tamburrelli, G. “Model evolution by run-time parameter adaptation“, Proc. 31st IEEE International Conference on Software Engineering (ICSE’09), 2009.

[50] Vincke, P. Multicriteria Decision-Aid. J. Wiley, New York, 1992.

[51] Baresi, L., Ghezzi, C., and Guinea, S. ”Smart monitors for composed services“, Proc. 2nd international Conference on Service Oriented Computing (ICSOC '04), pp. 193-202, 2004.

[52] Cheng, B.. and Atlee, J. ”Research Directions in Requirements Engineering“, In 2007 Future of Software Engineering, International Conference on Software Engineering (ICSE’07), pp. 285-303, 2007.

[53] Bencomo, N., Whitlle, J., Sawyer, P., Finkelstein, A., Letier, E. “Requirements Reflection: Requirements as Runtime Entities”, Proc. 32nd CM/IEEE International Conference on Software Engineering – vol. 2 (ICSE’10), pp. 199-202, 2010.

[54] Sutcliffe, A., Fickas, S., and Sohlberg, M. “PC-RE: a method for personal and contextual requirements engineering with some experience”, Requir. Eng. 11(3) (Jun. 2006), pp.157-173, 2006.

[55] van Lamsweerde, A. and Letier, E. “Handling Obstacles in Goal-Oriented Requirements Engineering”, IEEE Trans. Softw. Eng. 26(10) (Oct. 2000), pp. 978-1005, 2000.

[56] Salifu, M., Yu, Y., Nuseibeh, B. “Specifying Monitoring and Switching Problems in Context”, Proc. 15th IEEE International Conference on Requirements Engineering (RE’07), pp. 211-220, 2007.

[57] Berry, D., Cheng, B., and Zhang, J. “The four levels of requirements engineering for and in dynamic adaptive systems”, Proc. 11th International Workshop on Requirements Engineering Foundation for Software Quality (REFSQ’05), 2005.

[58] Smith, B. Reflection and Semantics in a Procedural Programming Language, Ph.D. Thesis, MIT Laboratory for Computer Science Report MIT-TR-272, Cambridge, MA, USA, 1982.