Requirements Reflection: Requirements as Runtime Entities [*]

Nelly Bencomo,Jon Whittle,Pete Sawyer
Computing Department
Lancaster University, UK
{nelly,whittle,sawyer}@comp.lancs.ac.uk

Anthony Finkelstein,Emmanuel Letier
Department of Computer Science
University College London, UK
{a.finkelstein,e.letier}@cs.ucl.ac.uk

ABSTRACT

Computational reflection is a well-established technique that gives a program the ability to dynamically observe and possibly modify its behaviour. To date, however, reflection is mainly applied either to the software architecture or its implementation. We know of no approach that fully supports *requirements reflection*- that is, making requirements available as runtime objects. Although there is a body of literature on requirements monitoring, such work typically generates runtime artefacts from requirements and so the requirements themselves are not directly accessible at runtime. In this paper, we define requirements reflection and a set of research challenges. Requirements reflection is important because software systems of the future will be self-managing and will need to adapt continuously to changing environmental conditions. We argue requirements reflection can support such self-adaptive systems by making requirements first-class runtime entities, thus endowing software systems with the ability to reason about, understand, explain and modify requirements at runtime.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements Specifications—*Languages, Methodologies*

General Terms

Design

Keywords

Requirements, reflection, runtime, self-adaptive systems

1. MOTIVATION

The development of software-intensive systems is driven by their requirements. Traditional requirements engineering (RE) methods focus on resolving ambiguities in requirements and advocate specifying requirements in sufficient detail so that the implementation can be checked against them for conformance. In an ideal world, this way of thinking can be very effective. Requirements can be specified clearly, updated as necessary, and evolutions of the software design can be made with the requirements in mind.

Increasingly, however, it is not sufficient to fix requirements statically because they will change at runtime as the operating environment changes. Furthermore, as software systems become more pervasive, there is growing uncertainty about the environment and so requirements changes cannot be predicted at design-time [25, 4]. It is considerations such as these that have led to the development of self-adaptive systems (SASs) [3], which have the ability to dynamically and autonomously reconfigure their behavior to respond to changing external conditions.

Consider a scenario involving a robot vacuum cleaner for domestic apartments. The vacuum cleaner has goals *clean apartment*, *avoid tripping hazard* and *minimize energy costs.* Further, it has the domain assumption *energy is cheapest at night.* To satisfy the *avoid tripping hazard* goal, a requirement is derived that it should stop operating as soon as any human activity is detected. Night operation satisfies the *minimize energy costs* goal. Thus all the goals are satisfiable, with no trade-offs necessary. In operation, the sound of the cleaner awakens a light sleeper. In the dark, they trip over the now stopped and therefore silent cleaner. Key features of this scenario are:

- Environmental uncertainty, in terms of resident behaviour, led to the accident;
- Disturbance of light sleepers is an emergent property;
- Tripping hazard avoidance and energy efficiency might not be so compatible after all. A switch to a day-time cleaning strategy would imply trading the energy efficiency goal off against tripping hazard avoidance;
- The night-time tripping hazard could be mitigated by deriving a new requirement that on detecting movement the cleaner stops moving but keeps the vacuum motor operating so that the sound alerts residents to its presence.

With the current state-of-the-art, the vacuum cleaner could adapt its behaviour. However any adaptation would either have to be pre-defined at design time, or would have to be a reflexive response to some monitored parameter(s), perhaps using machine-learning algorithm. An effective pre-defined response would be dependent on the requirements analyst anticipating and enumerating all environmental states and the corresponding behaviour required of the vacuum cleaner. In the case of a reflexive response, the relationship between the adaptation and the goals would be at best implicit, mak-

[*]Work partially funded by the EUFP7 STREP DiVA and EPSRC MaTREx projects.

ing verification of goal satisfaction hard or impossible.

The key argument of this paper is that current software engineering (SE) methods do not support well the kind of dynamic appraisal of requirements needed by an SAS and illustrated by the scenario. In most software, information about the definition and structure of requirements is lost as requirements are refined into an implementation. Even in cases where requirements monitoring is explicitly included, high-level system requirements must be manually refined into low-level runtime artefacts during the design process so that they can be monitored. The vacuum cleaner's requirements, for example, could be monitored by instrumenting the code to sense characteristics of the environment, such as energy consumed and collision events.

Usually, however, the link between low-level sensors and the high-level goals that motivated them is not explicitly recorded in the running system. This makes it very difficult to re-assess or revise requirements at runtime when the environment changes. The only information available to the system is low-level information and so reasoning can only be carried out at this level.

This paper proposes a new paradigm for SE, called requirements reflection, in which requirements are reified as runtime entities. This would allow systems to dynamically reason about themselves at the level of the requirements - in much the same way that architectural reflection [5, 2] currently allows runtime reasoning at the level of the software architecture. We believe that requirements reflection will support the development and management of SASs because it will raise the level of discourse at which a software system is able to reflect upon itself.

2. STATE-OF-THE-ART

SASs are challenging to develop because they operate under uncertain conditions. Researchers in many fields are responding to this challenge [3]. The networking community, for example, has developed networked systems capable of autonomous changes in topology, load, tasks, and physical and logical network characteristics [7]. The intelligent agent, machine learning and planning communities have also had a long interest in autonomous and adaptive systems.

SE advances for SASs have resulted in novel software architectures and programming paradigms [3]. Novel software architectures for SASs include mechanisms for swapping out components and/or connectors at runtime (c.f., [14]). Similar techniques have been investigated in the middleware community - in particular, reflection has been used to construct middleware platforms [18, 13] that allow systems to introspect about their structure at runtime, so informing their automatic reconfiguration. Well-established work on reflective middleware [2, 5] uses architecture-based models of component compositions to enable reconfiguration. Reflective architectures are now well established but only support reflection over architecture not requirements.

Important early research in requirements monitoring and diagnosis have laid the foundations for requirements reflection. Fickas and Feather [10] provide a framework for systems to monitor their executions and modify themselves at runtime to better satisfy stakeholders' goals. Similar approaches are proposed in [8, 16, 11]. Several frameworks have been developed for the generation of software monitors from requirements models [8, 21, 19, 6, 24]. Specifically, [24] goes a step further allowing not just monitoring but also diagnosis of software requirements. However, it is our understanding the solution do not offer explicit representation of requirements and only monitors system failures and needs extensions to handle failures [24].

In the research reported above, information about the definition and structure of requirements is lost as requirements are refined into implementations. Current approaches do not reify requirements as runtime objects. Rather, the requirements are used to generate other runtime artefacts, such as requirements monitors. There is therefore only a loose connection between the requirements and the resulting code. This drawback leads to fundamental limitations in how the system can converse about requirements since the system does not have access to the original requirements, but only derived artefacts that may be incomplete or at inappropriate levels of abstraction.

3. PROPOSED APPROACH

In this section, we outline preliminary ideas on how to achieve the requirements reflection vision. We present three key challenges that we see are necessary to realize requirements reflection.

Challenge 1: Runtime representation of requirements. The first challenge is the runtime representation of requirements in a form suitable for introspection and adaptation. Introspection implies the ability of a runtime entity to reveal information about itself. Here, adaptation refers to the ability of a program to modify entities discovered through introspection.

RE is concerned with the identification of the goals to be achieved by the system, the refinement of goals into specifications of services, and the assignment of responsibilities for services among human, physical, and software components forming the system [22]. Goals can be refined and assigned in many different ways and a significant part of the RE process consists of exploring the alternatives and selecting the most preferable option by evaluating the impacts on the system and its organizational context. Selecting among these alternatives is critical to the success of a system, but, in a SAS, an optimal selection is notoriously hard - and perhaps impossible - to achieve before runtime due to inherent uncertainties in the environment. Runtime re-assessment of these choices is therefore crucial as a way to optimize (or satisfice) the system goals in the current context during execution.

Requirements reflection depends on a runtime representation of system requirements (i.e. its runtime model [1]) that is rich enough to support the wide range of runtime analyses outlined above concerning stakeholders' goals, software functional and non-functional requirements, alternative choices, domain assumptions, scenarios, risks, threats, and conflicts. Such runtime representation will underpin the way a system can reason and assess requirements during runtime. To support such dynamic assessment of requirements, language features found in goal-oriented requirements modeling languages such as KAOS [22] and i* [26] hold particular promise. KAOS, for example, integrates the intentional, structural, functional, and behavioral aspects of a system, and has formal semantics permitting automated reasoning over goals.

One way to achieve a runtime representation of requirements, therefore, is to base it on goal-based RE and, in particular, to provide language support for representing, navigating and manipulating instances of a meta-model for goal modeling (e.g., the KAOS meta-model [22]). The meta-model could be provided as a set of built-in constructs to a programming language, but need not be, and, could alternatively be provided in the form of (e.g.) a library. The key point is that the meta-model provides a way to represent and maintain relationships between requirements and code that implements them. This representation must be not only readily understandable by humans but also easily manipulable by the system itself. This will allow programs to query themselves 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?

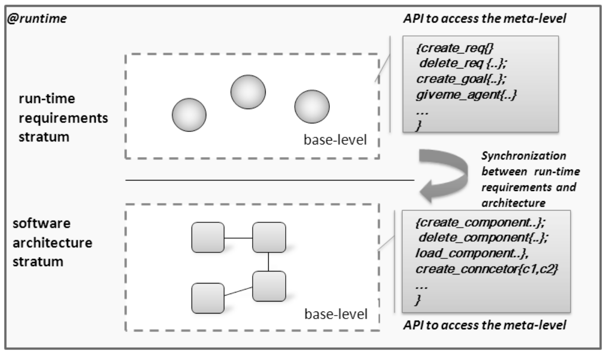

Figure 1: Goal and architecture synchronization.

Challenge 2: Synchronization between goals and architecture. An important purpose of requirements reflection is to enable self-adaptive systems to reason over and re-evaluate their requirements at runtime. Any re-assessments to the requirements must, of course, be reflected in the running system and the crucial link to enable this to happen is to synchronize the runtime representation of the requirements and the software architecture. We therefore see a major challenge of requirements reflection to maintain this synchronization as either the requirements are changed from above or the architecture is changed from below.

Existing work on reflection offers a potential way to structure the runtime relationship between requirements and architecture. As an example, reflective middleware infrastructure, as developed at Lancaster University [5], is organized into two causally-connected layers - the base -layer, which consists of the 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 propose an analogous strategy for realizing requirements reflection: a base- layer consisting of runtime requirements objects and a meta-layer allowing dynamic manipulation of requirements objects (including 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 comprising a causally-connected base and meta-layer. Inspired on 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 - e.g., add_req, delete_req, replace_req, add_goal, delete_goal, replace_goal, obtain_agent_from_goal, assign_agent_to_goal (Figure 1).

The overarching research challenge of the proposed structure in Figure 1 is to coordinate the upper requirements stratum and the lower architecture stratum. That is, there needs to be a tight semantic integration between the strata so that changes in the requirements are seamlessly effected in the architecture (and vice versa). 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; changes to the requirements at runtime should be reflected in the running system by dynamic generation of changes to the software architecture.

Challenge 3: Dealing with Uncertainty. Representing requirements as runtime entities and synching these with architectural meta-data provide the fundamental building blocks to support dynamic re-assessment of requirements, but a key additional challenge, and one which requirements reflection is intended to help with, is to deal with the inherent uncertainties of self-adaptive systems. Uncertainties arise because of the stochastic nature of events in the environment, limited sensor capabilities, and difficulties in predicting how the modification of system services will affect agents' behaviors and the system goals. For instance, the introduction of new capabilities into the system may produce unintended effects. For example, introducing an automatic light off switch in a house may cause residents to unconsciously use more energy in other part of the house because they feel they are already saving on lighting.

Requirements reflection, therefore, includes a consideration of how to reason about uncertainty at runtime and how to reflect this reasoning by manipulating the requirements and architecture strata. 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 [9]. However, only limited attention has been shown so far to the treatment of uncertainty in RE 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 runtime models. Firstly, we have developed the RELAX language [25], which defines a vocabulary for specifying varying levels of uncertainty in natural language requirements and whose semantics is defined formally in terms of fuzzy branching temporal logic. Secondly, we have developed a quantitative goal modelling framework that extends KAOS goal models with a probabilistic layer for the precise specification of quality concerns expressed in terms of application-specific measures [17].

Because of the nature of conflicting requirements, runtime resolutions of uncertainty inherently involve multi-objective decision making. In SE, 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. Considering again the scenario from Section 1, the decision-making is necessarily multi-objective with comfort of the users, economy and the needs of individuals within a household potentially in tension. At runtime 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 runtime . 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 runtime 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 runtime. Such a framework can build on existing outranking and interactive approaches to multi-criteria decision making [20], as well as on our previous research on evaluating alternatives [17] and dealing with conflicts in goal models [23].

4. CONCLUSIONS

We have argued that a SAS may need to address requirements we are only aware of once the system is running. We propose a change of perspective for developing and maintaining requirements for SASs. New approaches should follow other non-traditional principles such as:

- The key role of explicit runtime representation of systems' requirements and goals as an appropriate formalism for endowing systems with self-awareness capabilities;
- The subsequent need to maintain the relationship at runtime between goals and underlying system structures;
- A recognition that uncertainty is intrinsic to SASs and therefore must be managed at all stages of the life-cycle including runtime.

We believe that uncertainty can only be handled effectively if a SAS's requirements can be reasoned over and (e.g.) re-prioritized at runtime. This is what mandates the availability of requirements as runtime objects along with their interrelationships and dependencies, and their relationships with the architecture of the SAS with the monitorable phenomena of the environment. We also envision that requirements reflection will allow higher-level kinds of adaptation more related to system evolution than the standard adaptation provided by standard control systems. We have pointed out challenges associated with the above principles. To address these challenges contributions from colleagues in different fields including (e.g.) architectural reflection and autonomic computing, as well as from RE, will be needed.

5. REFERENCES

[1] N. Bencomo, G. Blair, and R. France. Guest editor's introduction: Models@run.time. *IEEE Software*, 2009.

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

[3] B. H. Cheng, H. Giese, P. Inverardi, J. Magee, and R. de Lemos. Software engineering for self-adaptive systems: A research road map, dagstuhl-seminar on software engineering for self-adaptive systems. 2008.

[4] B. H. Cheng, P. Sawyer, N. Bencomo, and J. Whittle. Goal-based modeling approach to develop requirements for adaptive systems with environmental uncertainty. In *MODELS Conf.*, 2009.

[5] G. Coulson, G. Blair, P. Grace, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan. A generic component model for building systems software. *ACM Transactions on Computer Systems*, February 2008.

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

[7] S. Dobson, S. Denazis, A. Fernandez, D. Gaiti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, and F. Zambonelli. A survey of autonomic communications. *ACM Trans. on Auton. and Adapt. Systems*, 2, 2006.

[8] M. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard. Reconciling system requirements and runtime behavior. *Workshop Software Specification and Design*, 1998.

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

[10] S. Fickas and M. Feather. Requirements monitoring in dynamic environments. In *RE Conf.*, 1995.

[11] H. J. Goldsby, P. Sawyer, N. Bencomo, D. Hughes, and B. H. Cheng. Goal-based modeling of dynamically adaptive system requirements. In *ECBS Conf.*, 2008.

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

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

[14] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *FOSE '07: 2007 Future of Software Engineering*, pages 259–268. IEEE Computer Society, 2007.

[15] M. Z. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with prism: A hybrid approach. In *TACAS*, pages 52–66, 2002.

[16] A. Lapouchnian, S. Liaskos, J. Mylopoulos, and Y. Yu. Towards requirements-driven autonomic systems design. In *DEAS Workshop*, 2005.

[17] E. Letier and A. van Lamsweerde. Reasoning about partial goal satisfaction for requirements and design engineering. In *Symposium on Foundations of software engineering*, pages 53–62, NY, USA, 2004. ACM.

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

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

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

[21] G. Spanoudakis and K. Mahbub. Requirements monitoring for service-based systems: Towards a framework based on event calculus. In *ASE*, pages 379–384, 2004.

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

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

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

[25] J. Whittle, P. Sawyer, N. Bencomo, B. H. Cheng, and J.-M. Bruel. Relax: Incorporating uncertainty into the specification of self-adaptive systems". In *RE Conf.*, 2009.

[26] E. Yu. Towards modeling and reasoning support for early-phase requirements engineering. In *RE Conf.*, USA, 1997.