Reflecting on Self-Adaptive Software Systems

Jesper Andersson[1], Rogerio de Lemos[2], Sam Malek[3], Danny Weyns[4]

[1] Dept. of Computer Science, Växjö University, jesper.andersson@vxu.se
[2] Dept. of Informatics Engineering, University of Coimbra, rdelemos@dei.uc.pt
[3] Dept. of Computer Science, George Mason University, smalek@gmu.edu
[4] Dept. Computerwetenschappen, Katholieke Universiteit Leuven, danny.weyns@cs.kuleuven.be

Abstract

Self-adaptability has been proposed as an effective approach to automate the complexity associated with the management of modern-day software systems. While over the past decade we have witnessed significant progress in the manner in which such systems are designed, constructed, and deployed, there is still a lack of consensus among the engineers on some of the fundamental underlying concepts. In this paper, we attempt to alleviate this issue by exploring the crucial role of computational reflection in the context of self-adaptive software systems. We show that computational reflection forms the foundation of a self-adaptive system, and an understanding of its properties is a prerequisite to intelligent and predictable construction of such systems. Examining several systems in light of computational reflection has helped us to identify a number of key challenges, which we report on and propose as avenues of future research.

1. Introduction

As software engineers have developed new technologies for managing the ever-increasing complexity of constructing modern-day software systems, it has become apparent that there is an equally pressing need for mechanisms that automate and simplify the adaptation and modification of software systems after they are deployed. This has called for the development of software systems with *self-** (self-configuring, self-healing, self-optimization, etc.) capabilities. However, the development of such systems has been shown to be significantly more challenging than traditional, relatively more static and predictable, software systems.

To that end, over the past decade, researchers and practitioners have developed a variety of methodologies and frameworks that are intended to serve as guidelines for the design and development of software systems with such characteristics [1][9][10][14], which are named in this paper as self-adaptive software systems. While these approaches have been successful at alleviating some of the associated challenges of constructing self-adaptive software systems, numerous other challenges persist. We have argued that the lack of consensus among researchers on primitive and fundamental concepts to be one of the main culprits in hindering further progress in this area, and for that we have developed a preliminary classification of self-adaptive software systems in terms of their intrinsic properties [2].

To better understand the underlying principles of self-adaptive software systems, in this paper we study such systems through the prism of computational reflection, i.e., executable logic dealing with the system itself [11]. Computational reflection is an established and well-understood concept in the programming-in-the-small. It has traditionally been studied at the level of programming languages and realized using compiler technologies. We take the position that the principles of computational reflection are also applicable in the programming-in-the-large, which represents the complex self-adaptive software systems we are interested in our study.

The theory of computational reflection provides our study with firm grounding. Maes argues that reasoning about control, self-optimization, and learning are examples of reflective computation [11]. Cazzola et al. adopt the principles of computational reflection and apply them at the software architecture level [5]. The authors argue that architectural reflection provides a conceptually clean model for designing self-adaptive systems. Tanter et al. state that reflection is a powerful approach for adapting the behavior of running applications [16]. Coulson et al. argue for a reflective middleware to support run-time reconfiguration of a component-based target system [6].

Reflection is commonly used in the construction of self-adaptive software systems,[1] albeit often

[1] For the exposition purpose we use the term *reflection* as a shorthand way of referring to *computational reflection*. We acknowledge that the general concept of reflection has been used in a wide variety of contexts other than computation.

978-1-4244-3724-5/09/$25.00

unconsciously disguised under an assortment of other terminologies. A self-adaptive software system is essentially one that changes its behavior by reflecting on itself. In this paper, we show that computational reflection is a necessary criterion for any self-adaptive software system. Clearly, understanding the role of reflection in such systems is an important stepping stone to unraveling the challenges associated with the construction of such systems.

A thorough study of the literature has helped us to identify a list of key reflection properties that can be applied to self-adaptive systems. In turn, these properties have helped us to discern the key, in some cases hidden, characteristics of systems developed previously in our respective research groups. This exercise has illuminated the existence of a rich spectrum of reflection techniques at the engineer's disposal. We believe a thorough understanding of the implications of selecting one technique over another is a prerequisite to intelligent and predictable construction of self-adaptive systems.

Finally, our study has revealed several key challenges with applying the principles of computational reflection in the *programming-in-the-large*. These challenges represent the essence of the difficulty associated with the construction of self-adaptive software systems. They serve as a guideline for the future research in this area.

The rest of the paper is organized as follows. Section 2 introduces a reference model for reflection. Section 3 defines the "reflection prism", where properties described by Maes, Cazzola and Coulson are reformulated and exemplified in the context of self-adaptive systems. Section 4 applies the notion of reflective prism to several systems from our respective research groups, dealing with different application domains, and compares the findings. Based on these findings, we enumerate several challenges in Section 5. The paper concludes with an outline of future work.

2. Reflection Reference Model

In this section we introduce the basic concepts of computational reflection. Reflection is about meta-computation, i.e., computation about computation. Reflection was proposed by Smith as a programming paradigm [15]. He describes self-knowledge as pivotal in computational reflection. In her seminal paper [11], Maes extends Smith's work and provides a comprehensive definition of computational reflection as "*the behavior exhibited by a reflective system*".

Figure 1 shows a reference model for reflection that we have developed and rely on for explaining the various properties of reflection. Figure 1a shows a traditional computational system consisting of two entities: *system* and *domain*. A system typically can be

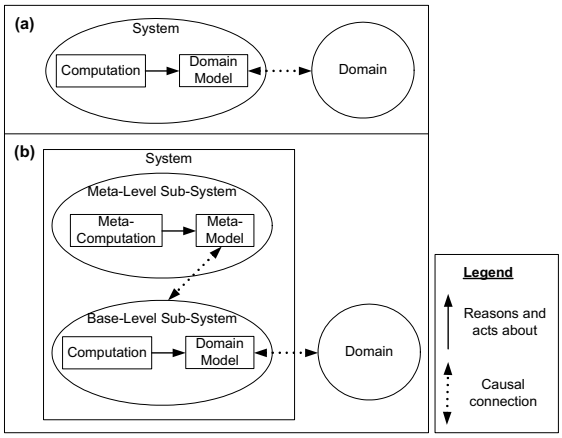

Figure 1. (a) Traditional System, (b) Reflective System.

broken down further to two parts: *computation* and *domain model*. The computation part corresponds to the application logic responsible for providing the system's functionality. The domain model is a representation of the external world (e.g., business problem) that drives the functionality. In other words, the domain model is used by the computation part to reason about and act upon changes in the environment.

An important property of any computational system is the causal connection between the domain model and domain (depicted in Figure 1a). This relationship means that if the domain changes, the domain model changes as well, and vice versa. Moreover, we observe that through the domain model, the computation part and the domain are also causally related. The degree to which this causality is achieved (enforced) depends on many factors. This is an issue that will be revisited again in this paper.

So far, we have described a non-reflective system. Reflection is, as stated before, computations about computations, i.e., a computational system performs computation about its own computation. If a computational system is expected to reason about and act upon itself, the system must reify a representation of itself (self-representation). In Figure 1b we show the reference model of a typical computational reflective system. The system consists of two parts: *meta-level* and *base-level*. The base-level provides the system's functionality. Very similar to a traditional software system, the base-level also contains a computation part and domain model. The meta-level provides the reflective capability and consists of two parts: *meta-computation* and *meta-model*. Meta-model is a reification of the base-level. In other words, meta-model is the self-representation of the system. Meta-computation is the logic dealing with the changes in the meta-model. In this paradigm, meta-model is causally connected to the base-level system, where changes in one are reflected in the other.

The meta-computation may provide two types of activities: *introspection* and *intercession*. Introspection is when meta-computation inspects and reasons about the self-representation. Intercession is when meta-computation acts upon and modifies the system's self-representation (meta-model), which in turn through the causal connection may result in the propagation of change to the base-level.

3. Reflection Prism

To develop a better understanding of the important role of reflection in self-adaptive software systems, we performed a thorough study of the literature. Our objective has been to identify (1) the crucial properties of reflection, and (2) the variation points for each property. In this section, we describe these properties and delineate their importance using examples drawn from well-known frameworks for self-adaptive software systems.

In particular, we have heavily relied on three previous works among others: the work by Maes [11] that constitute a de facto reference point in the field of computational reflection, and the work by Cazzola [5] and Coulson [6] describing computational reflection in the context of programming-in-the-large. For the purposes of our research, the properties are reformulated and exemplified in the context of frameworks for self-adaptive software systems. We have grouped the properties into a reflection prism with three sides: *Self-Representation*, *Reflective Computation*, and *Separation of Concerns*. Each of these and their associated properties are discussed below.

3.1. Self-Representation

Any computational system has a domain model, which corresponds to the type of the application domain (business problem) addressed by the system. In a reflective system, there is a distinction between the domain model and the self-representation. Self-representation is a key characteristic of any reflective software system. We have identified four key properties associated with self-representation.

3.1.1. Type of Representation

A reflective system needs to have access to a representation of itself. For the self-representation we see two distinct approaches described by Maes [11]: (1) *procedural*, the representation is part of the implementation, and (2) *declarative*, the representation is a stand-alone and independent entity. These two cases represent the two end-points on a self-representation continuum. In *procedural* reflection a separate model is not maintained, i.e., the reflective model is used to implement the system. This means that causality is implicit, but on the other hand the single model of procedural reflection must be optimized for multiple usages (i.e., execution and reflection). In *declarative* reflection at least two models are maintained. With multiple models, each model may be optimized for a more specific scope, but multiple models require that the causality is achieved by means of an implementation.

3.1.2. Granularity

Granularity refers to the smallest element of base-level that is reified in the meta-model. The granularity of meta-models pose engineering trade-offs [4]. A fine granularity permits more flexibility and modularity at the cost of larger and more complex models. In the programming-in-the-small, the meta-models typically consist of entities at the granularity of *classes*, *objects*, *methods*, *method calls*, and so on. On the other hand, in the programming-in-the-large, the meta-models typically consist of *components*, *connectors*, *ports*, *ducts*, *styles*, *events*, and so on [6][9][12][14].

3.1.3. Uniformity

Uniformity deals with the degree of variability in the representation of base-level constructs that can be reflected upon. In other words, uniformity is a measure of the homogeneity in meta-models. Maes identifies uniformity as a key criterion for reflective object-oriented systems, and proposes a programming language 3-KRS, whereby every programming language construct is represented as an object, and every object is accompanied with a meta-level equivalent [11]. In the programming-in-the-large, the uniform property can be achieved at a particular level of granularity (recall Section 3.1.2). For instance, the UCI framework provides a self-representation of all the architecturally significant constructs via the equivalent meta-model constructs. Each C2 architectural construct is realized as an object in C2 framework [14], and each object has a corresponding meta-model construct in C2SADL [12].

3.1.4. Completeness

This property expresses whether the information represented in the meta-level is complete [11]. It deals with the extent of information about the base-level system that is available in the meta-level. The completeness has been identified as a crucial property in the context of reflective object-oriented programming languages. The 3-KRS system proposed by Maes uses the meta-level information available about language construct to realize the language interpreter. In the programming-in-the-large, where the meta-models are typically represented at the granularity of the architecture, completeness

corresponds to the information available about the various architectural constructs: component and connector connections, ports and ducts used for the linkage, component dependencies in terms of required and provided interfaces, and so on. The reflective middleware approach in *OpenCom* provides architecture meta-model and an interface meta-model representing the topology of the current set of components within a system and their interactions points [6]. The reflective middleware uses the meta-models for course-grained inspection and adaptation of the structure and behavior of the system at run-time.

3.2. Reflective Computation

The rationale for using reflection in a system varies, and the behavioral properties of a reflective system affect several system properties. These together, make up the *reflective computation* side of the prism.

3.2.1. Type of Reflection

A reflective programming language allows for systems developed using it to reflect on the language construct and potentially modify them. At this level, two distinct reflection variations have been identified: *structural* and *behavioral*. Structural reflection [8] is concerned with the reification of structural program aspects, such as data types. Behavioral reflection [11] is concerned with the reification of computations and their behavior. In practice this means that structural reflection provides access to an application's static structure, such as classes, attributes, and method definitions, while behavioral reflection provides access to the dynamic structures, such as objects' state, messages, and call stack.

Cazzola [5] also identifies two types of reflection at the architectural level that are closely related: *topological* and *strategic*. A system with topological reflection capability performs computations about its own configuration. Examples of reflective activities include the addition and removal of architectural elements. Strategic reflection is concerned with system coordination [7]. For example, observations made on state reifications may trigger change to some coordination policy. Topological and strategic reflections respectively realize structural and behavioral reflections at the architectural level (programming-in-the-large).

3.2.2. Causality

Two entities are causally connected if they are linked in such a way that if one of them changes, it leads to a corresponding effect upon the other [11].

In a self-adaptive system two types of causal relationship may exist. The first type of causal relationship is *between a system and the domain* in which it is deployed. This type of causal relationship is more generally applicable to traditional software systems as well. For example, an autonomous vehicle system has a representation of the domain (e.g., road, cars, traffic lights), and changes in the domain (e.g., movement of other cars on the road) may result in updating the representation of the domain. This in turn may result in a response from the system (e.g., new steering direction), which corresponds to the functionality expected of the system.

The second type of causal relationship is *between a system and its self-representation*. For instance, in the case of the autonomous vehicle, the system may have a representation of itself (e.g., an architectural model). This self-representation may be causally connected to the running system, in which case changes on the self-representation (e.g., adding a component) would result in changes on the software system and vice versa. In the architecture-based approach to self-adaptive software, Oreizy et al. argue for a strict correspondence between the architectural model (at the meta-level) and the executing implementation [14]. The authors propose an *Architecture Evolution Manager (AEM)*. The AEM maintains consistency as changes are applied, reifies changes in the architectural model to the implementation, and prevents changes from violating architectural constraints.

3.2.3. Level Shifts

A level shift happens when the computation switches context from base-level to meta-level and back [4]. A level shift is facilitated using *trap*s. Designated base-entity actions are trapped (caught) by a meta-entity, which performs a meta-computation, then it allows such base-entity to perform the action. Examples of traps in an object oriented context include, among others, access to attributes, method calls, and instance creations. Traps in the programming-in-the-large paradigm are realized using *probes* and *gauge*s [9]. Probes are the system-level units that perform measurements and data collection (e.g., available bandwidth on a network link, interface invocation frequency). Gauges are used to identify particular patterns in the data obtained using probes. If gauges identify changes in the monitored data that violate the system's objectives, meta-computing regarding the system is initiated.

3.2.4. Frequency of Shifts

This is a measure of how frequently the level shifts occur. Level shifts have an impact on the system's performance. How frequently a system shifts levels is an important criteria in the design of both reflective programming language and self-adaptive

systems. An important responsibility of gauges is to filter unnecessary level shifts and minimize the overhead of meta-level computing [9].

3.3. Separation of Concerns

Separation of concerns is a principle tool in any problem solving toolbox. In the context of reflective computation, separation of concerns is pivotal as the reflective behavior itself increases the system's overall complexity. If the reflective system is able to support separate models of different system aspects, possibly at different reflective levels, the overall complexity may be reduced. In this prism surface you find the following properties.

3.3.1. Disciplined Split

Maes emphasizes the importance of a clear separation between the base-level and the meta-level of a reflective system and a causal connection between the models at both levels [11]. The base-level uses the domain model to reason about the domain and provide the system's expected functionality. The meta-level uses the self-representation to reason about the system itself and act upon the system. Consistency is achieved through casually connected models where a change to any of the models is immediately reflected in the other (recall Section 3.2.2). An explicit separation between the domain model and the self-representation is important, given the fact that the two levels deal with different types of concerns. Base-level computation is the common computation we use in every software system. A simple example of a meta-level computation is to keep performance statistics. An advanced example is the scenario described in [9] in which an adaptation engine monitors the response time of a group of servers to client requests and either adds a server in case the server group's load exceeds a predefined threshold, or moves the client to another group in case the available bandwidth between the client and the current server group drops too low. The authors of *OpenCom* argue that the reflective infrastructure (reflective extensions) help to maintain an architectural separation of concerns between system building and system configuration and adaptation [6].

3.3.2. Transparency

This property refers to the extent to which system's base-level is "unaware" of the "reflective levels". The degree of transparency is measured by the number of changes that must be made to the base-level to integrate with the meta-level. For example, in the IBM's autonomic computing architecture [1], the interfacing with managed resources is handled with manageability endpoints (touchpoints). *The Autonomic Integrated Development Environment (AIDE)* enables a developer to describe the manageability capabilities of a managed resource and build automatically manageability interface, transparently to the managed resource. The AIDE tool supports the generation of the manageability capabilities required by WSDM [20] and also the manageability capability defined to represent a Java EE server.

3.3.3. Hierarchy

A reflective system may be hierarchical and allow for several layers, where each layer *n* reflects on layer *n-1* and is reflected upon by layer *n+1*. In other words, models for reflection are defined at different levels of abstraction. Maes discusses reflection at the programming language level. More recent works discuss reflection at the level of architectural constructs [5][6] and aspects [17]. The critical element here is to maintain model consistency when the semantic gap between the models is widened. Cazzola and colleagues address this problem through actuators and system state abstractions [5]. In IBM's autonomic computing architecture, several autonomic elements can be arranged in hierarchies [1]. Kramer and Magee propose a two layer hierarchy, where one layer reacts to exceptions in the base-level by deploying change plans. If no suitable plan is available, computations shift to the top layer, where new plans may be derived in line with the system's overall goals and the current situation [10].

3.3.4. Extensibility

This property deals with how one may reuse, adapt, combine, and integrate reflective layers to form a new reflective tower system [4]. A reflective system is considered extensible, if it is possible to specify new meta-level concerns. As an example, consider a system where the base-level provides the system's functionality and reflective layers that extend the system with support for fault tolerance and persistency. The reflective system is extensible if it can be easily extended with support for another non-functional concern, for example self-optimization. Related to the notion of extensibility is the number of different models that are reflected upon. Okamura, discusses multi model reflection in the context of distributed systems [13]. The inclusion of multiple models for reflection is critical in this domain. However, multiple models add complexity, which affects both the models consistency and the reification performance. One example of extensibility is the orchestration autonomic elements, IBM suggests in their architecture [10]. In *OpenCom*, the reflective middleware provides an extensible set of orthogonal meta-models, each of which is optional and can be dynamically loaded when required, and unloaded when no longer required [6].

4. Case Studies

We performed several case studies from different application domains with the intent to exemplify the crucial properties of the reflection prism and their variation points. For brevity we present two of these case studies below.

4.1. Self-Healing Traffic Monitoring Cameras

Providing scalable solutions for traffic monitoring is an important challenge in intelligent transportation systems. Figure 2 shows an excerpt of a multi-agent system architecture for traffic monitoring. Each node in the system comprises an intelligent camera on which a software agent is deployed that is able to process data of the monitored traffic and communicate with agents on other cameras. The task of the camera agents is to detect and monitor traffic jams on a highway in a decentralized way, avoiding the bottleneck of a centralized control center. Possible clients of the monitoring system are traffic light controllers, driver assistance systems, etc. Since traffic jams can span the viewing range of multiple cameras and can dynamically grow and dissolve, camera agents have to collaborate and distribute the aggregated data to the clients. In a realistic distributed setting, we must consider failures as an essential part of the dynamic environment. Failures will bring the system to an inconsistent state and probably disrupt its services. To make the system capable of dealing with failure dynamics, we have extended the camera with a self-healing subsystem.

The self-healing subsystem monitors the camera base system for failures and only interferes in its operation when a failure occurs. To that end, the self-healing subsystem maintains a dependency model of the camera. The dependencies represented in the model can be architectural, e.g., a connector between interacting components, but also logical, e.g., a commitment between agents. Currently the dependency model is specified at design time. More specifically, the camera is designed to provide a set of interfaces, which allow the self-healing subsystem to query the current values of the relevant dependencies and perform repair actions when a failure occurs according to the applicable repair scenario. To check whether a dependency is still valid, the subsystem periodically requests alive signals from the dependent nodes in the dependency model using the existing communication functionalities of the camera as shown in Figure 2. Repair actions bring the camera back to a consistent state. Examples are updating the set of references to neighboring cameras, removing a commitment with an agent, and removing a communication link in the agent middleware. Currently, the self-healing subsystems only apply local repair actions, i.e., each subsystem locally monitors the dependencies and engages in local repair actions when a failure is detected. There are no negotiations between subsystems with respect to repair actions. The idea is just to bring the camera back to a consistent state from which it can continue its correct operation autonomously. However, more advanced repair actions are possible, such as executing an election protocol to reestablish an ongoing collaboration between agents from which the coordinator has failed. We now apply the reflective prism to the self-healing traffic monitoring cameras.

4.1.1. Self-representation

Type of Representation. The self-representation is declarative and consists of the dependency model that provides a representation of the dependencies of the base system with respect to self-healing.
Granularity. The dependencies refer to diverse aspects of the system ranging from primitive data types such as the identity of a neighboring camera agent to high-level concepts such as a commitment among agents.
Uniformity. Dependencies maintained in the self-representation are uniformly specified and represented as objects.
Completeness. The self-healing subsystem maintains a partial representation of the camera, i.e., only the dependencies relevant for self-healing are part of the dependency model. The self-representation changes dynamically as the state of the system evolves over time.

4.1.2. Reflective Computation

Type of Reflection. The self-healing subsystem is concerned with structural aspects as well as behavioral aspects of the camera. More specific, referring to Cazzola [4], the self-healing subsystem includes

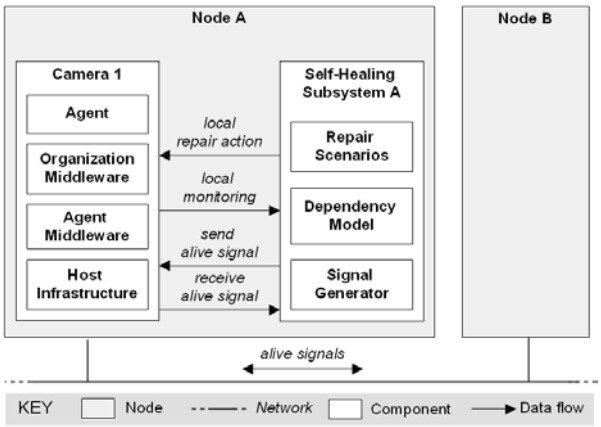

Figure 2. Architecture for Self-Healing Traffic Monitoring Cameras.

aspects of topological as well as strategic reflection, although the latter is not exploited to its full extent, since currently only local repair actions are supported. Distributed repair actions would enable changing the coordination among agents when necessary.

Causality. The causality between the domain and the domain model of the camera is established through the monitoring of the traffic conditions. Guarantees about the causality are based on best effort. To establish a causal connection between the camera and the self-representation, the self-healing subsystem monitors the camera via dedicated interfaces. The dependencies of the camera with cameras on other nodes refer to the availability of other cameras and can be considered as part of the domain. The causality between these representations and the actual status of the cameras is based on the exchange of alive messages among cameras. Guarantees about the causality are based on best effort.

Level Shifts. The self-healing subsystem interferes with the camera only when a failure is detected. However, background activities are required to maintain the self-representation. As explained above, the camera provides dedicated interfaces that allow the self-healing subsystem to monitor the relevant aspects of the camera and perform repair actions when needed.

Frequency of Shifts. Since node failures are expected to happen rarely, the frequency of level shifts is very low.

4.1.3. Separations of Concerns

Disciplined Split. The self-healing subsystem is concerned with node failures. The subsystem cleanly separates this concern from the domain concerns of the camera (i.e., monitoring traffic jams and informing interested clients).

Transparency. In the current version of the system, the camera is designed to support self-management. Dedicated interfaces enable the self-healing subsystem to access the camera, so there is no transparency. Ongoing research investigates how techniques from aspect-orientation enable a clean separation of concerns. Nevertheless, whether self-healing can be achieved in a non-invasive manner remains an open research question.

Hierarchy. The self-healing subsystem is a single layer reflective system. The policy of self-healing is encoded in the dependency model and the repair scenarios. In order to deal with additional, probably more complex failure scenarios, an additional layer may be introduced that reflects on the activities of the self-healing subsystem and adapts the repair policy or selects an alternative policy under certain conditions.

Extensibility. The current self-healing subsystem is conceived as an independent reflective subsystem. Extensibility and reuse were not main concerns in the design of the subsystem. The aspect-oriented approach referred to above aims to bring these important quality attributes to the forefront.

4.2. High-Performance Computing

The concrete application for this scenario is a sensor network. It consists of geographically distributed digital receptor units connected to computing facilities with a high-speed network. In this scenario, the system should dynamically schedule and deploy experiments. The problem is that experiments and analyses often are data-parallel programs that must be scheduled on the underlying hardware architecture. To address these issues we have developed the Dynamic Model-Driven Architecture (DMDA).

The DMDA is a platform that combines three architectural principles, Model Driven Architecture (MDA), layered architectures, and dynamism. These three combine into an architecture that supports: (1) programmability, since end-users, i.e., experiment developers, are mainly non-computer scientists, the development and deployment of the application must be intuitive and straightforward, (2) adaptability, the platform must have means to deal with run-time change to continuously satisfy QoS requirements, including changed conditions in the observed world and resource availability.

A DMDA level, depicted in Figure 3, includes pairs of Adaptation Engines and System models organized in layers. In this architectural principle each level is similar to the reflective reference model described above. The bottommost level interfaces with the running system directly or indirectly via a system model. Each level contains a *Generator* that produces new/updated models, a set of *Actuators* that achieve *intercession* on the system models using a set of *Probes* that *introspects* the models, and a *Coordinator* which is responsible for coordinating actions and probing the layer immediately above to maintain model causality. We now apply the reflective prism.

4.2.1. Self-Representation

Type of Representation. The self-representation is a separate entity and not part of the base-level implementation.

Granularity. The self-representation granularity in our current implementation is *task*. Each task corresponds to some piece of code to be executed on a *processor* in the system.

Uniformity. The self-representation is not uniform. The self-representation is structured and represented differently at different layers in the current implementation.

Completeness. The self-representation is partial. The available information is restricted to task structures,

and the information forwarded from probes and retrieved by coordinators.

4.2.2. Reflective Computation

Type of Reflection. The reflection is structural, for instance, generating new deployments, and behavioral, e.g., when QoS parameters are changed.

Causality. Causality needs to be investigated from two perspectives. DMDA is a platform and hence not responsible for maintaining causality for applications running in the system. The domain of the DMDA is parallel computing. The causality between the base-level and meta-level is maintained by restricting the capabilities of a task on the base-level. Structural system modification is the result of a deployment action initiated by the DMDA system. Changes to the executing system, for instance, node failures, should be probed and information concerning such changes forwarded to the DMDA system. Within the reflective system causality is dealt by the Coordinator, in a similar manner.

Level Shifts. Level shifts may be triggered from the top-down or bottom-up. If a new experiment is added to the platform, this triggers a level shift top-down where the new experiment is scheduled and prepared for deployment, while a node failure triggers a bottom-up shift.

Frequency of Shifts. Level shifts are assumed to happen rarely, hence the frequency is considered to be low.

4.2.3. Separation of Concerns

Disciplined Split. The split between the base-level and the bottommost meta-layer is disciplined. In the DMDA, the base level hosts experiments, while the meta level is dealing with the management of experiments. The meta-level retrieves application specific information from the base-level to support decision making and construction of management directives.

Transparency. The current system implementation is not transparent. The base system contains probe and actuator programs. The nature of a data-parallel application requires that all code is scheduled on the target machine; hence this kind of functionality is a part of the base-level system.

Hierarchy. DMDA is a multi-layer reflective system. Each layer is a reflective system working on layer specific models and communicates only with the level immediately below and actions on "higher levels" are "triggered" by probes.

Extensibility. The layered structure of the DMDA reference architecture conceptually makes it easy to add a new layer. In practice it is not as straightforward since models must be adjusted to layers immediately above and beneath. The current DMDA implementation is restricted to one property, i.e., performance (completion time). We have tested other criteria but not combinations of criteria. Hence, we cannot express how extensible the system currently is.

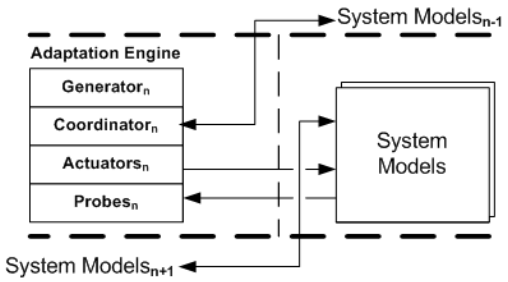

Figure 3. DMDA Reference Architecture.

5. Challenges

The application of reflective prism to different case studies has led us to identify key challenges when building self-adaptive systems. The following list of challenges should not be considered exhaustive. They have been grouped in key headings, but most are highly interrelated.

5.1. Expressiveness of Self-Representation

Self-adaptive software systems require a representation of the system at a suitable level of abstraction. Our experience indicates that software architecture has been shown to provide an appropriate level of granularity for self-representation. A variety of Architectural Description Languages (ADLs) [12] have been utilized for modeling a software system's architecture. However, the existing ADLs suffer from several shortcomings when employed for self-representation.

Firstly, our study has revealed that in the programming-in-the-large the notion of domain plays an important role that must be accounted for at the meta-model level. At the same time, existing research on ADLs has often focused on providing generic solutions for modeling the base-level characteristics [12], while in most self-adaptive systems there is an intrinsic causal relationship between the system's domain and its self-representation that needs to be represented. Moreover, the majority of existing ADLs are intended for modeling the design decisions, and less suitable for representing the dynamic, in some case unpredictable, behavior of self-adaptive systems at run-time. We believe further research is needed in the development of domain-specific ADLs that incorporates not only the system's properties, but also domain characteristics. This could pave the way in managing the causal relationship between the domain and self-representation.

Secondly, most existing ADLs do not provide adequate expressiveness for the level of dynamism that characterizes self-adaptive software systems. Moreover, an ADL intended for self-adaptive software systems needs to provide support for distinguishing between the base-level and meta-level representations. ADLs in this setting should also account for the complexity of self-representation in distributed setting.

5.2. Meta-Level Conflicts

Self-adaptation is concerned with preserving or changing a system's functionality and quality footprint. Ever-present grand challenges for software engineers are extensibility and composability. A seamless introduction of this kind of support in a self-adaptive system requires multiple-model and multiple meta-computation support. In our study we have experienced a number of problems related to managing multiple models at the meta-level. For example, in the DMDA, each layer uses one or more models, it also interfaces with models on other layers. How to effectively manage all models and adapt functionality for seamless model integration remains an issue

Maintaining multiple models brings several challenging research issues to the table; multi-model consistency and multi-model causality are two examples. In addition, multiple models require that mechanisms for uniformity are developed that can bridge currently heterogeneous representations.

A seamless integration of meta-level computations is another challenge. Multiple meta-level computations may, for instance, realize several quality attributes and extensive support is critical for the system's extensibility. In the long term, we may see configurable standard models and computations, which represent specific system aspects, integrated and reused in self-adaptive systems. In this setting, model management frameworks provide developers with a standardized, yet configurable, architecture for efficient management of multiple aspects in self-adaptive software systems.

5.3. Uncertainty

In order to perform its reflective computation properly, a self-adaptive system needs an accurate self-representation. Typically, this representation is an abstraction of (particular aspects of) the base-level system. In addition, a self-adaptive system often needs to take into account domain information. For example, in the self-healing traffic monitoring cameras, a self-healing subsystem requires information about the status of cameras on other nodes.

The need for an abstract representation of the base-level system and domain information introduce uncertainty. First, uncertainty can result from inaccurate models. Providing an abstract representation of the base-level may introduce certain inaccuracies, which also cannot be avoided when modeling relevant aspects of the domain. Second, uncertainty can be caused as a result of ensuring causality. Implementing causality is challenging, in particular in uncertain real-world settings. Since in general one can have no guarantees about the domain, if causality cannot be ensured, the lack of consistency between the domain and its model introduces uncertainty when using this model to adapt the base-level of the system.

Tackling the problems related to uncertainty is challenging. In particular since some of the causes of the problems are not under control of the designer. To deal with the challenges of uncertainty, we need (1) suitable languages that abstract self-representations (see Section 5.1), (2) domain models that take into account uncertainty explicitly, (3) disciplined approaches for maintaining causality between the different types of models at run-time (see Section 5.2).

5.4. Autonomy

A self-adaptive system is expected to act autonomously, i.e., based on its self-representation, and the ability to interpret and manipulate this representation, the system should be able to make its own decisions for achieving some goal. One way for achieving autonomy is to make use of feedback control loops that incorporate the principles associated with monitoring-analysis-plan-execute (MAPE) from autonomic computing [1]. While the "plan" activity provides the ability for the system to reason about itself, the feedback loop provides the necessary robustness to deal with uncertainty. Since a system depending on its different goals might have several representations of self, a clear challenge that emerges is how to solve meta-level conflicts (see Section 5.2). In addition, such systems should not be considered in isolation, they are usually part of larger more complex systems. To deal with scalability, these systems are typically decentralized, which raises further challenges. How to achieve self-representation in decentralized setting? How to identify and achieve a common goal? How conflicts in the process of decision making can be solved in such a setting?

5.5. Transparency

While it may be straightforward to achieve transparency for reflective programming languages, it is a major challenge for complex self-adaptive software systems. Techniques for probing and actuating have been proposed in various frameworks (e.g., touchpoints [1], probes and effectors [9]). However, the degree to which these techniques can be implemented transparently to the base system is not well studied. A

promising idea to achieve transparency is to employ techniques from aspect-orientation [19]. On the other hand, since for many concerns considered in self-adaptive systems (self-healing is a typical example) there is a tight connection between the base system and the meta-system, one can argue that instead of aiming for full transparency, it might be better to develop disciplined approaches that integrate a base system with a meta-system. In line with this idea, some researchers have proposed new component models to build self-adaptive systems. An interesting example is the Fractal component model that endows system components with customized introspection and intercession capabilities [3].

5.6. Performance

A commonly-cited disadvantage of interception is that it incurs an inherent performance overhead. Interception typically requires bindings to support a level shift. In general, research on self-adaptive system pay little attention on performance issues. Example approaches that provide dedicated support for interception based on the requirements of the application at hand are proposed in [3] and [6].

6. Conclusion

Development of self-adaptive software systems poses significant challenges to the software engineering community. These challenges are exacerbated by a lack of consensus among researchers on very fundamental concepts. We have attempted to alleviate this issue by exploring the crucial role of computational reflection in the context of self-adaptive systems. Our study resulted in the identification of key reflection properties and their variation points. Analyzing several self-adaptive software systems from different domains in light of these properties has helped us to verify our initial hypothesis that reflection is a necessary criterion for any self-adaptive software system. Moreover, our experience shows that while the theory of computational reflection, which was initially developed by the programming languages researchers, provides firm grounding for the study of self-adaptive system, it needs to be extended to accommodate the unique challenges posed by the programming-in-the-large. We hope these results can serve as an impetus to propel the future research in this area.

7. Acknowledgments

This work is partially supported by grant CCF-0820060 from the National Science Foundation, and the Research Foundation Flanders (FWO).

8. References

[1] An Architectural Blueprint for Autonomic Computing, IBM White Paper, 2006.

[2] J. Andersson, et al. Towards a Classification of Self-Adaptive Software System. *Software Engineering for Self-Adaptive Systems,* Eds. B. H. C. Cheng, et al. LNCS, 2009. (to appear)

[3] E. Bruneton, et al. The Fractal Component Model and Its Support in Java. *Journal of Software Practice and Experience*, 36(11-12), 2006.

[4] W. Cazzola. Evaluation of Object-Oriented Reflective Models. *Workshop on Reflective Object-Oriented Programming and Systems*, London, 386-387.

[5] W. Cazzola, et al. Rule-Based Strategic Reflection: Observing and Modifying Behaviour at the Architectural Level. *Int'l Conf. on Automated Software Engineering*, Oct 1999.

[6] G. Coulson, et al. A Generic Component Model for Building Systems Software, *ACM Transactions on Computer Systems*, 26(1), 2008.

[7] C. E. Cuesta, et al. Dynamic Coordination Architecture through the Use of Reflection. *ACM Symposium on Applied Computing,* Las Vegas, Nevada, 2001.

[8] J. Ferber. Computational Reflection in Class Based Object-Oriented Languages. *OOPSLA*, New Orleans, Louisiana, October, 1989.

[9] D. Garlan, et al. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *IEEE Computer*, Vol. 37, No. 10, pp. 46-54, Oct 2004.

[10] J. Kramer, and J. Magee. Self-Managed Systems: an Architectural Challenge. *Int'l Conf. on Software Engineering*, Minneapolis, Minnesota, May 2007.

[11] P. Maes. Concepts and Experiments in Computational Reflection. *OOPSLA*, Orlando, FL, Oct 1977.

[12] N. Medvidovic, et al. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. on Soft. Eng.*, vol. 26, 2000.

[13] H. Okamura, Y. Ishikawa, and M. Tokoro. AL-1/d: A Distributed Programming System with Multi-Model Reflection Framework. *Workshop on New Models for Software Architecture*, November 1992.

[14] P. Oreizy, et al. Architecture-Based Runtime Software Evolution. *ICSE*, Kyoto, Japan, May 1998.

[15] B. C. Smith. Reflection and Semantics in LISP. *Symposium on Principles of Programming Languages*, Salt Lake City, Utah, 1984.

[16] E. Tanter, et al. Partial Behavioral Reflection: Spatial and Temporal Selection of Reification, *OOPSLA*, Anaheim, CA, 2003.

[17] E. Tanter. From Metaobject Protocols to Versatile Kernels for Aspect-Oriented Programming. PhD Thesis, University of Chile, 2004.

[18] F. Tisato et al. Architectural Reflection: Realising Software Architectures via Reflective Activities. W. Emmerich and S. Tai, Eds. *LNCS 1999*, London, 2001.

[19] E. Truyen and W. Joosen, Towards an aspect-oriented architecture for self-adaptive frameworks. *Workshop on Aspects, Components, and Patterns for Infrastructure Software*, Brussels, Belgium, 2008.

[20] WSDM 1.1 OASIS Standard, 2006.