Models@Runtime to Support the Iterative and Continuous Design of Autonomic Reasoners*

Franck Chauvel, Nicolas Ferry, Brice Morin,
Alessandro Rossini, and Arnor Solberg

SINTEF ICT - MOD Group
Oslo, Norway
{First.Last}@sintef.no

Abstract. Modern software systems evolve in a highly dynamic and open environment, where their supporting platforms and infrastructures can change on demand. Designing and operating holistic controllers able to leverage the adaptation capabilities of the complete software stack is a complex task, as it is no longer possible to foresee all possible environment states and system configurations that would properly compensate for them. This paper presents our experience in using models@runtime to foster the systematic design and evaluation of self-adaptive systems, by enabling the coevolution of the reasoning engine and its environment. This research was carried out in the context of the Diversify project, which explores how bio-diversity can be used to enhanced the design of self-adaptive mechanisms.

Keywords: models@run.time, self-adaptation, cloud computing

1 Introduction

Modern software systems are increasingly complex and distributed as they materialize into large scale assemblies of technologies, such as databases, middleware, business logics or graphical front-ends. Typically, cloud-based systems push this complexity even further as their platform and infrastructure, traditionally fixed once and for all, can now be dynamically adjusted [17].

This complexity exceeds by far the capabilities of IT teams, who need to rely on software solutions to automate as much as possible the related maintenance operations [15]. Load-balancing is a typical example easily automated by cloud providers using basic rules. This need for automation is addressed by self-adaptive systems [7], which adjust their behavior to their changing environment. However, the design of self-adaptive systems, especially of their reasoning engine, remains an open challenge which often results in *ad hoc* solutions [7,21] tailored to specific environments.

* This work has partially been funded by the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement numbers: 318484 (MODAClouds), 317715 (PaaSage), 257793 (REMICS) and 600654 (DIVERSIFY).

A key issue, while designing such systems, is the *uncertainty* pertaining to their open, dynamic and heterogeneous environment [10]. Designers and even tools are no longer able to statically foresee all possible environment states as well as all the system configurations that would properly handle them. In the cloud settings for instance, the virtually infinite set of resources combined with the ever changing requirements, prevent to anticipate all the possible underlying platforms and infrastructures. Unfortunately, unforeseen environments are likely to produce an outage and in turn, to generate undesirable overpriced maintenance operations.

To minimize this risk of not being able to manage unforeseen environments, our contribution is twofold. We first propose an iterative process to *proactively* explore possible evolutions of the environment, and allow designers to incrementally adjust the reasoning engines accordingly. We then describe how the models@runtime paradigm [3] can foster the evolution of the reasoning engine together with the system's environment.

In the context of the Diversify project [1], we combined existing self-reparation techniques with a mechanism for exploring environment evolutions driven by an analogy with the concept of bio-diversity. The loose coupling between the reasoning engine and the environment provided by the model@runtime paradigm facilitates their interdependent evolution. On one hand, by replacing the running system with simulated environments, one can evaluate the effectiveness of a given reasoning engine. On the other hand, by plugging alternative reasoning engines, one can evaluate their robustness to unforeseen environments.

The remainder of this paper is organized as follows. Section 2 uses a cloud-based multi-tenant system as an example of a self-adaptive system facing unforeseen environments. Section 3 presents our design process and its realization on a models@runtime platform. Section 4 details how this approach was applied to our example in the context of the Diversify project. Section 5 discusses related work before Section 6 concludes and discusses possible future work.

2 Motivating Example: Unforeseen Multi-tenancy

SensApp [19] is an open-source platform[2] fostering the seamless integration of the Internet of Things (IoT) with the cloud. SensApp users can register sensors into the platform, which will push data available to other users or services from a REST interface. SensApp users and third-party services can subscribe to notifications: for instance, when the temperature of a given building is exceeding a given threshold.

SensApp is implemented as four REST services to: *i*) register sensors, *ii*) store their data, *iii*) notify third-party services, and *iv*) to orchestrate these services from a unified facade service (dispatcher). As these services are loosely coupled (they only use their public REST APIs to communicate between them), SensApp can thus be distributed in various ways. Each service indeed only

[1] http://diversify-project.eu/

[2] https://github.com/SINTEF-9012/sensapp

requires a servlet container, such as Apache Tomcat or Jetty, and a set of open ports to communicate over HTTP.

In order to reduce the management overhead related to operating SensApp in the cloud, we need to develop a self-adaptive version of SensApp, which shall recover as much as possible from failures of its own components (*i.e.*, its four services) and from failures occurring in its environment (*i.e.*, any failures of the underlying software stack including software, platform and hardware infrastructure). In a cloud setting, software failures are recovered by reinstalling the erroneous component, whereas hardware failures require in addition the provisioning of a new virtual machine. Our first solution to self-adaptation relies on a set of event-condition-action rules (ECA) binding some anticipated failures with imperative repair procedures. These procedures are sequences of atomic actions such as provisioning a virtual machine or deploying a service, which apply on a CloudML model (*cf.* Section 4) describing the system and its environment [11].

As cloud-based systems operate in an open environment, it is very difficult (and even not tractable) to get insight on the impact that unforeseen environments may have. In a real cloud setting, nothing guarantees for instance that the SensApp application will run in complete isolation: other applications or services may be added into the same war container, or simply be hosted on the same virtual machines. From the SensApp perspective, this kind of environment with multi-tenancy might alter its performances (*e.g.*, response time) or the associated reasoning engine (*e.g.*, decision relevance).

Existing approaches to manage such uncertainty (*cf.* Section 5) either let the system learn from its errors using machine learning techniques or directly leverage a model of the uncertainty itself. Machine learning techniques are robust but require to let the system's performance drop for a moment, until the system has learned. Modeling uncertainty to make better decisions implies the existence of partial knowledge, which is not always available.

Designing a self-adaptation mechanism, robust to unforeseen environments, thus remains a challenge. In the following, we explore how an alternative approach, inspired by agile practices, can be used to enhance the reasoning engine of our autonomic SensApp in terms of effectiveness and performances.

3 Approach Overview

We present in this section the design process, inspired by agile practices, and its underlying models@runtime architecture.

3.1 An Iterative Design Process

Our overall approach relies on an iterative process that successively refines the design of the reasoning engine, and the environments in which it operates, until enough confidence is gained regarding the robustness of the system. The interdependent evolutions of the reasoning engine and its environment (see the two cycles in Fig. 1), referred to as *coevolution*, is organized as follows:

1. **Check robustness to unforeseen environment**: assess whether designers have gained enough confidence regarding the robustness of the system under unforeseen environments. Depending on the concern of interest, this process can be automated. Once this evaluation is successful, the overall design process is terminated. Otherwise, a new environment simulator is designed.
2. **Design a new environment simulator**: First, the designer identifies an uncertainty axis along which she suspects that the robustness could be an issue. Based on this, the simulated environments (potentially including some randomness) will evolve along this axis.
3. **Evaluate the reasoning engine**: Evaluates how the reasoner behaves with respect to the simulated environment. Depending on the concern of interest, this process can be automated. If the reasoning engine passes the evaluation then the design process restarts from Step 1, otherwise the reasoning engine is updated (Step 4).
4. **Update the reasoning engine**: Adapts the reasoning engine to the simulated environment. It may lead to a change of reasoning engine as well as in its configuration. Once updated, the new engine is evaluated again (Step 3).

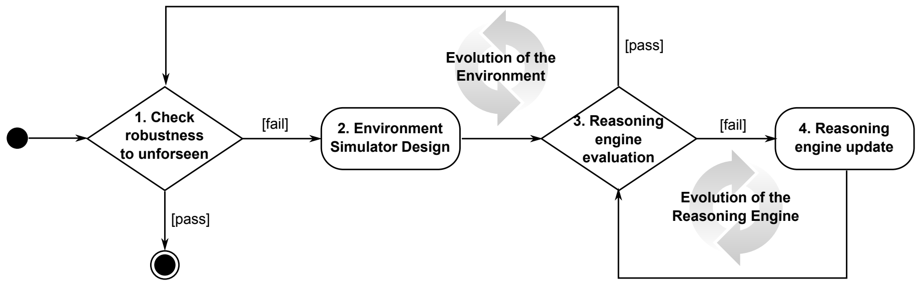

1. Check robustness to unforseen
[fail]
[pass]
2. Environment Simulator Design
Evolution of the Environment
[pass]
3. Reasoning engine evaluation
[fail]
4. Reasoning engine update
Evolution of the Reasoning Engine

Fig. 1. Proposed iterative design process, depicted as a UML activity diagram

Appying coevolution to address unforeseen environment assumes that the *Pareto Principle* [4] (also known as the 80/20 rule) applies to self-adaptive systems design. This principle states that 80 % of the effects are due to 20 % of the possible causes. This means that 80 % of issues due to unforeseen environments will be reproduced by only 20 % of them. Our approach aims at exploring unforeseen environments using random generation techniques, while targeting the key 20 % using a proper control of diversity, as done in evolutionary algorithms [22] for instance.

3.2 A Models@Runtime Architecture to Support the Process

The proposed architecture is depicted in Fig. 2 and is inspired by our previous work [18]. The models@runtime engine acts as an intermediary layer between the reasoner and the runtime system with its environment.

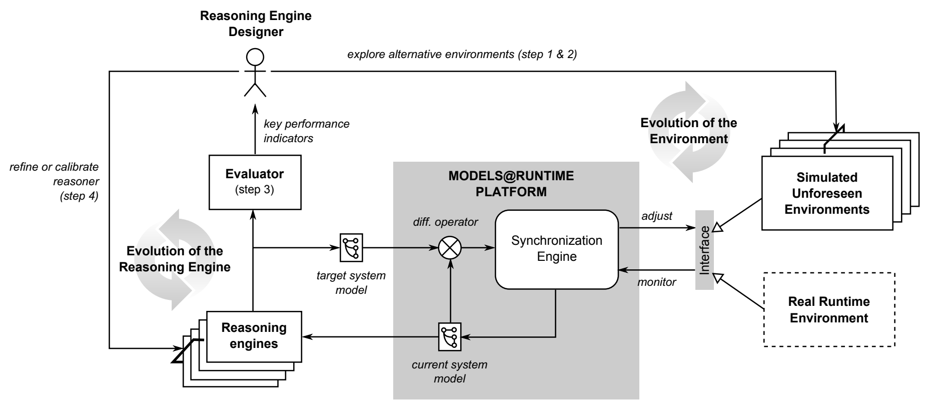

Reasoning Engine Designer
explore alternative environments (step 1 & 2)
key performance indicators
refine or calibrate reasoner (step 4)
Evaluator (step 3)
Evolution of the Environment
MODELS@RUNTIME PLATFORM
diff. operator
Synchronization Engine
adjust
monitor
Interface
Simulated Unforeseen Environments
Real Runtime Environment
Evolution of the Reasoning Engine
target system model
Reasoning engines
current system model

Fig. 2. The overall DIVERSIFY architecture, leveraging the models@runtime paradigm to coevolve (see the two cycles) the reasoning engine and its environment

The models@runtime idea is reused as such in Fig. 2, where a synchronization engine maintains the causal link between the system and the model. Any change in the environment is reflected on the model, and similarly, any change in the system model is enacted on the system. The interactions between the models@runtime engine and both interfaces consist in exchanging models or manipulating the models through an API. A reasoner can retrieve the current model of the running system and push a new target model to the engine, while the runtime environment can push a new model of the current running system to the engine.

In our approach, the configuration or replacement of a reasoning engine or of the environment is enacted by a designer. In order to help the designer in this task, an evaluator is used to assess whether the reasoner behaves correctly with respect to the simulated environment and its variations.

The Role of Models@Runtime: In this context, the separation of concerns offered by models@runtime brings three main benefits: *(i)* easier integration of the reasoning engine,*(ii)* easier integration of environment simulators, and *(iii)* coevolution of the reasoning engine and the simulator.

From the reasoning engine's point of view, models@runtime provides both abstraction and anticipation capabilities. The abstraction provided by the model let reasoning engine to analyze a simplified version of the running system, while the anticipation allows the reasoning engine to conduct *what-if* scenarios in a safe modeling space, with no impact on the running system. If and only if a valid model is identified, it will be automatically enacted via the causal link and the running system will consequently be dynamically adapted. If the reasoning engine at some point yields an invalid model, it is simply discarded, with no need to perform expensive roll-back since the system has not been adapted yet. The reasoning engine can be applied to any environment including simulators as far as it can be modelled by the models@runtime engine (*i.e.*, it respects the interface).

From the environment point of view, the models@runtime architecture facilitates the simulation of alternative environments, by replacing the actual runtime environment by simulated ones. The ability to easily switch between environments and the loose coupling between the reasoning engine and the environment, avoid the recurrent design of *ad hoc* simulators tightly-coupled to a specific reasoning engine. Moreover, once the design process is completed, integrating the resulting reasoning engine with the running system does not require any further change on the self-adaptation loop.

The overall architecture relies on models@runtime, as it provides a clear interface and a loose coupling between both the environment and the reasoning engine sides, which facilitates their *coevolution*. This in turn facilitates the continuous design of complex adaptive systems.

The next section reports on how we used this process and the related architecture in the context of the DIVERSIFY project.

4 The Diversify Experiment

The DIVERSIFY project aims at creating a synergy between ecology (a discipline of biology), and software engineering. We aim at understanding the key mechanisms underlying bio-diversity and how they enhance the robustness and resilience of biological systems, and thus porting them to software systems. We in particular focus on cloud systems, which offers a large scale, distributed and complex software ecosystem, approaching the complexity of natural ecosystems.

4.1 Modeling Deployment and Provisioning of Cloud-based Systems

We model the deployment and provisioning of cloud systems with CLOUDML, a DSML developed as a joint effort between the REMICS, MODAClouds and PaaSage projects. As shown on Fig. 3, a CLOUDML model captures the deployment of a cloud-based system, including the platform and infrastructure level. In this example, SENSAPP is deployed on three virtual machines hosting: the "admin" application, the SENSAPP application, and the underlying database, respectively. The semantics of such a description is embedded in the models@runtime platform, which maintains a causal link between the model and the running cloud system.

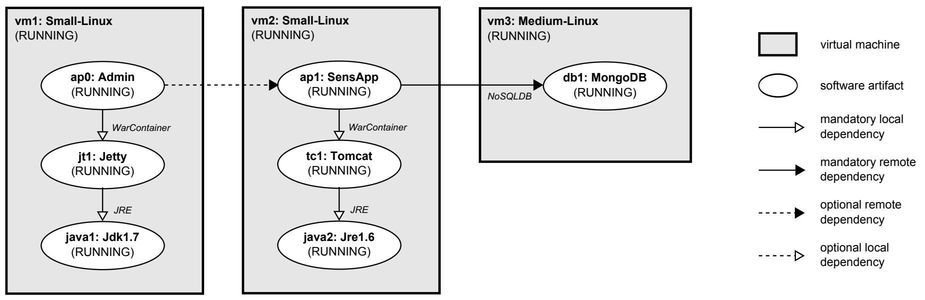


Fig. 3. A possible distributed deployment of SensApp, modeled using CloudML

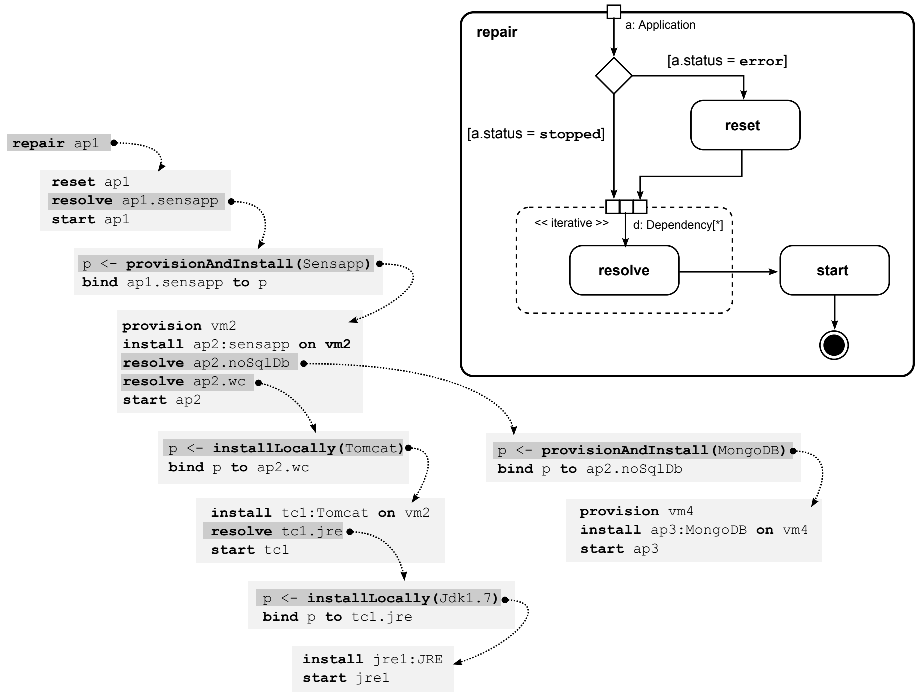
In order to observe how software diversity helps self-repair mechanisms in recovering failures, the notion of failure is reflected on the model by the state of the entity being switched from `running` to `error`. Failures are also propagated in the model along mandatory dependencies, as a software artifact cannot operate without them.

Our first self-repair algorithm is specified as a set of ECA rules, which bind potential failures to some predefined repair procedures. We identified eleven atomic modifications on CloudML models, (*e.g.*, provision of new node, deployment of a given application, starting a given application), which are combined into repair procedures. Fig. 4 depicts the `repair` procedure, which, given an application, generates the following repair plan: if the application is in an erroneous state, it first resets the application, then it resolves all missing dependencies, and finally starts the application. It is worth to note that the resolution of a dependency is a separate procedure (called `resolve`), which is itself resolved to generate the final plan. In Fig. 4, the dotted arrows depict the refinements of `repair` procedure according to the CloudML model introduced in Fig. 3, where the `vm2` failed. Four procedures are initially defined: `resolve`, `install locally`, `provision and install`, and `repair`. This planning technique is referred in the literature as hierarchical tasks networks (HTN) [12], whose tasks (in our case) are strictly ordered.

4.2 Applying our Iterative Process on SensApp

The following paragraphs summarize how we executed the iterative process presented in Fig 1, on SensApp. In our experiment, the random generation of alternative environments is automated, whereas their integration in the models@runtime architecture is still done manually, as is the evaluation of key performance indicators.

Check Robustness to Unforeseen Environments The first step of the process evaluates the confidence the designer has in the system to operate properly under unforeseen circumstances.



```
repair ap1

  reset ap1
  resolve ap1.sensapp
  start ap1

    p <- provisionAndInstall(Sensapp)
    bind ap1.sensapp to p

      provision vm2
      install ap2:sensapp on vm2
      resolve ap2.noSqlDb
      resolve ap2.wc
      start ap2

        p <- installLocally(Tomcat)
        bind p to ap2.wc

          install tc1:Tomcat on vm2
          resolve tc1.jre
          start tc1

            p <- installLocally(Jdk1.7)
            bind p to tc1.jre

              install jre1:JRE
              start jre1

        p <- provisionAndInstall(MongoDB)
        bind p to ap2.noSqlDb

          provision vm4
          install ap3:MongoDB on vm4
          start ap3
```

Fig. 4. An example of repair procedure template, depicted as UML activity diagram, and the sequence of refinement steps needed to extract the related concrete repair plan

1. Having a clear definition of the objectives of the self-adaptive system is the *sine qua non* condition to any assessment of its effectiveness or its performance. Some self-adaptation techniques such as control theory, optimization methods, or planning techniques require a formal definition of those objectives, but others, such as ECA rule sets, do not. In SensApp for instance, these objectives are not explicit, but the effectiveness and the performance of the reparation process have implicitly driven the selection of the rules.

2. Once the objectives of the system are defined, the designer must identify the key *axes of uncertainty* in the system's environment. SensApp may be subject to failures and may be deployed on a multi-tenant environment, where other applications that cannot be controlled may evolve and impact SensApp's behavior. Multi-tenancy and failures are thus the axes of uncertainty investigated hereafter.

3. Finally, designers have to prioritize the different axes of uncertainty according to their impact on the requirements and the system's objectives: the multi-tenancy hypothesis in the case of SensApp.

Since the multi-tenancy hypothesis is critical, the designer should then simulate failures in environments including applications evolving out of SensApp's control.

Design a "Diversified" Environment Simulator The multi-tenancy hypothesis can be reproduced by breaking down the isolation in which the system is running, which can be done by artificially controlling the level of software diversity in the environment. Indeed, isolation can be seen as a lack of software diversity in the system's environment. Interested reader may consult [14] for further details on biodiversity.

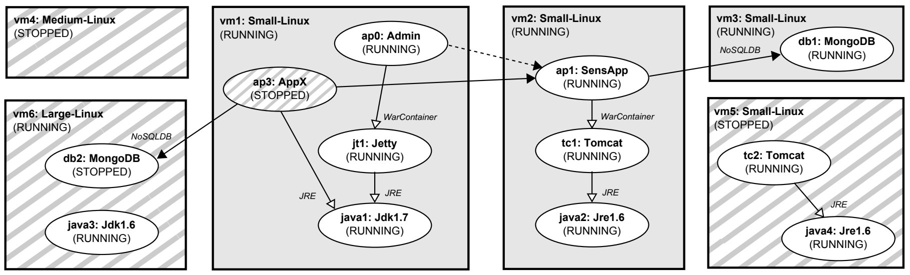


Fig. 5. A "diversified" distributed deployment of SensApp, modeled with CloudML

Fig. 5 depicts an example of a simulated environment where diversity was injected into SensApp (see crosshatched virtual machines and applications). Three extra virtual machines are provisioned (*i.e.*, `vm4`, `vm5`, and `vm6`) where additional applications are randomly deployed. In addition, applications are also randomly injected into existing virtual machines (*i.e.*, `ap3` is deployed on `vm1` and is bound to the existing JRE as well as to the new remote applications). Our simulation introduces all possible failures of the SensApp system (*e.g.*, failure of `vm2`).

Evaluation of the Reasoning Engine This step evaluates SensApp against randomly generated diversified environments, to check how well its self-adaptation mechanisms behave under multi-tenancy. This evaluation covers two dimensions: the effectiveness of the reparation and its performance.

The effectiveness of the self-repair strategies of SensApp are measured by its ability to restore itself into a `running` state. A repair is successful if and only if its execution results in a valid deployment of SensApp. Then, the manual inspection of simulation results showed that the effectiveness is not impacted by multi-tenancy.

In addition, the performance of the repair procedure is also measured and is defined as the cost of executing the resulting repair plan. In a cloud setting, different repair actions may have different costs: expensive actions are typically the provisioning of a new node and the deployment of applications on existing virtual machines (as it may require heavy network traffic). In this respect, simulation results revealed that the generated plans are suboptimal, as the planner provisions new virtual machines for each failure, without leveraging available opportunities in its environment.

Update of the Reasoning Engine Several alternative designs may help mitigating the performance drop under multi-tenancy, namely the modification of ECA rules or the replacement of the reasoning engine.

The first update consisted in modifying the set of ECA rules and evaluating their effectiveness and performance (Step 3). This updated reasoning engine preserved the scalability of the reasoning engine, but may still miss optimal plan.

A second update led to the use of automated planning techniques able to reach the optimal solution. Manual investigations of planning traces showed that generic automated planning techniques do not scale to large systems (about 30 entities in the models).

5 Related Work

Uncertainty has been identified has one of the key obstacles to the design and development of self-adaptive systems [10]. We discuss below a selection of related approaches, namely specific theoretical frameworks, machine learning techniques, agile practices and risk analysis methods.

From a theoretical standpoint, several frameworks have emerged to capture different types of uncertainty [20]: probability theory focusing on *eventuality*, fuzzy sets theory focusing on *cognitive imprecision*, or grey systems theory to handle *incompleteness*. Existing approaches such as RELAX [23], FLAGS [1] or POISED [9] integrate some of those theories to build reasoning engines making the most of the partial knowledge available. Our approach is complementary and can be used to evaluate the robustness of these systems, when they operate in environments which were not initially taken into account at design time.

An alternative approach to uncertainty is to rely on online machine learning. Approaches such as FUSION [8] or RESIST [5] result in systems able to learn from their inability to handle unforeseen environments. Whereas these approaches accept to let the system effectiveness drop for a short time (until it has learned), our approach aims at searching, *a priori*, for possible challenging environments and adjusting the reasoning engine accordingly. These two approaches are not conflicting either and could be combined to minimize for instance the need for online learning.

In contrast to the above approaches, our iterative process (*cf.* Fig. 1) shares more with the ideas of *agile practices* [16] such as test-driven development (TDD) [2]. Agile practices advocate the continuous feedback from the end-users (using more frequent and shorter development iterations). They aim at breaking down the complexity, by using incremental development, where at each step, developers augment the system with a small number of features, and then validate it with the end-users. Similarly, our process also breaks down the complexity of self-adaptation mechanisms, by gradually addressing the various axes of uncertainty existing in the environment. To the best of our knowledge, this research is a first attempt to apply agile practices to self-adaptive system design.

Our iterative process is also inspired by existing risk analysis methods such as CORAS [13]. Risk analysis traditionally implies to identify key assets, major threats on these assets, and possible mitigations which could minimize the associated risks. In our process, assets are the effectiveness and performance whilst threats are the sources of uncertainty and mitigations are update of the reasoning engine. To the best of our knowledge, this is also a first attempt to combine risk analysis and self-adaptive system design.

6 Conclusion

This paper presented how models@runtime enables an iterative process to foster the design of reasoning engines for self-adaptive systems operating under certainty. This process is broken down into four steps: selection of uncertainty axis, exploration of unforeseen environments through simulation, evaluation of reasoning engine, followed by its potential refinement. Models@runtime appeared to be the architectural pattern which facilitates such a coevolution of the reasoning engine and its environments. We reported how this process is used in the Diversify project to develop a self-repair mechanism for large scale cloud-based systems, inspired by principles defined in ecology about biological diversity.

This research effort is a first step toward an automated refinement process for self-adaptive systems. Our future work will consider the continuous design of distributed and collaborative reasoning engines.

References

1. Luciano Baresi, Liliana Pasquale, and Paola Spoletini. Fuzzy goals for requirements-driven adaptation. In *RE*, pages 125–134. IEEE Computer Society, 2010.
2. Kent Beck. *Test-driven development : by example*. Addison-Wesley, Boston, 2003.
3. Gordon S. Blair, Nelly Bencomo, and Robert B. France. Models@run.time. *IEEE Computer*, 42(10):22–27, 2009.
4. Y.-S. Chen, P.P. Chong, and M.Y. Tong. Mathematical and computer modelling of the pareto principle. *Mathematical and Computer Modelling*, 19(9):61 – 80, 1994.
5. Deshan Cooray, Sam Malek, Roshanak Roshandel, and David Kilgore. Resisting reliability degradation through proactive reconfiguration. In Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto, editors, *ASE*, pages 83–92. ACM, 2010.
6. Rogério de Lemos, Holger Giese, Hausi A. Müller, and Mary Shaw, editors. *Software Engineering for Self-Adaptive Systems II - International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*, volume 7475 of *Lecture Notes in Computer Science*. Springer, 2013.

7. Rogério de Lemos et al. Software engineering for self-adaptive systems: A second research roadmap. In de Lemos et al. [6], pages 1–32.
8. Ahmed M. Elkhodary, Naeem Esfahani, and Sam Malek. Fusion: a framework for engineering self-tuning self-adaptive software systems. In Gruia-Catalin Roman and Kevin J. Sullivan, editors, *SIGSOFT FSE*, pages 7–16. ACM, 2010.
9. Naeem Esfahani, Ehsan Kouroshfar, and Sam Malek. Taming uncertainty in self-adaptive software. In Tibor Gyimóthy and Andreas Zeller, editors, *SIGSOFT FSE*, pages 234–244. ACM, 2011.
10. Naeem Esfahani and Sam Malek. Uncertainty in self-adaptive software systems. In de Lemos et al. [6], pages 214–238.
11. Nicolas Ferry, Alessandro Rossini, Franck Chauvel, Brice Morin, and Arnor Solberg. Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems. In *CLOUD 2013: IEEE 6th International Conference on Cloud Computing*, pages 887–894. IEEE Computer Society, 2013.
12. Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.
13. Mass Soldal Lund, Bjornar Solhaug, and Ketil Stolen. *Model-driven risk analysis: the CORAS approach*. Springer, 2011.
14. Anne E Magurran and Anne E Magurran. *Ecological diversity and its measurement*, volume 179. Princeton university press Princeton, 1988.
15. E. Mainsah. Autonomic computing: the next era of computing. *Electronics Communication Engineering Journal*, 14(1):2–3, 2002.
16. Robert Cecil Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.
17. Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing. Special Publication 800-145, National Institute of Standards and Technology, September 2001.
18. Brice Morin, Olivier Barais, Grégory Nain, and Jean-Marc Jézéquel. Taming Dynamically Adaptive Systems with Models and Aspects. In *31st International Conference on Software Engineering (ICSE'09)*, Vancouver, Canada, May 2009.
19. Sébastien Mosser, Franck Fleurey, Brice Morin, Franck Chauvel, Arnor Solberg, and Iokanaan Goutier. SENSAPP as a Reference Platform to Support Cloud Experiments: From the Internet of Things to the Internet of Services. In *Management of Resources and Services in Cloud and Sky Computing (MICAS)*, Timisoara, September 2012. IEEE.
20. Michael Oberguggenberger. The mathematics of uncertainty: models, methods and interpretations. In Wolfgang Fellin, Heimo Lessmann, Michael Oberguggenberger, and Robert Vieider, editors, *Analyzing Uncertainty in Civil Engineering*, pages 51–72. Springer Berlin Heidelberg, 2005.
21. Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *TAAS*, 4(2), 2009.
22. Matej Črepinšek, Shih-Hsi Liu, and Marjan Mernik. Exploration and exploitation in evolutionary algorithms: A survey. *ACM Comput. Surv.*, 45(3):35:1–35:33, July 2013.
23. Jon Whittle, Peter Sawyer, Nelly Bencomo, Betty H. C. Cheng, and Jean-Michel Bruel. Relax: a language to address uncertainty in self-adaptive systems requirement. *Requir. Eng.*, 15(2):177–196, 2010.