Walter Cazzola. Cogito, Ergo Muto! In Danny Weyns, Sam Malek, Rogério de Lemos, and Jesper Andersson, editors, Proceedings of the Workshop on Self-Organizing Architecture (SOAR'09)*, pages 1–7, Cambridge, United Kingdom, September 2009.*

Cogito, Ergo Muto!

(Invited Paper)

Walter Cazzola
Department of Informatics and Communication
Università degli studi di Milano
cazzola@dico.unimi.it

***Abstract*—No system escapes from the need of evolving either to fix bugs, to be reconfigured or to add new features. To evolve becomes particularly problematic when the system to evolve can not be stopped.**

Traditionally the evolution of a continuously running system is tackled on by calculating all the possible evolutions in advance and hardwiring them in the application itself. This approach gives origin to the code pollution phenomenon where the code of the application is polluted by code that could never be applied. The approach has the following defects: i) code bloating, ii) it is impossible to forecast any possible change and iii) the code becomes hard to read and maintain.

Computational reflection by definition allows an application to introspect and intercede on its own structure and behavior endowing, therefore, a reflective application with (potentially) the ability of self-evolving. Furthermore, to deal with the evolution as a nonfunctional concerns, i.e., that can be separated from the current implementation of the application, can limit the code pollution phenomenon.

To bring the design information (model and/or architecture) at run-time provides the application with a basic knowledge about itself to reflect on when a change is necessary and on how to deploy it. The availability of such a knowledge at run-time frees the designer from forecasting and coding all the possible evolutions in favor of a sort of evolutionary engine that, to some extent, can evaluate which countermove to apply.

In this contribution, the author will explore the role of reflection and of the design information in the development of self-evolving applications. Moreover, the author will sketch a basic reflective architecture to support dynamic self-evolution and he will analyze the adherence of the existing frameworks to such an architecture.

***Index Terms*—reflection, software evolution, self-adaptation**

I. Introduction

All software systems are subject to evolution, they evolve over time as new requirements emerge, or adaption and extensions are necessary. Studies pointed out that up to 80% [35] of the system lifetime will be spent on maintenance and evolution activities. A program that is useful in a real-world environment necessarily must change or become progressively less useful in that environment [33]. The continuously running systems do not escape this law.

We could state that a well-planned evolution should pass through the evolution of system's design information and then through the propagation of such changes to the implementation. This approach should be the most natural and intuitive to use (because it adopts the same mechanisms adopted during the development phase) and it should produce the best results (because each evolutionary step is planned and documented before its application) and the general quality of the code will not decline (as stated by the 7th law of software evolution [34]).

Unfortunately, this approach takes more time than to directly modify the code itself and, in principle, needs to be planned off-line. Even though its benefits, it is in contrast with the urgency often required by the change (e.g., to fix a bug in a critical system or to enhance an application with rough competitors) and badly fits with the unstoppable characteristic of the continuously running systems (design information is not available at run-time).

Normally, the evolution of critical, distributed and/or continuously running systems is emulated by directly enriching the original design information (and consequently code) with aspects concerning possible evolutions. This approach has several drawbacks:

- all possible evolutions are not always foreseeable *a priori*;
- system's design information and code are *polluted* by details related to the evolutionary design;
- the evolution is not really modeled; it is specified as a part of the behavior of the whole system, rather than as an extension that *could* be used in different contexts;
- code and model pollution hinders application maintenance and reduces possibility of reuse.

Clearly, this cannot be the ultimate solution to the problem of promptly evolving a system without interruptions in the service provision. Rather, *the promptness of action could be granted if the system itself should be able to plan and carry out its own evolution*.

Software evolution is an aspect orthogonal to (current) system behavior that crosscuts both application code and design; hence it should be subject to separation of concerns [29]. Separating evolution from the rest of a system is worthwhile, because evolution is made independent of the evolving system and the abovementioned problems are overcame. Design information will not be polluted by non pertinent details and will exclusively represent current system functionality without patches. This leads to simpler and cleaner implementation that can be analyzed without discriminating between what is and what could be the application's structure and behavior.

Reflection [36] is one of the mechanisms that easily permits to separate crosscutting concerns and to get self-aware applications. Reflective systems have the capability to reason about and act on their own behavior and structure so that could be able to decide how to evolve and apply the necessary steps and face their own evolution.

In the rest of the paper we will explore how reflection can cope with design information to develop a self adaptable system that can operate without human interference. In Sect. II we give an overview of the reflective architecture of a self-evolving system; in Sect. III we will explore the role of design information in the generic reflective architecture and the impact that different kinds of design information have on the approach to self-evolution; whereas in Sect. IV we explore how to design self-evolving systems. Finally in Sect. V we will have some discussions on the topic and draw our conclusions.

II. Evolution and Reflection

Reflection is defined as the activity, both *introspection* and *intercession*, performed by an agent when doing computations about itself [36]. A reflective system is layered in two or more levels (base-, meta-, meta-meta-level and so on) constituting a *reflective tower*; each layer is unaware of the above one(s). Base-level entities perform computations on the application domain entities whereas entities on the meta-levels perform computations on the entities residing on the lower levels. Computational flow passes from a lower level (e.g., the base-level) to the above level (e.g., the meta-level) by intercepting some events and specific computations (*shift-up action*) and backs when meta-computation has finished (*shift-down action*). All meta-computations are carried out on a representative of lower-level(s), called *reification*, that is kept *causally connected* to the original level (see details in [10]).

A. Evolution as a Crosscutting Concern.

Since the beginning, we are used to think the reflection as the perfect mechanism to separate nonfunctional and crosscutting concerns — i.e., concerns that do not contribute to the application's main functionality and whose implementation is tangled with such an implementation — from the rest of the application [29]. This is particularly true when we are speaking about separating a clearly defined feature whose implementation can be easily identified as in the case of logging and authentication but the truthfulness of such a statement is arguable when these characteristics are not present at design-time or are difficult to grasp as when the feature implementation is scattered in distributed components whose code could be inaccessible or when it should be part of a continuously running system.

Often software evolution implies to reengineering the design and the code of software systems. In any case, when applying such modifications to existing software, the change often cannot be localized but involves several components; this leads to *tangled* and/or *scattered* code. Traditionally the evolution of a continuously running system is tackled on by calculating all the possible evolutions in advance and hardwiring them in the application itself. This approach gives origin to the *code pollution phenomenon* where the code of the application is polluted by code that could never be applied. Moreover, the approach has the following defects: i) code bloating, ii) it is impossible to forecast any possible change and iii) the code becomes hard to read and maintain.

It should be fairly evident that the capability to evolve and any particular evolution — especially in the case of the evolution of continuously running systems — are clearly nonfunctional and crosscutting concerns [13], [39], [41]. The code necessary to evolve an application cannot contribute to the application's basic functionality until the urgency for the evolution comes up. Moreover the evolution of a continuously running system hardly can be prepared in advance but need to be decided according to the (sudden and unexpected) risen necessity and the decision must be immediate due to the urgency often required by the adaptation, e.g., to reconfigure a urban traffic control system to face a traffic jam [14], [48]. Due to the previous considerations it is still more natural to think about the evolution as something to add when necessary and to be kept separate from the core of the application.

B. Reflecting on the Design Information

To automate the evolutionary process the application must be able to work out how to evolve from its current state. To this respect, it needs to reflectively introspect into its state and structure and know how to decide which is the correct strategy to solve the risen issue; this means to select one among several prearranged strategies and adapt it to fit the problem or (more difficult) define a new strategy to fix it. Reflection helps to introspect and to apply the strategy but computational reflection is grounded on the application's code so it provides a local view of the application limited to the code and, as previously stated, a correct evolutionary strategy can be decided only if the whole architecture and behavior of the application is known and taken in consideration: evolution impact not only affect the part of the application responsible of the issue but several other not necessarily correlated parts (*ripple effect* [4], [5]), e.g., you cannot close a road for maintenance without considering the traffic flow through such a road and without changing the direction of the incident roads to avoid further disruptions.

Design information, when consistent with the implementation, provides an accurate snapshot of the application's structure and behavior. Moreover, the application's design information abstracts from the code — i.e., a very local and concrete view — giving a *global view* of the whole system — i.e., it gives a glimpse at the application that immediately shows which part of the application could be affected by a change into another application's part and which part is in charge of which feature *without* looking at the code details. As stated in [15], [16], this summarizes the overall knowledge about the application in a handy form that is suited to plan the evolution.

Design information as a base of knowledge for planning the evolution is not a *panacea* for automating the evolutionary process but it can be considered as a step towards a self-evolving software system. It is necessary to extend the reflection to deal with the design information as application domain, similarly to [18], [19] (software architecture) and [13] (UML models) and to causally connect the design information to the code to avoid the *design/implementation gap* [17], [49].

In this way, reflective introspection and intercession apply to the application's design information and the relationship of causally connection will realize the evolution when necessary; the shift-down operation will take care of coordinating the changes and to avoid inconsistencies.

C. A Generic Reflective Architecture for SW Evolution

Summarizing the previous considerations, a system that can self-evolve (reconfiguring, extending and maintaining) should have a reflective structure. The self-evolving application runs in the base-level whereas the, so called, *evolutionary engine* runs in the meta-level.

The reflective architecture is completed by a pool of *reflectors*. These components take care of realizing the reflective behavior of the whole framework realizing the shift-up and -down operations and keeping the causal connection between the application in the base-level and its representative in the meta-level (the reification). These components also deal with the particular application domain keeping a binding among the code and the design information.

The evolutionary engine is in charge of evolving the application running in the base-level when happens a specified event. The engine works on a representative of the application (called *reification* and built by the reflectors) and the engine has two kinds of cooperating components: *planners* and *actuators*. The multiplicity of the evolutionary engine's components depends on the granularity of the evolutionary actions: each aspect of the system could be handled by a different component.

An external event triggers the planners which use the reification and an *evolutionary base of knowledge* to plan the strategy to face the risen situation. The evolutionary base of knowledge contains strategies, i.e., predetermined solutions to situations that could happens; the base of knowledge can be fixed, augmentable with the strategies derived by the planner and/or dynamically enrichable by an human operator. The actuators cooperate with the planners to render effective the planned evolution: they apply the strategy decided by the planner to the reification validating the consistency of the result before asking the reflectors for reflecting the changes on the base-level.

Fig. 1 depicts the described reflective architecture. A preliminary version of such architecture has been introduced in [11] through a pattern family describing its behavior.

III. Reflective Architecture for Self-Reconfiguration and -Adaptation

The logic architecture introduced in the previous section adopts reflection and design information to provide the application (running in the base-level) with the capability of planning and actuating its own evolution.

The introduced architecture is generic and can be used to deal with evolution in general. By the way there are several kinds of evolution [38] each with specific requirements so to correctly face their variability the abstraction provided by the design information must have different granularities case by case.

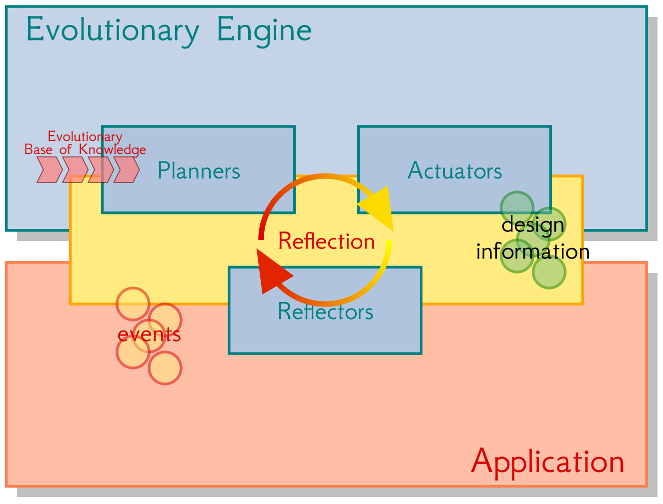

Figure 1. Architecture for a Self-Evolving Application

For example, reconfiguration takes place at component level, the system's architecture changes (added/removed or differently connected components) to face the requested reconfiguration. To plan its own reconfiguration, the application must reason and act *in-the-large*[1] i.e., on how the components interact rather than on how they work.

A. Self-Reconfiguration: Reflecting on the Architecture

Several definitions for software architecture are available[2] but the one that better fits our needs is:

> Software Architectures deals with the design and implementation of the high-level structure of the software. It is the result of assembling a certain number of architectural elements in some well-chosen forms to satisfy the major functionality and performance requirements such as scalability and availability. Software architecture deals with abstraction, with decomposition and composition, with style and aesthetics [32].

Software architectures provides a global view of how application's components fit together neglecting to detail what every component does, that is, they describe the application *in-the-large* rather than *in-the-small* [25].

The software architecture's higher abstraction explicits a link to components and connectors as a whole rather than to their implementation; this helps to plan the application's reconfiguration [37], [40] but hinders a deeper adaptation involving changes to the code.

In the years a plethora of approaches that support self-reconfiguration through software architecture's manipulation

[1] After [25] we use the term *in-the-large* in contrast to *in-the-small* to put in evidence the necessary abstraction level.

[2] Gives a look at http://www.sei.cmu.edu/architecture/published_definitions.html to get a grasp.

have been explored, e.g., architectural reflection [19], [20], [46], Rainbow [27], and K-Components Architecture [26].

Architectural Reflection [19], [20] exploits the application's software architecture as the application domain for reflective activities; all the reflective concepts have been moved from the code to the architectural domain. The software architecture is reified[3], decomposed into topology and strategy and manipulated, respectively, by two meta-level components, called *topologist* and *strategist*. These meta-level components plan and force the application's reconfiguration through the manipulation of the (reified) software architecture a specific actuator is used to reflect the change on the system. Comparing this architecture with the general one introduced in Sect. II-C it should be evident the similarity: topologist and strategist are two planners, each of them cooperates with their own actuator, and both planning and validation pass through the design information (that is, the software architecture) and through the reification, the application can be reconfigured.

The Rainbow framework [27] adopts an architecture-based approach, it provides reusable infrastructure together with mechanisms for specializing that infrastructure to the needs of specific systems. The Rainbow framework includes an application's architectural model in its run-time system. In particular, developers of self-reconfiguring capabilities use a system's software architectural model to monitor and reason about the system (to some extent a reflective system whose application domain is the application's software architecture). The Rainbow's control loop for self-reconfiguration passes through the following steps: i) a *model manager* handles and provides access to the application's architectural model; ii) a *constraint evaluator* checks the model periodically and triggers adaptation if a constraint violation occurs; then iii) an *adaptation engine* determines the course of action and carries out the necessary adaptation; finally iv) an *adaptation executor* triggers the *effectors* to reflect the changes on the application. The similarities with the proposed infrastructure are still evident: the model manager together with the adaptation executor and the effectors play the role of reflectors whereas the adaptation engine and the constraint evaluator play the role of the evolutionary engine.

The K-Components Framework [26] reifies the application's software architecture as a configuration graph (i.e., a graph whose nodes and edges are respectively components and connectors) and the reconfiguration is yielded through reconfiguration protocol used to rewrite the configuration graph; both the evolution planning and validation are made on the reification by a *configuration manager* which also reflect the changes on the application. In the K-Components framework planners, actuators and reflectors collapse in a single meta-entity, the configuration manager, that manage the configuration graph evolution but the evolutionary control loop is still the same: reification, event-checking, planning, validation and reflection.

[3]This process permits to render the software architecture explicit and observable and the system controllable through its architecture [20].

Other self-reconfiguring approaches that reflectively exploits software architecture in their control loop and confirm the general architecture proposed in Sect. II-C are: ArchStudio [43], TranSAT [1], PRISMA [22] and Dellarocas *et al.* [24].

B. Self-Adaptation: Reflecting on the Model

Reconfiguration works at component level by adding, removing or substituting components and/or rearranging their connections; so the planning of the reconfiguration can pass through the high-level representation provided by the software architecture. In general evolution also includes the possibility of fixing bugs and extending/adapting already implemented code; to this respect it is impossible to plan this kind of evolutions on the application's software architecture since you have to deal with how the component works and how it is implemented. In this case, the design information should represent the application *in-the-small* as the UML models [6] do.

Self-Adaptation implies to adapt how the components work not just how they interact with other components (problem that can easily be faced through the manipulation of configuration files) that means to alter their implementation. This poses some new challenges: i) how to modify the code during the execution (several tools, as Java Hot Spot and Javeleon [28], help with this issue) ii) how to associate the design information to the running code (normally models and code are statically coupled and the generation of the new code is model-driven). These additional challenges rendered less appealing to perform dynamic self-adaptation and few attempts can be found in the literature; most of them exploit the model driven engineering [31] methodology where models have a proactive role and are used to generate the application itself, e.g., WEAVR [23] and the Adaptive Object Model Architecture [50]; in the next we will neglect to analyze these approaches since they have a different perspective: the application is generate from the design information rather than simply used as a source of information to drive the adaptation of existing code.

RAMSES (Reflective and Adaptive Middleware for Software Evolution of Systems) [12], [13] is a reflective framework that provides an application with the capability of dynamically self-adapting. The framework has two logic levels: the application prone to be adapted runs in the base-level whereas in the meta-level a couple of meta-objects (the *evolutionary* and *consistency checker* meta-objects) take care of planning and validating the application's evolution. The work of both such meta-objects is supported by dedicated engines that applies validation and evolutionary rules (ruby scripts) to the application's reification when triggered by the meta-objects. The base-level models' are reified in the meta-level as XMI [42] schema and the changes are reflected back on the application through techniques of code instrumentation [44], [45]. Even without going deep in the architecture's description it is evident the similarity with the general approach examined in Sect II-C the only difference is the complexity of the reflector that needs to fill the gap between code and model.

Chisel [30] is an open framework for dynamic adaptation of services using reflection in a policy-driven, context-aware manner. The system is based on decomposing the particular aspects of a service object that do not provide its core functionality into multiple possible behaviors. As the execution environment, user context and application context change, the service object will be adapted to use a different behaviors, driven by a human-readable declarative adaptation policy script. The Chisel framework has a *meta-level adaptation manager* that coordinates the whole adaptation process by monitoring the application's execution environment, by planning the adaptation driven by a set of *adaptation policies* and by extracting (reifying) the *meta-types* from the running application and reflecting the adaptation. In the Chisel framework the meta-level adaptation manager plays the role of adaptation engine (including planners, actuators and reflectors), the adaptation policies are the evolutionary strategies and the meta-types plays the role of design information.

Genie [2], [3] is a reflective framework to support dynamic adaptation of a software system through its design information. Even if it adopts software architecture to represent the application in the meta-level it also enables a quite limited self-adaptation through the self-generation of models describing the application's state transitions that can be used to change the application's behavior. Genie framework monitors the application's context, reifies the application's software architecture, a specific component plans the adaptation strategy as a delta from the current design and the design that should be and finally passes the generated *reconfiguration script* to a specific component (named *configurator*) that changes the application. The script generator plays the role of planner whereas the configurator is the actuator/reflector of the architecture presented in Sect. II-C.

IV. Modeling a Self-Evolving System

An application able to evolve itself or at least this capability cannot be modeled with traditional design techniques, such as Petri nets, UML, and so on. To some extent this is reasonable: how can I model something that I cannot imagine? Why should I model something doomed to change or that could be never applied?

Unfortunately, this issue nullifies all the benefits we got from designing the application: the adapted application cannot be tested; the quality of service cannot be granted and, last but not least the capability of self-evolving is not documented and cannot be considered during the evolution, i.e., the set of evolutionary strategies cannot be affected by the application's adaptation and are inclined to rapidly become obsolete and therefore useless for planning the successive evolution (we refer to this problem as the *fragility problem of the evolutionary strategies*).

A few attempts to model the self-evolvable capability have been done and the reflective Petri nets [7]–[9] seems to be the most promising. A reflective Petri net is a high-level Petri net extension that permits to model a self-evolving application. The application's current model defines the base-level model

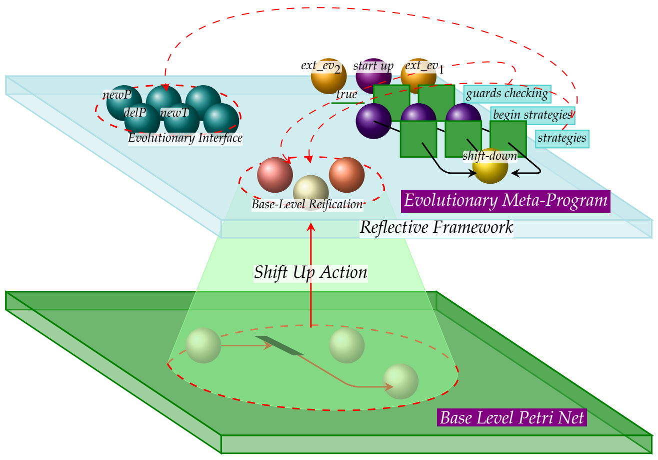

Figure 2. Reflective Petri Net Model

(a classic place/transition Petri net) whereas the evolutionary strategies applicable to the application define the meta-level model (each strategy is modeled by a Petri net). The base-level model is reified as a marking in the meta-level and a set of primitive operations (such as add a place or remove a transition) can be used to evolve the reification. The semantics of the reflective Petri nets will take care of keeping the base-level model and its reification consistent over the time.

V. Conclusions

Nowadays, with the increasing diffusion of the Internet, more and more applications need to run continuously and provide their services without interruptions, e.g., air and urban traffic control systems, nuclear plant control systems, electronic shops and so on. To stop such a kind of systems could be mission critical but also a disservice to the customers and a potential loss of money.

Of course continuously running applications can not be bug free or equipped with all the desired features over the long period, so they need to evolve exactly as the other application but with the constraint that they have to evolve *during* their execution, without disservice and possibly promptly and autonomously.

Such considerations drive forth to the need of self-evolving systems, that is, systems that can reasons about themselves and, in case, decide how to change their own behavior and structure.

In the last few years a plethora of frameworks supporting self-evolution have been developed [2], [12], [18], [22], [24], [26], [27], [30]. Even if apparently all these frameworks differ, they have several in common: they all exploit reflection and have a decisional engine that, to some extent, deal with the self-evolution.

The contribution of this paper is to sort out this plethora of proposals and to extrude a common architecture for the self-evolving approaches. From a deeper analysis of some of the existing frameworks we pointed out that they all share a common control loop (event monitoring, reification, evolution planning and validation and finally adaptation) that can be logically summarized by Fig. 1. The frameworks differ on the realization and on what is used as a knowledge base to plan the evolution. Salehie and Tahvildari [47] in their overview confirm our conclusions giving a similar but less detailed abstraction for the self-adaptive architecture.

A second consideration we have pointed out regards the granularity used for the application's reification (that is, the kind of design information used) directly affects the kind of self-evolution that the self-evolving application can carries out. For example, software architecture well fits the self-reconfiguration but is inadequate for self-adaptation.

In the future we would like to widen our analysis including aspect-oriented approaches as well. Moreover we want to compare our approach to self-evolution based on reflection and design information with those approaches that directly evolve the code through deltas application such as [21].

Acknowledgment

The author would like to thank Sonia Pini and Ahmed Ghoneim that helped him in exploring the role of the design information in the evolution/co-evolution of a software system. Moreover, he wishes to thank Francesco Tisato and Massimo Ancona that introduced him to the problems of reflection and software evolution.

References

[1] O. Barais, E. Cariou, L. Duchien, N. Pessemier, and L. Seinturier, "TranSAT: A Framework for the specification of Software Architecture Evolution," in *Proceedings of the 1st ECOOP Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT'04)*, Oslo, Norway, Jun. 2004, pp. 31–38.

[2] N. Bencomo, "Supporting the Modelling and Generation of Reflective Middleware Families and Applications using Dynamic Variability," PhD Thesis, Computing Department, Lancaster University, Lancaster, United Kingdom, Mar. 2008.

[3] ——, "On the Use of Software Models during Software Execution," in *Proceedings of the ICSE Workshop on Modeling in Software Engineering (MISE'09)*, Vancouver, Canada, May 2009.

[4] K. H. Bennett and V. T. Rajlich, "Software Maintenance and Evolution: a Roadmap," in *The Future of Software Engineering*, A. Finkelstein, Ed. ACM Press, 2000, pp. 75–87.

[5] S. Black, "The Role of Ripple Effect in Software Evolution," in *Software Evolution and Feedback*, N. H. Madhavji, J. C. Fernández-Ramil, and D. E. Perry, Eds. John Wiley & Sons, Ltd, Jun. 2006, ch. 12, pp. 249–268.

[6] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, 3rd ed., ser. Object Technology Series. Reading, Massachusetts: Addison-Wesley, Feb. 1999.

[7] L. Capra and W. Cazzola, "Self-Evolving Petri Nets," *Journal of Universal Computer Science*, vol. 13, no. 13, pp. 2002–2034, Dec. 2007.

[8] ——, "An Introduction to Reflective Petri Nets," in *Handbook of Research on Discrete Event Simulation Environments: Technologies and Applications*, E. M. O. Abu-Taieh and A. A. El Sheikh, Eds. IGI Global, Nov. 2009, ch. 9, pp. 191–217.

[9] ——, "Trying out Reflective Petri Nets on a Dynamic Workflow Case," in *Handbook of Research on Discrete Event Simulation Environments: Technologies and Applications*, E. M. O. Abu-Taieh and A. A. El Sheikh, Eds. IGI Global, Nov. 2009, ch. 10, pp. 218–233.

[10] W. Cazzola, "Evaluation of Object-Oriented Reflective Models," in *Proceedings of ECOOP Workshop on Reflective Object-Oriented Programming and Systems (EWROOPS'98)*, ser. in 12th European Conference on Object-Oriented Programming (ECOOP'98), Brussels, Belgium, on 20th-24th Jul. 1998, extended Abstract also published on ECOOP'98 Workshop Readers, S. Demeyer and J. Bosch editors, LNCS 1543, ISBN 3-540-65460-7 pages 386-387.

[11] W. Cazzola, J. O. Coplien, A. Ghoneim, and G. Saake, "Framework Patterns for the Evolution of Nonstoppable Software Systems," in *Proceedings of the 1st Nordic Conference on Pattern Languages of Programs (VikingPLoP'02)*, P. Hruby and K. E. Søresen, Eds. Højstrupgård, Helsingør, Denmark: Microsoft Business Solutions, on 20th-22nd of Sep. 2002, pp. 35–54.

[12] W. Cazzola, A. Ghoneim, and G. Saake, "Reflective Analysis and Design for Adapting Object Run-time Behavior," in *Proceedings of the 8th International Conference on Object-Oriented Information Systems (OOIS'02)*, ser. Lecture Notes in Computer Science 2425, Z. Bellahsène, D. Patel, and C. Rolland, Eds. Montpellier, France: Springer-Verlag, on 2nd-5th of Sep. 2002, pp. 242–254, iSBN: 3-540-44087-9.

[13] ——, "Software Evolution through Dynamic Adaptation of Its OO Design," in *Objects, Agents and Features: Structuring Mechanisms for Contemporary Software*, ser. Lecture Notes in Computer Science 2975, H.-D. Ehrich, J.-J. Meyer, and M. D. Ryan, Eds. Heidelberg, Germany: Springer-Verlag, Jul. 2004, pp. 69–84.

[14] ——, "System Evolution through Design Information Evolution: a Case Study," in *Proceedings of the 13th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE 2004)*, W. Dosch and N. Debnath, Eds. Nice, France: ISCA, on 1st-3rd of Jul. 2004, pp. 145–150.

[15] W. Cazzola, S. Pini, and M. Ancona, "AOP for Software Evolution: A Design Oriented Approach," in *Proceedings of the 10th Annual ACM Symposium on Applied Computing (SAC'05)*. Santa Fe, New Mexico, USA: ACM Press, on 13th-17th of Mar. 2005, pp. 1356–1360.

[16] ——, "The Role of Design Information in Software Evolution," in *Proceedings of the 2nd ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'05)*, ser. in 19th European Conference on Object-Oriented Programming (ECOOP'05), W. Cazzola, S. Chiba, G. Saake, and T. Tourwé, Eds., Glasgow, Scotland, on 25th Jul. 2005, pp. 59–70.

[17] W. Cazzola, S. Pini, A. Ghoneim, and G. Saake, "Co-Evolving Application Code and Design Models by Exploiting Meta-Data," in *Proceedings of the 12th Annual ACM Symposium on Applied Computing (SAC'07)*. Seoul, South Korea: ACM Press, on 11th-15th of Mar. 2007, pp. 1275–1279.

[18] W. Cazzola, A. Savigni, A. Sosio, and F. Tisato, "Architectural Reflection: Bridging the Gap Between a Running System and its Architectural Specification," in *Proceedings of 6th Reengineering Forum (REF'98)*. Firenze, Italia: IEEE, on 8th-11th Mar. 1998, pp. 12–1–12–6.

[19] ——, "Architectural Reflection: Concepts, Design, and Evaluation," DSI, Università degli Studi di Milano, Technical Report RI-DSI 234-99, May 1999, available at http://homes.dico.unimi.it/~cazzola/cazzolawbib-by-year.html.

[20] ——, "Rule-Based Strategic Reflection: Observing and Modifying Behaviour at the Architectural Level," in *Proceedings of 14th IEEE International Conference on Automated Software Engineering (ASE'99)*, Cocoa Beach, Florida, USA, on 12th-15th Oct. 1999, pp. 263–266.

[21] S. Cech Previtali and T. Gross, "Dynamic Updating of Software Systems Based on Aspects," in *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM'06)*, Philadelphia, USA, Sep. 2006, pp. 83–92.

[22] C. Costa-Soria, D. Hervás-Muñoz, J. Pérez Benedí, and J. A. Carsí Cubel, "A Reflective Approach for Supporting the Dynamic Evolution of Component Types," in *Proceedings of the 14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'09)*, Potsdam, Germany, Jun. 2009, pp. 301–310.

[23] T. Cottenier, A. van den Berg, and T. Elrad, "Motorola WEAVR: Aspect Orientation and Model-Driven Engineering," *Journal of Object Technology*, vol. 6, no. 7, pp. 51–88, Aug. 2007.

[24] C. Dellarocas, M. Klein, and H. Shrobe, "An Architecture for Constructing Self-Evolving Software Systems," in *Proceedings of the 3rd International Workshop on Software Architecture* , Orlando, FL, USA, Nov. 1998, pp. 29–32.

[25] F. DeRemer and H. H. Kron, "Programming-in-the-large versus

Programming-in-the-small," *"IEEE Trans. Software Engineering"*, vol. SE-2, pp. 80–86, Jun. 1976.

[26] J. Dowling and V. Cahill, "The K-Component Architecture Meta-Model for Self-Adaptive Software," in *Proceedings of 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection'2001)*, ser. LNCS 2192, A. Yonezawa and S. Matsuoka, Eds. Kyoto, Japan: Springer-Verlag, Sep. 2001, pp. 81–88.

[27] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-Based Self Adaptation with Reusable Infrastructure," *IEEE Computer*, vol. 37, no. 10, pp. 46–54, Oct. 2004.

[28] A. R. Gregersen and B. N. Jørgensen, "Dynamic Update of Java Applications — Balancing Change Flexibility vs Programming Transparency," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 21, no. 2, pp. 81–112, Mar./Apr. 2009.

[29] W. Hürsch and C. Videira Lopes, "Separation of Concerns," Northeastern University, Boston, Tech. Rep. NU-CCS-95-03, Feb. 1995.

[30] J. Keeney and V. Cahill, "Chisel: A Policy-Driven, Context-Aware, Dynamic Adaptation Framework," in *Proceedings of the 4th International Workshop on Policies for Distributed Systems and Networks (POLICY 2003)*, Como, Italy, Jun. 2003, pp. 3–14.

[31] S. Kent, "Model Driven Engineering," in *Proceedings of the 3rd International Conference on Integrated Formal Methods (IFM'02)*, ser. LNCS 2335, M. J. Butler, L. Petre, and K. Sere, Eds. Turku, Finland: Springer, May 2002, pp. 286–298.

[32] P. Kruchten, "The 4+1 View Model of Architecture," *IEEE Software*, vol. 12, no. 6, pp. 61–70, 1995.

[33] M. M. Lehman, "Programs, Life Cycles, and Laws of Software Evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, Sep. 1980, special Issue on Software Engineering.

[34] ——, "Laws of Software Evolution Revisited," in *Proceedings of the 5th European Workshop on Software Process Technology (EWSPT'96)*, ser. LNCS 1149, C. Montangero, Ed. Nancy, France: Springer, Oct. 1996, pp. 108–124.

[35] M. M. Lehman, J. C. Fernández-Ramil, and G. Kahen, "A Paradigm for the Behavioural Modelling of Software Processes using System Dynamics," Imperial College, Department of Computing, London, United Kingdom, Technical Report 2001/8, Sep. 2001.

[36] P. Maes, "Concepts and Experiments in Computational Reflection," in *Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, ser. Sigplan Notices, N. K. Meyrowitz, Ed., vol. 22. Orlando, Florida, USA: ACM, Oct. 1987, pp. 147–156.

[37] J. Magee and J. Kramer, "Self-Organising Software Architecture," in *Proceedings of 2nd International Software Architecture Workshop (ISAW'96)*, A. L. Wolf, A. Finkelstein, G. Spanoudakis, and L. Vidal, Eds. San Francisco, CA, USA: ACM, Oct. 1996, pp. 35–38.

[38] T. Mens, J. Buckley, M. Zenger, and A. Rashid, "Towards a Taxonomy of Software Evolution," in *Proceedings of the 2nd International Workshop on Unanticipated Software Evolution (USE 2003)*, Warsaw, Poland, Apr. 2003.

[39] T. Mens and M. Wermelinger, "Separation of Concerns for Software Evolution," *Journal of Maintenance and Evolution*, vol. 14, no. 5, pp. 311–315, 2002.

[40] G. C. Murphy, "Architecture for Evolution," in *Proceedings of 2nd International Software Architecture Workshop (ISAW'96)*, A. L. Wolf, A. Finkelstein, G. Spanoudakis, and L. Vidal, Eds. San Francisco, CA, USA: ACM, Oct. 1996, pp. 83–86.

[41] O. Nierstrasz and F. Achermann, "Supporting Compositional Styles for Software Evolution," in *Proceedings of the International Symposium on Principles of Software Evolution (ISPSE 2000)*, Kanazawa, Japan, Nov. 2000, pp. 11–19.

[42] OMG, "OMG-XML Metadata Interchange (XMI) Specification, v1.2," OMG Modeling and Metadata Specifications available at http://www.omg.org, Jan. 2002.

[43] P. Oreizy, N. Medvidovic, and R. N. Taylor, "Architecture-Based Runtime Software Evolution," in *Proceedings of the International Conference on Software Engineering (ICSE'98)*, Kyoto, Japan, Apr. 1998, pp. 177–186.

[44] M. Pukall, C. Kästner, and G. Saake, "Towards Unanticipated Runtime Adaptation of Java Applications," in *Proceedings of the 15th Asia-Pacific Software Engineering Conference (APSEC'08)*. Bejing, China: IEEE Computer Society, Dec. 2008, pp. 85–92.

[45] M. Pukall, N. Siegmund, and W. Cazzola, "Feature-Oriented Runtime Adaptation," in *Proceedings of ESEC/FSE Workshop on Software INTegration and Evolution @ Runtime (SINTER'09)*. Amsterdam, The Netherlands: ACM, on 25th of Aug. 2009, pp. 33–36.

[46] S. Rank, "Architectural Reflection for Software Evolution," in *Proceedings of ECOOP'2005 Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'05)*, W. Cazzola, S. Chiba, G. Saake, and T. Tourwé, Eds., Glasgow, Scotland, Jul. 2005, pp. 53–58.

[47] M. Salehie and L. Tahvildari, "Self-Adaptive Software: Landscape and Research Challenges," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 4, no. 2, pp. 14:1–14:42, May 2009.

[48] A. Savigni and F. Tisato, "Designing Traffic Control Systems. A Software Engineering Perspective," in *Proceedings of Rome Jubilee 2000 Conference (Workshop on the International Foundation for Production Research (IFPR) on Management of Industrial Logistic Systems – 8th Meeting of the Euro Working Group Transportation - EWGT)*, Roma, Italy, Sep. 2000.

[49] N. Ubayashi, H. Akatoki, and J. Nomura, "Pointcut-based Architectural Interface for Bridging a Gap between Design and Implementation," in *Proceedings of the 6th ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'09)*, M. Oriol, W. Cazzola, S. Chiba, and G. Saake, Eds., Genoa, Italy, Jul. 2009.

[50] J. W. Yoder and R. E. Johnson, "The Adaptive Object-Model Architectural Style," in *Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture (WICSA'02)*, ser. IFIP Conference Proceedings, J. Bosch, W. M. Gentleman, C. Hofmeister, and J. Kuusela, Eds., vol. 224. Kluwer, Aug. 2002, pp. 3–27.