Proc. of the International Workshop on Self-Managed Systems (WOSS'04), Newport Beach, California, USA, October/November 2004.

A Survey of Self-Management in Dynamic Software Architecture Specifications *

Jeremy S. Bradbury[a], James R. Cordy[a†], Juergen Dingel[a], Michel Wermelinger[b‡]
[a]School of Computing, Queen's University, Kingston, Ontario, Canada
[b]Departamento de Informática, Universidade Nova de Lisboa, Caparica, Portugal
{bradbury, cordy, dingel}@cs.queensu.ca, mw@di.fct.unl.pt

ABSTRACT

As dynamic software architecture use becomes more widespread, a variety of formal specification languages have been developed to gain a better understanding of the foundations of this type of software evolutionary change. In this paper we survey 14 formal specification approaches based on graphs, process algebras, logic, and other formalisms. Our survey will evaluate the ability of each approach to specify self-managing systems as well as the ability to address issues regarding expressiveness and scalability. Based on the results of our survey we will provide recommendations on future directions for improving the specification of dynamic software architectures, specifically self-managed architectures.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications—*languages*; D.2.9 [**Software Engineering**]: Management—*software configuration management*; D.2.11 [**Software Engineering**]: Software Architectures

Keywords

dynamic software architecture, architectural formalism, dynamism, run-time evolution, specification, self-management

1. INTRODUCTION

Dynamic software architectures modify their architecture and enact the modifications during the system's execution [18]. This behavior is most commonly known as run-time evolution or dynamism. Self-managing architectures are a specific type of dynamic software architectures. We define a system that enacts architectural changes at run-time as having a *self-managing architecture* if the system not only implements the change internally but also initiates, selects, and assesses the change itself without the assistance of an external user. Programmed dynamism [10], self-organising architectures [16, 12], self-repairing systems [23], and self-adaptive software [21] are all examples of self-managing architectures. Programmed dynamism is an early type of dynamism in which a fixed change is conditionally triggered by the system. The other examples listed support more advanced notions of self-management in architectural reconfiguration.

Dynamic software architectures and specifically dynamic components have been identified as "challenging in terms of correctness, robustness, and efficiency" [24]. This is especially true for self-managing architecture since systems that are self-managed have to implement the initiation and selection of a change. Conversely, user-managed architectures usually exhibit ad-hoc change [10] in which the initiation and selection occur external to the software, thus simplifying the development.

Formal specification is one way to support the development of correct and robust dynamic software architectures. In this paper we present a survey of 14 dynamic software architecture specification approaches. Our goal is to focus on the ability of each approach to specify self-managing architectures. First, we determine if each specification approach supports our definition of a self-managing architecture. Second, we evaluate each approach with respect to the expressiveness of the approach in specifying different types of change and different levels of change from a fixed selection approach to an unconstrained approach. Third, we compare the scalability of the approaches to specify decentralized management schemes which are more likely in large-scale systems. After evaluating all of the specification approaches we use the results to make recommendations on how formal specification approaches for dynamic software systems in general, and self-managing in particular, can be improved.

Related work to our survey includes several papers that have surveyed Architecture Description Languages (ADLs) and provided broad comparisons [6, 18]. The survey in [6] compared ADLs on attributes related to scope of language, expressive power, tool maturity, and others. The survey in [18] compared ADLs in terms of their ability to model components, connectors and configuration as well as tool support for such things as analysis and refinement. Our work differs from previous work in that we consider only formal specification approaches and provide a narrower comparison, focusing on the ability of each approach to specify self-managing architectures. The previous approaches have not focused on self-managing architectures. In fact, only the survey in [18] even considers run-time evolution in its evaluation.

In Section 2 we will provide an overview of formal specification for dynamic software architecture including some details regarding the 14 specification approaches surveyed. In Section 3 we

†Current address: ITC-IRST, Trento, Italy.
‡Current address: Computing Department, The Open University, Milton Keynes, UK.
*This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

		Architectural Structure		Architectural Element Behavior		Architectural Reconfiguration
		Architectural Style	System Architecture	Components	Connectors	
Graph	Le Métayer approach	context-free graph grammar	graph (formally defined as a multiset)	nodes of a graph and a CSP like behavior specification	edges of a graph	graph rewriting rules with side conditions to refer to the status of public variables
	Hirsch et al. approach	context-free graph grammar	hypergraph	edges of a graph with CCS labels	nodes of a graph [point-to-point communication (white nodes) and broadcast communication (black nodes)]	graph rewriting rules
	Taentzer et al. approach	-	distributed graph (network graph)	local graph for each network graph node and local transformations between local graphs	edges of a graph	graph rewriting rules
	CommUnity	-	Categorical diagram	a program in CommUnity (a Unity-like language)	a star-shaped configuration of programs	category theory using double-pushout graph transformation approach
	CHAM	creation CHAM	-	molecule	links between two component molecules	evolution CHAM reaction rules
Process Algebra	Dynamic Wright	-	implicit graph representation (components and connectors are nodes)	ports (interface) + computation (behavior)	roles (interface) + glue (behavior)	CSP
	Darwin	-	implicit graph representation	programming language + component specification of comm. objects	support for simple bindings	π-calculus
	LEDA	-	implicit graph representation	interface specification, composition and attachment specification (if composite)	attachments at top level components	π-calculus
	PiLar	-	implicit graph representation	components with ports, instances of other components and constraints	support for simple bindings	CCS
Logic	Gerel	-	implicit graph representation	interface in Gerel language and behavior in a programming language	defined by bind operation in configuration components	first order logic
	Aguirre-Maibaum approach	-	implicit graph representation	class with attributes, actions and read variables	association consisting of participants and synchronization connections	first order logic, temporal logic
	ZCL	-	implicit graph representation (defined by set of state schemas in Z)	state schema in Z	connection between ports of components	operation schema in Z (predicate logic and set theory)
Other	C2SADEL	only supports the C2 arch. style	implicit graph representation (defined by Architecture Description Language (ADL))	element with top and bottom interface and behavior (defined by Interface Definition Language (IDL))	element with top and bottom ports and filtering mechanisms (defined by ADL)	Architecture Modification Language (AML)
	Rapide	-	implicit graph representation	types language for component interface (plus other sublanguages for behavior)	broadcast connection rule (||>) or pipe (=>)	where statement or execution architecture events

Table 1: Support for structure, behavior, and reconfiguration in formal specification approaches for dynamic software architectures

will evaluate the ability of each specification approach to support self-management and address expressiveness and scalability. The information presented in this section was gathered from published research papers and our own experience using each approach. Finally, we conclude and discuss future work in Section 4.

2. FORMAL SPECIFICATION

Formal approaches to dynamic software architectures involve the specification of the architectural structure of a system, the architectural reconfiguration of a system, and usually the behavior of a system. The formal approaches to specifying dynamic software architectures that we consider are divided into four categories: graph-based approaches, process algebra approaches, logic-based approaches, and other approaches.

Graph-Based Approaches. A natural way to specify software architectures and architectural styles is to use a graph grammar to represent the style and a graph to represent a specific system's architecture. Furthermore, a natural way to specify reconfiguration in a dynamic architecture is to use graph rewriting rules. We include the following graph-based approaches in our survey: the Le Métayer approach [19, 20], the Hirsh et al. approach [13], the Taentzer et al. approach [25], CommUnity approach [28, 29], and Chemical Abstract Machine (CHAM) approach [27].

Process Algebra Approaches. Process algebras are commonly used to study concurrent systems. Processes in the concurrent system are specified in an algebra and a calculus is used to verify the specification. A variety of process algebras exist including the Calculus of Communicating Systems (CCS), Communicating Sequential Processes (CSP), and the π-calculus. We consider four process algebra approaches in this paper: Dynamic Wright [2], Dar-

win [15], LEDA [5], and PiLar [7].

Logic-Based Approaches. Logic is also used as a formal basis for dynamic software architecture specification. Specific approaches include the Generic Reconfiguration Language (Gerel) [11], the Aguirre-Maibaum approach [1], and ZCL [8] which uses the Z specification language.

Other Approaches. Finally, other approaches exist that do not have a reconfiguration semantics based on graph theory, process algebra, or logic. Two approaches discussed in this paper are in this category: C2SADEL [17, 22] and RAPIDE [14, 26].

An overview of the above mentioned approaches can be found in Table 1. A detailed survey using a running example and a comprehensive description of each approach is given in [4].

3. SUPPORT FOR SELF-MANAGEMENT

All dynamic architectural changes have four steps (see Figure 1): initiation of change (①), selection of architectural transformation (②), implementation of reconfiguration (③), and assessment of architecture after reconfiguration (④).

We defined a self-managing architecture as an architecture in which the entire change process occurs internally. We determine which specification approaches support self-management by considering if the initiation of the change occurs internal to the software. In all of the approaches, if the initiation occurs internally then the selection and other steps of the change process can also be specified internally. Internal initiation usually involves monitors that provide the run-time information on which self-management decisions are based. Monitors are not specified explicitly in the surveyed approaches.

An example of an approach that supports internal initiation is the Le Métayer approach which provides this support through side conditions in the rewriting rules. For example, consider the rule given in [20]

$$\texttt{C(c), c.leave=true, CR(c,m), CA(m,c)} \rightarrow \emptyset$$

which removes a component `c` and two connectors `CR` and `CA`. The side condition `c.leave=true` in this rule refers to a public variable `leave` in a component `c` being true. The rewriting rule can only be applied when this side condition is satisfied.

Of the 14 specification approaches surveyed, we evaluated each approach to see if it supported internal initiation, external initiation

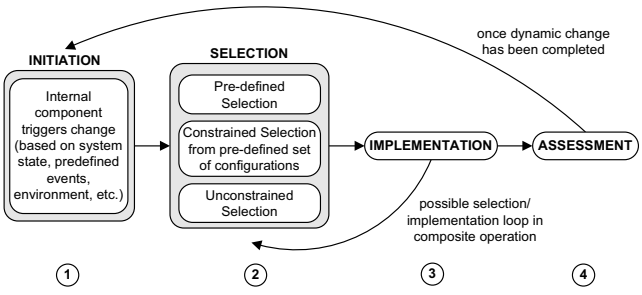

Figure 1: Change process in a self-managing architecture

The change process shown above has the same order of steps given in [3]. However, it is also possible to vary this order, for example, to conduct assessment during selection (②) as well as before the implementation of the change (③).

		Initiation	
		Internal	External
Graph	Le Métayer approach	●	●
	Hirsch et al. approach	○	○
	Taentzer et al. approach	?	?
	COMMUNITY	●	●
	CHAM	●	●
Process Algebra	Dynamic Wright	●	○
	Darwin	●	○
	LEDA	●	○
	PiLar	●	○
Logic	Gerel	●	⊙
	Aguirre-Maibaum approach	●	○
	ZCL	●	○
Other	C2SADEL	○	⊙
	RAPIDE	●	○

Table 2: Change initiation support

(e.g. external user), or both. 11 of the approaches explicitly allowed for internal initiation (see Table 2). In all of the tables in this paper we distinguish between criteria that are supported by the specification explicitly (●), supported externally by a tool or infrastructure (⊙), not supported (○), support unknown (**?**), and not applicable (-). We include the notion of a criterion being not applicable because not every specification approach can be classified perfectly using our criteria. We also include support unknown because despite our best efforts we were occasionally unable to discern how all of the approaches fit. For example, we were unable to determine the location of the initiation of the change process in the Taentzer et al. approach. In Sections 3.1 and 3.2 we will only discuss the approaches that explicitly support internal initiation since these are approaches that satisfy our definition of self-management. Details regarding the approaches that have been omitted can be found in [4].

3.1 Expressiveness

Our definition of a self-managing architecture includes systems with very limited forms of self-management. However, in many systems increased expressiveness is desirable. The ability of a system to manage its own architecture is, in general, limited by the types of changes it can make and the freedom to choose the appropriate change. We will now survey the expressiveness provided by different specification approaches in the context of these limitations.

3.1.1 Types of Change

The types of change that a self-managing system can make are limited at the architectural level by the reconfiguration operations that are available. For example, if a system can only add connectors but not components it is limited in the ways it addresses reconfiguration needs. In the context of change type we consider the ability of each approach to specify basic reconfiguration operations (the addition and removal of components and connectors) and composite reconfiguration operations (see Table 3).

Our comparison shows that the majority of approaches support all of the basic change operations. For example RAPIDE has one execution architecture event type for each of the basic operations:

		Basic Reconfiguration Operations				Composite Operations			
							Operation Constructs		
		Component Addition	Component Removal	Connector Addition	Connector Removal	Basic Support	Sequencing	Choice	Iteration
Graph	Le Métayer approach	●	●	●	●	●	○	●	○
	COMMUNITY	●	●	●	●	●	●	●	●
	CHAM	●	●	●	●	●	●	●	●
Process Algebra	Dynamic Wright	●	●	●	●	●	●	●	●
	Darwin	●	⊙	–	–	●	⊙?	⊙?	⊙?
	LEDA	●	○	●	○	○	○	●	○
	PiLar	●	●	●	●	●	●	●	○
Logic	Gerel	●	●	●	●	●	●	●	●
	Aquirre-Maibaum approach	●	●	●	●	●	?	●	?
	ZCL	●	●	●	●	●	?	?	?
Other	RAPIDE	●	●	●	●	●	?	●	○

Table 3: Reconfiguration operations support

```
CreateModule(...);
DeleteModule(module :  Event);
CreatePathway(...);
DeletePathway(pathway :  Event);
```

Approaches that did not support all of the basic operations include two of the process algebra approaches (Darwin, LEDA) that do not allow for the removal of architectural elements. The limitation in these approaches appears to be a result of high-level design decisions, not limitations of the underlying formalism. For example, Darwin was originally designed as a configuration language to be used for distributed systems and the removal of components in such a system can still occur at the programming language level.

In composite reconfiguration operations we consider not only the ability to add or remove subsystems or groups of architectural elements but also the constructs that can be used in specifying the operation (e.g., sequencing, choice, and iteration). Almost all of the approaches considered provide support for composite operations. However, only a few of the approaches provide full support for composite operation constructs such as sequencing, choice, and iteration. The scripts used in COMMUNITY and Gerel both provide these constructs. Consider, for example, a bank architecture in which connectors link customers (c) to their accounts (a). The following COMMUNITY script uses iteration to replace all `VIP` connectors (which allow overdrafts) by `standard` connectors (which do not) between a given account and the owners of the account.

```
script RestoreStandard
  in a:  Account
  prv i:  record(c:Customer; co:VIP)
  for i in match {c:Customer:  co:VIP | co(c,a)} loop
    remove i.co;
    create standard(i.c, a);
  end loop
end script
```

3.1.2 Selection

The ability to select different changes also provides increased expressiveness to self-managing systems. We distinguish between three levels of selection that a specification approach may support:

1. *Pre-defined Selection:* Once a dynamic change has been initiated, a change operation is chosen based on a pre-defined selection made prior to run-time.

2. *Constrained Selection from a Pre-defined Set:* Once a dynamic change has been initiated there is some choice in what operation to use. For example, a set of operations may be defined prior to run-time for a given situation or state. The system, upon reaching the situation, will select the appropriate change operation from the set.

3. *Unconstrained Selection:* Once dynamic change has been initiated there is an unconstrained choice regarding the appropriate change to make.

None of the approaches classified in this paper support unconstrained run-time selection, which provides the greatest level of expressiveness (see Table 4). The selection in most approaches is limited. Specifically, most approaches use a selection approach where one reconfiguration is pre-defined for a given situation. The exceptions include the graph rewriting approaches which allow for random selection of a reconfiguration, namely when multiple left hand sides of change rules match part of the current architecture.

An example of constrained selection from a pre-defined set in which the selection is not based on a non-deterministic choice, can be found in LEDA. Consider the following partial definition of a client-server system (originally given in [5]):

```
component DynamicClientServer {
   interface none;
   composition
     client:  Client;
     server[2]:  Server;
   attachments
     client request(r)<>
          if (server[1].n <= server[2].n)
              then server[1].serve(r);
              else server[2].serve(r);
}
```

		Selection		
		Pre-defined	Constrained from Pre-defined set	Unconstrained
Graph	Le Métayer approach	●	●	○
	COMMUNITY	●	●	○
	CHAM	●	●	○
Process Algebra	Dynamic Wright	●	●	○
	Darwin	●	○?	○
	LEDA	●	●	○
	PiLar	●	○	○
Logic	Gerel	●	?	○
	Aguirre-Maibaum approach	●	●?	○
	ZCL	●	○	○
Other	RAPIDE	●	●	○

Table 4: Selection support

		Management	
		Centralized	Distributed
Graph	Le Métayer approach	●	○
	COMMUNITY	●	○
	CHAM	●	○
Process Algebra	Dynamic Wright	●	○
	Darwin	●	●?
	LEDA	●	●?
	PiLar	○	●
Logic	Gerel	◉	○
	Aguirre-Maibaum approach	●	○
	ZCL	●	○
Other	RAPIDE	●	●

Table 5: Management support

In the example one client and two servers exist. The client is attached (<>) to one of the two servers based on the result of a boolean condition.

3.2 Scalability

Due to the growing size and complexity of software systems scalability is an important issue. The management of reconfiguration in dynamic software architectures can be either centralized in a specialized component or distributed across components. In general, a decentralized or distributed approach is more likely to scale. Currently, the management used in most of the specification approaches is centralized, not distributed (see Table 5). This is primarily because early types of dynamic architectural change such as ad-hoc and programmed often had centralized management. Newer definitions of change, as found in self-organising architectures, use distributed management in order to account for scalability in large-scale distributed systems [12].

Examples of centralized management can be found in Dynamic Wright where reconfigurations are specified in a configuror and in the Le Métayer approach where reconfiguration rewriting rules are specified in a coordinator. An example of distributed management can be found in the PiLar language. PiLar allows for multiple components to have constraints which may specify reconfiguration.

Some of the approaches such as Gerel do not specify the management but instead allow for the management to be determined in accompanying tools. In fact, one of the earliest approaches to distributed management was developed for approaches like Gerel [9]. In this approach a distributed management model was developed that would allowed programmed changes to be managed concurrently in a cooperative management setting.

4. CONCLUSIONS AND FUTURE WORK

In this paper we evaluate the ability of current dynamic software architectures specification approaches to represent self-managing architectures. The paper surveys 14 approaches to dynamic architectural specification in this context. We should note that it is difficult to survey all of the 14 approaches in such a small space. Our goal in this paper is two-fold. On the one hand, we want to provide all readers with a basic introduction to the challenges and issues faced in specifying self-managing architectures. On the other hand, for those readers who are familiar with the literature on software architecture specification we provide details on the ability of existing approaches to specify self-management. We are currently working on a journal version of this paper that will include all of the details about the approaches omitted from this paper due to space constraints. The journal version will also include additional classification dimensions not discussed in this paper.

For each specification approach, we consider basic support for self-management by evaluating the ability of the approach to specify systems in which a change is initiated internally. Additionally, we evaluate each approach in terms of expressiveness (ability to support multiple change types and selection approaches) and scalability (ability to support distributed management).

Our survey shows that the area of dynamic software architecture specification is well researched. There exist a lot of different sometimes conflicting notations, concepts, and definitions. Most of the approaches surveyed do reasonably well at answering questions dealing with the implementation of the change, such as *"Given system x, what happens when change y occurs?"*

However, a large number of the approaches support only limited forms of self-management. In the context of expressiveness, the results of our survey are mixed. On the one hand, many of the approaches support all of the basic operations as well as some form of composite operations. On the other hand, many of the approaches do not allow for more expressive selection to be specified. Selection is an important step in the dynamic architectural change process and needs to be better specified to enable more meaningful analysis. In the context of scalability many of the approaches do not consider distributed management schemes thus limiting the types and size of systems that can be specified.

To summarize, current approaches do a good job with specifying basic support for self-managing architectures. However, the approaches need to be adapted and updated to address the current limitations in terms of both expressiveness and scalability. An interesting example of this kind of work is the recent extension of Darwin. In [12], the traditional Darwin approach, surveyed in this paper, is extended by specifying Darwin architectures using a constraint-based approach in the Alloy modelling language. The extension is expressive in the selection of appropriate changes, provides scalability by supporting distributed management, and provides automatic analysis using the Alloy constraint analyzer.

5. REFERENCES

[1] N. Aguirre and T. Maibaum. A temporal logic approach to the specification of reconfigurable component-based systems. In *Proc. of the* 17th *Int. Conf. on Automated Software Engineering (ASE 2002)*, pages 271–274, 2002.

[2] R. Allen, R. Douence, and D. Garlan. Specifying and analyzing dynamic software architectures. In *Proc. of the* 1st *Int. Conf. on Fundamental Approaches to Software Engineering (FASE'98)*, 1998.

[3] J. Andersson. Issues in dynamic software architectures. In *Proc. of the* 4th *Int. Software Architecture Workshop (ISAW-4)*, pages 111–114, 2000.

[4] J. S. Bradbury. Organizing definitions and formalisms for dynamic software architectures. Technical Report 2004-477, Queen's University, 2004.

[5] C. Canal, E. Pimentel, and J. M. Troya. Specification and refinement of dynamic software architectures. In *Proc. of the Working IFIP Conf. on Software Architecture (WICSA'99)*, pages 107–126. Kluwer, 1999.

[6] P. Clements. A survey of architecture description languages. In *Proc. of the* 8th *Int. Work. on Software Specification and*

Design (IWSSD'96), pages 16–25. IEEE Computer Society, 1996.

[7] C. E. Cuesta, P. de la Fuente, and M. Barrio-Solórzano. Dynamic coordination architecture through the use of reflection. In *Proc. of the ACM Symp. on Applied Computing (SAC 2001)*, pages 134–140. ACM Press, 2001.

[8] V. C. C. de Paula. *ZCL: A Formal Framework for Specifying Dynamic Software Architectures*. PhD thesis, Federal University of Pernambuco, 1999.

[9] M. Endler. A model for distributed management of dynamic changes. In *Proc. of the 4th IFIP/IEEE Workshop on Distributed Systems: Operations and Management (DSOM'93)*, 1993.

[10] M. Endler. A language for implementing generic dynamic reconfigurations of distributed programs. In *Proc. of the 12th Brazilian Symp. on Computer Networks (SBRC 12)*, pages 175–187, 1994.

[11] M. Endler and J. Wei. Programming generic dynamic reconfigurations for distributed applications. In *Proc. of the Int. Work. on Configurable Distributed Systems*, pages 68–79. IEE, 1992.

[12] I. Georgiadis, J. Magee, and J. Kramer. Self-organising software architectures for distributed systems. In *Proc. of the 1st Work. on Self-Healing Systems (WOSS'02)*, pages 33–38. ACM Press, 2002.

[13] D. Hirsch, P. Inverardi, and U. Montanari. Graph grammars and constraint solving for software architecture styles. In *Proc. of the 3rd Int. Software Architecture Workshop (ISAW-3)*, pages 69–72. ACM Press, 1998.

[14] D. C. Luckham, J. L. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *IEEE Trans. on Software Engineering*, 21(4):336–355, 1995.

[15] J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proc. of the 4th ACM SIGSOFT Symp. on Foundations of Software Engineering (FSE-4)*, pages 3–14. ACM Press, 1996.

[16] J. Magee and J. Kramer. Self organising software architectures. In *Joint Proc. of the 2nd Int. Software Architecture Work. (ISAW-2) and Int. Work. on Multiple Perspectives in Software Development (Viewpoints '96) on SIGSOFT '96 Workshops*, pages 35–38. ACM Press, 1996.

[17] N. Medvidovic. ADLs and dynamic architecture changes. In *Joint Proc. of the 2nd Int. Software Architecture Work. (ISAW-2) and Int. Work. on Multiple Perspectives in Software Development (Viewpoints '96) on SIGSOFT '96 Workshops*, pages 24–27. ACM Press, 1996.

[18] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. on Software Engineering*, 26(1):70–93, 2000.

[19] D. L. Métayer. Software architecture styles as graph grammars. In *Proc. of the 4th ACM SIGSOFT Symp. on Foundations of Software Engineering (FSE-4)*, pages 15–23. ACM Press, 1996.

[20] D. L. Métayer. Describing software architecture styles using graph grammars. *IEEE Trans. Software Engineering*, 24(7):521–533, 1998.

[21] P. Oreizy, M. Gorlick, R. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[22] P. Oreizy and R. N. Taylor. On the role of software architectures in runtime system reconfiguration. In *Proc. of the 4th Int. Conf. on Configurable Distributed Systems (ICCDS '98)*, pages 61–70. IEEE Computer Society, 1998.

[23] B. Schmerl and D. Garlan. Exploiting architectural design knowledge to support self-repairing systems. In *Proc. of the 14th Int. Conf. on Software Engineering and Knowledge Engineering (SEKE 2002)*, pages 241–248. ACM Press, 2002.

[24] C. Szyperski. Component technology: what, where, and how? In *Proc. of the 25th Int. Conf. on Software Engineering (ICSE 2003)*, pages 684–693. IEEE Computer Society, 2003.

[25] G. Taentzer, M. Goedicke, and T. Meyer. Dynamic change management by distributed graph transformation: Towards configurable distributed systems. In *Proc. of the 6th Int. Workshop on Theory and Application of Graph Transformation (TAGT'98)*. LNCS 1764, Springer, 1998.

[26] J. Vera, L. Perrochon, and D. C. Luckham. Event-based execution architectures for dynamic software systems. In *Proc. of the Working IFIP Conf. on Software Architecture (WICSA'99)*, pages 303–318. Kluwer, 1999.

[27] M. Wermelinger. Towards a chemical model for software architecture reconfiguration. *IEE Proceedings - Software*, 145(5):130–136, 1998.

[28] M. Wermelinger and J. L. Fiadeiro. Algebraic software architecture reconfiguration. In *Proceedings of the 7th European Software Engineering Conference and 7th ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC/FSE'99)*, pages 393–409. LNCS 1687, Springer-Verlag, 1999.

[29] M. Wermelinger, A. Lopes, and J. L. Fiadeiro. A graph based architectural (re)configuration language. In *Proc. of the 8th European Software Engineering Conference and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2001)*. Software Engineering Notes, 26(5):21-32, ACM, 2001.