Organizing Definitions and Formalisms for Dynamic Software Architectures

Technical Report 2004-477

Jeremy S. Bradbury
Software Technology Laboratory,
School of Computing, Queen's University
Kingston, Ontario, Canada
bradbury@cs.queensu.ca

March 31, 2004

Abstract

Dynamic architectural change is defined as the addition and removal of components and connectors. Dynamic software architectures are those architectures that modify their architecture and enact the modifications during the system's execution. This behavior is most commonly known as run-time evolution or dynamism. As dynamic software architecture use becomes more widespread, it is important to gain a better understanding of this type of software evolutionary change and be able to classify formalisms, approaches and tools. Current evaluations in the areas of software architecture and evolutionary change have made strides in classification but are not sufficient to evaluate dynamic software architectures. A dedicated comparison of dynamic software architectures and architectural formalisms is necessary in order to gain a deeper understanding of run-time evolution. In this paper we present a set of classification criteria for the comparison of dynamic software architectures based on: change type, change process, and change infrastructure. We demonstrate the use of the criteria by classifying three types of dynamic software architectural change. In addition we survey 14 current approaches to the formal specification of dynamic software architectures based on graphs, process algebras, logic, and other formalisms. We then classify these approaches using the proposed criteria as well.

Contents

1 Introduction

Most research areas such as dynamic software architectures focus on the development of new approaches, new tools and new methods to advance the state-of-the-art. Although this is beneficial, there is also benefit in comparing existing approaches and assessing the progress that has been made. The understanding and perspective gained from such an evaluation not only highlights where we have been but also where we need to go.

Taxonomies provide us with a means to evaluate the work in a given area using a set of well-defined criteria. Currently there are no taxonomies or methods of organization that apply specifically to dynamic software architectures. Dynamic software architectures are software architectures that evolve at run-time, thus drawing expertise from both the software architecture and evolutionary change communities. In an effort to organize dynamic architectures we first consider existing taxonomies and classifications in the areas of architecture and evolutionary change.

Software Architecture. In the area of software architecture we considered an organizational framework for architectural formalisms, several surveys of architectural specifications, and an architectural ontology. In [SG95], Shaw and Garlan propose a two-dimensional organizational framework for formal approaches to software architecture. One dimension of the framework is genericity of the formalism. Genericity is divided into three categories: architectural instances, architectural styles, and architecture in general. The other dimension is the specification power of the formalism, divided into seven basic levels: capture, construction, composition, selection, verification, analysis, and automation. Also, in recent years several papers have surveyed Architecture Description Languages (ADLs) and provided broad comparisons [Cle96, MT00]. The survey in [Cle96] compared ADLs on attributes related to scope of language, capturing design history views, support for variability, expressive power, support for architecture creation, tool maturity, and others. The survey in [MT00] 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. In addition to organizational frameworks there has also been work on developing a software architecture ontology composed of architectural entities [GMW97]. The ontology consists of components, connectors, systems (configurations), ports (component interfaces), roles (connector interfaces), representations (hierarchical architecture layers), and representation maps (associations between layers).

Unfortunately, upon review the classification and ontology work in the area of software architecture is too general for our purposes. That is, it does not allow for a meaningful comparison of dynamic software architectures because it contains very little, if any, criteria related to evolutionary change. In fact, only the survey in [MT00] even considers run-time evolution in its evaluation.

Software Evolutionary Change. In the area of software evolutionary change we considered taxonomies based on the purpose of change [LS80, CHK+01] and the mechanisms required for change [BMZ+04]. The work in [LS80] distinguishes between software maintenance that is perfective, adaptive, and corrective. In [CHK+01] the purpose of maintenance and evolution is further divided into 12 distinct categories. In [BMZ+04], Buckley et al. develop a proposed taxonomy for change from a mechanisms perspective. Their taxonomy does not concentrate on *why* a change occurs but instead focuses on the *how*, *what*, *when*, and *where* of an evolutionary change.

Unfortunately, this work has a problem similar to that of the software architecture work mentioned earlier. The taxonomies in the area of software evolutionary change are too general and lack important information regarding architectural issues that are relevant in dynamic software architectures.

To address the need for a more focused comparison we propose a set of criteria based on the information we wish to understand about dynamic software architectures. Our goal is to evaluate dynamic architectures by answering three fundamental questions related to dynamic architectural change:

1. What type of change is supported?

2. What kind of process implements the change?

3. What infrastructure is available to support the change process?

In the next section (Section 2) we will discuss dynamic architectures in more detail and explain why a classification of dynamic software architectures is needed. We will then discuss our classification criteria (Section 3). We then evaluate the approach by classifying three types of dynamic architectural change (Section 4). After presenting our approach and demonstrating its use on different types of change we will shift focus to dynamic software architectural formalisms. In Section 5 we will discuss the use of formal specification techniques in the context of current practise. We will then survey 14 formal approaches to specifying dynamic software architectures based on graphs (Section 6), process algebras (Section 7), logic (Section 8), and other formalisms (Section 9). We will also classify each of these formal specifications using our approach (Section 10). Finally, a conclusion and open problems are given in Section 11.

2 Dynamic Software Architectures

Software architectures can be viewed using a common framework consisting of components and connectors. Components typically encapsulate information or functionality while connectors coordinate the communication between components. More specifically, software architectures describe the decomposition of a system into components, the interconnection of the components, and component interaction [SNH95].

Architectural modifications in software can occur at design time, pre-execution time, or run-time [Ore96]. Dynamic software architectures are those that modify their architecture and enact the modifications during the system's execution [MT00]. This behavior is most commonly known as run-time evolution or dynamism. Dynamic software architectures have several practical applications [OMT98]. For instance, in public information systems with high availability and in mission- and safety-critical systems the implementation of architectural change at run-time can decrease the cost of the change and remove the risk associated with taking the system off-line. Moreover, in many systems the use of dynamism can provide the end-user with the ability to modify the set of features available in the system.

Although dynamism has practical applications and has been researched by academics, there has been a lack of use of dynamism by software practitioners. This lack is particularly noticeable outside of the area of safety-critical and high availability software. In [Szy03], Szyperski addresses the lack of practical uses of dynamism in a discussion of the motivations for developing component-based systems. He believes the primary reasons dynamic components are not used is that systems developed with such components are "challenging in terms of correctness, robustness, and efficiency" [Szy03].

To support the development of correct and robust dynamic software architectures, many researchers have focused on the development of tools, approaches, and formalisms that provide guarantees regarding the correctness of dynamic architecture systems. A variety of different definitions of dynamic architectural change have been used in these approaches and given in the literature. The different definitions of dynamic architectural change help to demonstrate the need for a taxonomy. Below we list recent and widely cited definitions of dynamic architectural change. The list shows the high degree of variability with which dynamic architectural change is perceived within the research community.

Programmed dynamism [End94]. The change is triggered by the system and changes are defined prior to run-time.

Ad-hoc dynamism [End94]. Often initiated by the user as part of a software maintenance task, ad-hoc changes are defined at run-time and are not known at design-time.

Constructible dynamism [And00]. Constructible dynamism is a kind of ad-hoc change because it is initiated by an external event from, for example, a user or an external monitoring tool. A description language is used to describe the initial system configuration. A modification language is used to describe architectural changes. Finally, a dynamic updating system supports the architectural change via an architectural framework or middle-ware.

Adaptive dynamism [And00]. In a system that supports adaptive dynamism, the initialization and selection are done in the architectural framework or middle-ware (along with the implementation). Adaptive dynamism starts with a predefined set of configurations from the development stage of the software. The run-time dynamic changes are initiated by predefined events and then the system selects one of the predefined configurations and implements it. Hence, adaptive dynamism is a kind of programmed change.

Intelligent dynamism [And00]. Intelligent dynamism is also a kind of programmed change, similar to adaptive dynamism without the restriction of a predefined set of configurations. In intelligent dynamism possible configurations are dynamically constructed and the architecture assesses the quality of the configurations and implements the appropriate choice.

Constrained run-time dynamism [Ore96]. In this form of dynamism, a change can occur only after pre-defined constraints are satisfied. For example, constraints can be placed on the architectural topology and the program state.

Unconstrained run-time dynamism [Ore96]. No constraints are placed on the program state or the topology prior to running the system.

Transient connectors [WF98]. Dynamic reconfiguration of transient connectors is possible because of a boolean interaction condition which states the role of the connector and when the connector should be applied [FWM99]. The addition and removal of transient connectors is based on the status of the interaction condition.

Self-organising architectures [GMK02]. These are architectures that use no centralized management and allow components to connect or remove themselves from an architecture. A self-organising system is specified as a set of constraints on component composition, not in terms of a description of component composition and behavior.

Self-repairing systems [SG02]. Systems that have the ability to self repair are similar to self-organising architectures in that changes are initiated and assessed internally. Self-repairing system differ from self-organising in that they do not require distributed management. A self-repairing system has a control-loop of three primary activities: monitoring, interpretation, and reconfiguration. First, the run-time behavior is monitored to determine the need for change. Second, the monitored behavior is interpreted in the context of an architectural model. Third, reconfiguration is conducted once run-time analysis of the architectural model is complete. An example of run-time analysis would be to check if the model conforms to a given architectural style.

Self-adaptive software [OGT$^+$99]. Systems that can adapt to their environments by enacting run-time change. These system monitor observable behavior that may be external or internal to the system and in response enact change at the architectural level.

All of the definitions described above divide architectural change into the same four steps: initiation of change, selection of an architectural transformation, implementation of a reconfiguration, and assessment of the architecture after reconfiguration [And00]. Many of the definitions define different types of dynamism based on differences in these steps. The variety of definitions of dynamic architectural change and the need for an increased level of understanding regarding these systems underline the need for the development of a classification framework or taxonomy.

3 Organizing Dynamic Software Architectures

In developing useful criteria for classifying dynamic architectures we considered existing classification approaches in the software architecture community and the software evolutionary change community. First,

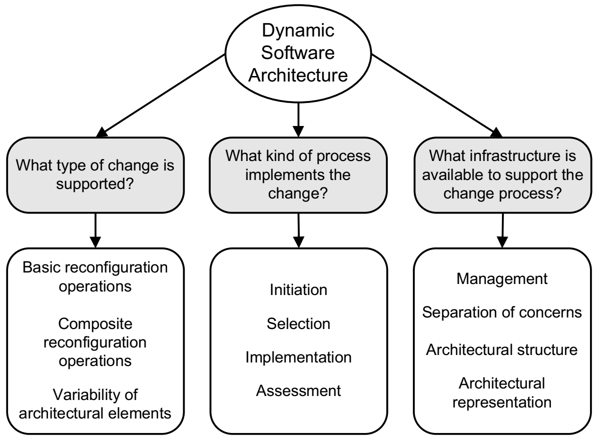


Figure 1: Classification criteria for dynamic software architectural change

The above diagram outlines the proposed classification criteria for dynamic software architectures. The approach is based on understanding three fundamental questions regarding change in a dynamic software architecture. The properties for each question are listed in the boxes at the bottom of the diagram.

we considered the Shaw and Garlan organizational framework for architectural formalisms, existing Architecture Description Language (ADL) surveys, and the ACME architecture ontology. Second, we looked at work on software evolutionary change taxonomies based on the purpose of change and the mechanisms for change. After considering the existing work we concluded that a set of new classification criteria for a dedicated comparison of dynamic software architectures was needed.

3.1 Proposed Classification Criteria

To address the need for a more focused comparison we propose a framework that answers the three fundamental questions presented in Section 1. That is, we organize dynamic architecture definitions and formalisms based on the type of change, the process of change, and the infrastructure for change (see Figure 1). A classification based on these three topics allows us to consider both evolutionary change and architectural issues. On the one hand, criteria related to change process demonstrate aspects of the evolutionary change within dynamic software architectures. On the other hand, criteria related to change infrastructure demonstrate architectural related issues.

3.1.1 Change Type

We evaluate the ability to specify the possible changes that can occur in a given system. The possible changes range over a two-dimensional grid consisting of reconfiguration operations as one dimension and architectural elements available to be used in the change as the other.

Basic Reconfiguration Operations. The basic reconfiguration operations are component addition, component removal, connector addition, and connector removal. The ability to represent these basic operations in a composite form is also considered. For example, the use of scripts is one means to achieve composite operations. In composite 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).

Composite Reconfiguration Operations. The ability to represent the basic operations in a composite form is also considered. For example, the use of scripts is one means to achieve composite operations. In composite 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).

Variability of Architectural Elements. The set of architectural element instances (the components and connectors) involved in the change can be fixed to the set of elements included with the software system prior to run-time or the set can be variable in components and connectors added during run-time. We can also have partial variability by fixing the types of elements but allowing new instances of elements. Variable sets of architectural element types are necessary to allow for the evolution of safety- and mission-critical software as well as high availability software.

3.1.2 Change Process

As previously mentioned all dynamic architectural changes have four steps: initiation, selection of architectural transformation, implementation of reconfiguration, and assessment of architecture after reconfiguration [And00] (Figure 2).

Initiation. The trigger that initiates a reconfiguration change can be external or internal. On the one hand, the run-time reconfiguration can be triggered externally by the environment or the user. On the other hand, the run-time reconfiguration can be triggered internally by the state of the system, for example, the status of a shared variable.

Selection. Reconfiguration operation selection considers the ability of a given dynamic architecture to support one of the following selection options:

1. *Explicit Selection:* Once dynamic change has been initiated an explicit selection by an external source

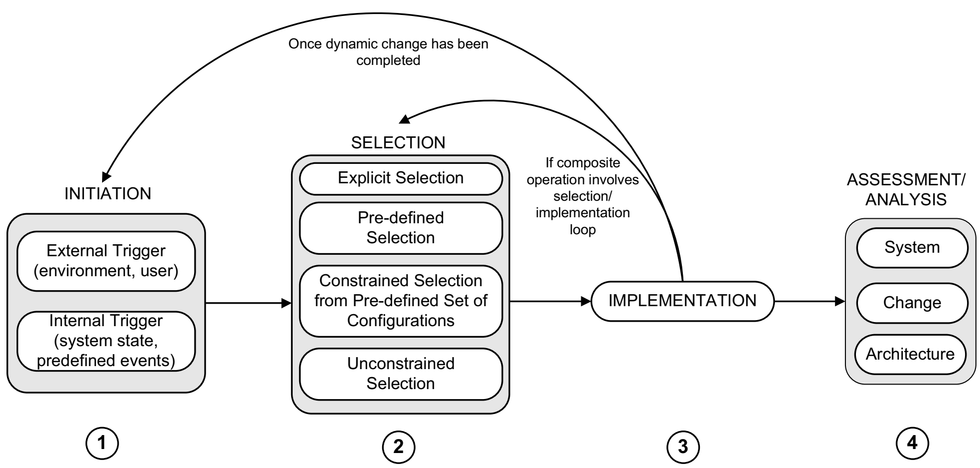


Figure 2: Change process

A typical change process has four distinct steps: initiation of change (①), selection of an architectural transformation (②), implementation of a reconfiguration (③), assessment of the architecture after reconfiguration (④). Although this is the order of steps given in [And00], it is also possible to conduct assessment during selection (②) as well as before the implementation of the change (③).

such as a user can occur at run-time.

2. *Pre-defined Selection:* Once dynamic change has been initiated, a selection pre-defined prior to run-time is made. This is similar to explicit selection in that the pre-defined selection is also explicit.

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

4. *Unconstrained Selection:* Once dynamic change has been initiated there is an unconstrained choice regarding the appropriate operation to use. On the one hand, if the architectural elements are fixed to those included prior to run-time this selection would involve operation selection from the entire set of possible operations available. On the other hand, if additional components are allowed to be made available for consideration in operations there could be a much larger set of operations from which to select.

Implementation. In the context of dynamic architectures the implementation is the underlying notation used to represent the change. For example, in formal approaches to dynamic architectures the implementation is simply the formalism used to represent the dynamic change.

Assessment. The assessment of a dynamic architecture usually involves constraints such as system constraints, change constraints, and architectural constraints. For example, in formal specifications of dynamic architectures we consider current support related to constraints. The assessment support in architectural specifications varies from direct formal analysis, to simulation, to formal analysis after translation to pure form of formalism used, to no assessment support.

3.1.3 The Infrastructure for Change

In addition to the specification of the change itself and the change process we also consider infrastructure issues which may influence the representation of dynamic reconfiguration systems.

Management. The management of reconfiguration can be either centralized in a specialized component or distributed across components. Self-organising architectures have distributed management, but otherwise the kind of management is typically independent of the kind of dynamic architecture.

Separation of Change and Computation. We consider the separation of change or reconfiguration from computation. A combination of reconfiguration and computation "...often results in a proposal that is not uniform, or has complex semantics, or does not make the relationship between reconfiguration and computation clear enough" [WLF01]. We distinguish between partial and full separation. That is, the ability to provide some separation of the computation and change is partial separation while the ability to provide separation of concerns into different distinct parts of the specification is full separation.

Architectural Structure. We consider how the structure or architecture of a system is represented. Some approaches explicitly represent the structure of the system's architecture, while others explicitly represent the architectural style of a given system.

Architectural Representation of Components and Connectors. We compare how components and connectors are represented within a given dynamic architecture. The components and connectors are the elements of the architecture that are used in the change and thus their representation is a criterion that should be considered. For example, the use of the CommUnity programming language in a category theory approach to dynamic architectures [WLF01] or the use of a notation similar to the Communicating Sequential Processes (CSP) in another graph-based approach [Mét96] are possible representations of architectural elements.

	Programmed	Ad-hoc	Self-organising
What type of change is supported?			
reconfiguration operations	possibly all	possibly all	possibly all
variability of architectural elements	fixed (most likely)	variable, fixed	fixed (most likely)
What kind of process implements the change?			
initiation	internal	external	internal
selection	constrained or possible pre-defined	explicit (most likely)	unconstrained (most likely)
implementation	*	*	*
assessment	*	*	*
What infrastructure is available to support the change process?			
management	centralized (most likely)	centralized (most likely)	distributed
separation of concerns	*	*	*
architectural representation: components	*	*	*
architectural representation: connectors	*	*	*

Table 1: Classification of programmed, ad-hoc, and self-organising dynamic change using proposed criteria

This table shows how programmed, ad-hoc, and self-organising systems would be classified using the proposed classification criteria. Values represented as "" indicate that the property is implementation dependent and not change dependent.*

4 Evaluation of Dynamic Software Architecture Definitions

The proposed classification criteria were developed with the intention of classifying dynamic architecture formalisms, tools and technologies. To demonstrate the use of the proposed taxonomy we consider how systems that use different types of change might vary. Specifically, we consider systems that are ad-hoc, programmed, and self-organising (See Section 2 for details). We have chosen these types of dynamism because they reflect many of the variations that exist within dynamic software architectures. Since we are considering types of systems and not actual systems, some of the property values are implementation dependent and cannot be defined. Table 1 provides complete details of our comparison.

Our comparison shows the distinguishing features that differentiate one type of change from another. For example, the initiation and selection steps in the change process, the management, and the variability of architectural elements. We are confident that if our approach was applied to specific systems of each type we would be able to further classify them by considering all of the implementation dependent properties.

The evaluation of different types of dynamic change is a first step in demonstrating the usefulness of our criteria. We will further evaluate our proposed criteria when we evaluate dynamic architectural formalisms.

5 Formal Specification Techniques for Software Architecture

Software architecture is most commonly described using informal or semi-formal techniques [SNH95], for example the Unified Modeling Language (UML) [HNS99]. Often architectures are expressed using box and line diagrams with a brief textual description outlining meaning and design choices [SG95]. Although informal and semi-formal techniques are most common, formal techniques do have advantages. In general formal approaches to architectural description are considered advantageous for several reasons: lack of ambiguity, ability to detect inconsistencies, the establishment of traceability between the system architecture and the source code, and the ability to provide automatic tools for analysis of the architecture.

A study of software architecture in industrial applications confirmed these results [SNH95]. The study also found that when architectures are described informally, similarities between architectures are difficult to exploit and differences between architectures are more difficult to identify.

Although formal specification has its benefits, it is not our intention to imply that informal techniques are not necessary. In fact, there is a need for both informal and formal specification of software architectures. This is reflected in the current practice of using formalisms. That is, even when formal techniques are used they are often complemented by the use of informal diagrams that enable greater understanding of the formal model.

In this paper the focus is on formal specification and we define a formal description of an architecture as a specification of an architecture that has the following [vL00]:

- *syntax:* rules for assessing well-formedness of sentences.
- *semantics:* rules for meaningful interpretation of syntax.
- *proof theory:* rules for inferring non-trivial information.

Formal approaches to dynamic software architectures involve the specification of the architectural structure of a system, the architectural reconfiguration of a system, and sometimes the component behavior of a system (see Figure 3). Approaches that specify all three aspects are said to be *complete specifications* while those that exclude component behavior are said to be *partial specifications*. In order to specify dynamic architectures the specification of both the architectural structure and the architectural reconfiguration are essential. The omission of one or both will lead to an incomplete specification in the dynamism context.

In addition to the distinction between partial and complete specifications it is important to discuss *configuration programming* [Kra90]. This notion is important to understand because of the strong connection between work on distributed system reconfiguration and software architectural reconfiguration. In fact, some of

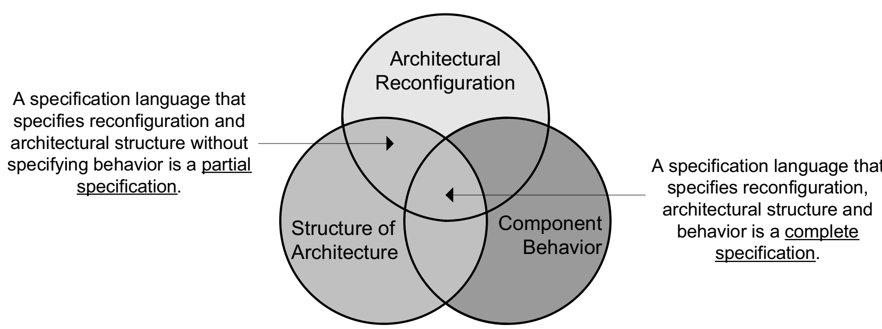


Figure 3: Partial and complete specification of dynamic software architectures

The above Venn diagram defines partial and complete specification in the context of architectural structure, architectural reconfiguration, and component behavior. If a specification approach consists of all three it is complete while the exclusion of component behavior makes a specification incomplete.

Dynamic Software Architecture Approaches	References
Graph-Based Formalisms	
Le Métayer Approach	[Mét96]
Hirsh et al. Approach	[HIM98], [HIM99], [HM99], [HM00], [Hir03]
Taentzer et al. Approach	[TGM98], [TGM99]
CommUnity	[Wer99], [WF99], [WLF01], [WF02]
CHAM	[Wer98b], [Wer98a], [Wer99]
Process Algebra Formalisms	
Dynamic Wright	[ADG97], [All97], [ADG98]
Darwin	[MDK93], [MDK94], [MDEK95], [MK96], [KM98], [Dar], [Dar97]
LEDA	[CPT98a], [CPT98b], [CPT99]
PiLar	[CdlFBS01], [CdlFBSB02a], [CdlFBSB02b]
Logic-Based Formalisms	
Gerel	[EW92], [End94]
Aguirre-Maibaum Approach	[AM02a], [AM02b], [AM03a], [AM03b]
ZCL	[dPJC98], [dP99], [dPJC00]
Other Formalisms	
c2sadel	[MORT96], [Med96], [Ore96], [OMT98], [Ore98], [OT98]
Rapide	[LKA$^+$95], [LV95], [Rap96], [VPL99]

Table 2: Formal specification approaches for dynamic software architectures

This table lists 14 approaches to the formal specification of dynamic software architectures and categorizes them according to the formalism that is used in the specification. The main references for each approach are also listed.

the approaches in the former have evolved to be used in the later. Configuration programming is a distributed programming approach in which a formal configuration language is developed to specify the configuration and reconfiguration of a system. This new specification language is designed to be used in conjunction with a standard programming language which specifies component behavior. The configuration language specification in this approach can be thought of as a partial specification.

The formal approaches to specifying dynamic software architectures that we consider are: the Aguirre-Maibaum approach, c2sadel, CHAM, CommUnity, Darwin, Dynamic Wright, Gerel, the Hirsh et al. approach, LEDA, the Le Métayer approach, PiLar , Rapide, Taentzer et al. approach, and ZCL. Details regarding these approaches can be found in Table 2.

6 Graph-Based Formalisms

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. One of the earlier approaches to formally specifying a static software architecture as a graph was the approach by Dean and Cordy which used typed directed multigraphs [DC95].

Graphs are not only a common formalism in the specification of static software architectures but also dynamic architecture. A natural way to specify reconfiguration in a dynamic architecture is to use graph rewriting rules to represent the reconfiguration. Approximately one-third of the approaches we consider in the specification of dynamic software architectures have used graph rewriting as an underlying formalisms. The Le Métayer approach and the Hirsh et al. approach are based on the use context-free graph grammars to represent architectural styles, graphs to represent an architectural instance, and standard rewriting rules to represent the reconfiguration. The Taentzer et al. approach uses the notion of a distributed graph. The CommUnity Approach has a formal specification language based firmly in category theory with architectural reconfiguration specified using the double-pushout graph transformation. Finally, the Chemical Abstract Machine (CHAM) approach uses a chemical metaphor that is similar to graph rewriting. We will now provide high-level summaries of these approaches. In our description of each approach we will use an example of a client-server system.

6.1 Le Métayer Approach

Architectural styles in the Le Métayer approach are defined as context-free graph grammars. Formally, the architectural style is defined by a four-tuple $[NT, T, PR, AX]$ where:

- *NT:* set of non-terminal symbols
- *T:* set of terminal symbols
- *PR:* finite set of production rules
- *AX:* the origin of derivation (an axiom)

A software architecture is generated from the graph grammar and is defined as a graph where nodes represent components and edges represent connectors [Mét96]. Nodes, in the graph of an architecture, map to programming language descriptions of the component's internal behavior to form an architectural instance. A coordinator is used to manage the architecture and is expressed using conditional graph rewriting.

Figure 4 provides an example of a client-server architecture using the Le Métayer approach. For the client-server example the architectural style is defined as the four-tuple:

$$H_{cs} = [\{CS, CS_1\}, \{M, X, C, S, CR, CA, SR, SA\}, R, CS]$$

where R is the set of productions in the "Architectural Style" section of Figure 4. In addition to the architectural style, the representation of a system in the Le Métayer approach also defines: link types, a coordinator, and entities. The link types are used as terminal symbols in the graph grammar and are defined in terms of the components they connect. The coordinator defines the reconfiguration of the system and consists of a set of graph rewriting rules. In Figure 4, there are two rules, one to add a client component and the other to remove the component. To restrict the behavior of the system side conditions are used in the rewriting. For example, consider the rule:

```
X(x), M(m), x.newc = true → X(x'), M(m), CR(c,m), CA(m,c), C(c)
```

The side condition `x.newc = true` in this rule refers to the public variable `newc` in the external environment being true. The rewriting rule for adding a client component can only take place when this side rule is satisfied. An algorithm is also provided to verify that the changes made by the rewriting rules of the coordinator do not violate the architectural style. As previously mentioned, the components can be specified in any programming language however Le Métayer defines a component as consisting of public and private variables, input and output links, entity names (used in the architectural style production rules), and a body indicating the components behavior. The body is expressed using a notation similar to the Communicating Sequential Processes (CSP). In this notation the $\Box$ is used to represent choice, the * is used to represent iteration, and the ? and ! are used to specify input and output commands. For example, in the client-server system the output command $s \in SR!r$ in the manager is matched up with the input command $m \in SR?r$ in some client to form a connection. In this case the client instance is not explicitly named. Commands of the

```
Architectural Style
CS       ⇒ CS1(m)
CS1(m)  ⇒ CR(c,m), CA(m,c), C(c), CS1(m)
CS1(m)  ⇒ SR(m,s), SA(sm,), S(s), CS1(m)
CS1(m)  ⇒ M(m), X(x)

Link Types
C: client                        S: server
M: manager                       X: external
CR: client × manager             CA: manager × client
SR: manager × server             SA: server × manger

Coordinator
Coo_CS =
  X(x), M(m), x.newc = true → X(x'), M(m), CR(c,m), CA(m,c), C(c)
  C(c), c.leave=true, CR(c,m), CA(m,c) → ∅

Entities

client:   pub  leave : bool
          priv r,a : int
          out  CR
          in   CA
          ent  m
          body Init_c;
               leave := false;
               *[Cond1 → C1 □
                 Cond2 → m ∈ CR ! r;
                 m : CA ?a];
               leave := true

server:   priv r : int
          out  SA
          in   SR
          ent  m
          body *[m ∈ SR ? r → m : SA ! f(r)]

manager:  priv r,a : int
          out  SR, CA
          in   CR, SA
          ent  c,s
          body *[c ∈ CR ? r → s ∈ SR ! r;
                 s: SA ? a;
                 c: CA ! a]

external: pub  newc : bool
          body Init_e;
               newc := false;
               *[Cond3 → C3];
               newc := true;
```

Figure 4: Sample code for a client-server example using the Le Métayer specification language [Mét96]

form $s : SA?a$ are used when there is an explicit specification of the other component involved. For example, in the above it is server s. Additionally, in the client-server example $\mathtt{C}_1$, $\mathtt{C}_3$, $\mathtt{Cond}_1$, $\mathtt{Cond}_2$, and $\mathtt{Cond}_3$ are boolean expressions which for simplicity have not been completed.

Informally the CSP-like notation, used in the bodies of the components in our client-server example, does the following:

- *Client:* The client is first initialized and the public variable `leave` is set to false. Then the component does some internal computation or sends a request to a manager component and waits for a response. It is possible for this behavior to occur multiple times before the `leave` variable is set to true. The coordinator uses the `leave` variable as a side condition in the rewriting rule for removing a client.
- *Server:* The server receives requests from the manager and for each request, sends a response back to the manager.
- *Manager:* The manager continuously waits for client requests, sends them to the server, receives an answer from the server, and finally sends the answer to the client.
- *External:* The external component that represents the environment is first initialized and the pulic variable `newc` is set to false. Then the environment does some internal computation, possibly multiple times, before setting the `newc` variable to true. This variable is used as previously discussed in the coordinator.

6.2 Hirsh et al. Approach

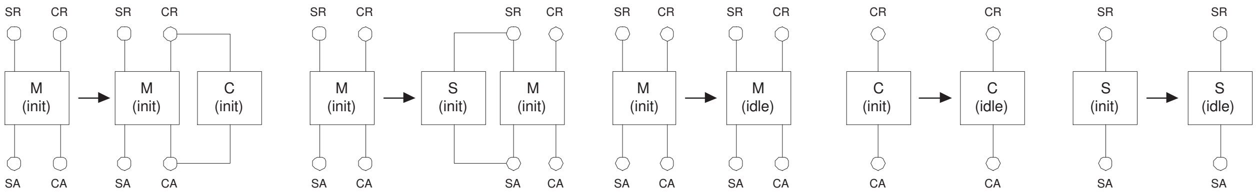
The Hirsh et al. approach to representing architectural styles is similar to the Le Métayer approach. As in [Mét96], context-free graph grammars are used to represent architectural styles and graphs are used to represent architectures. Unlike the Le Métayer approach, edges represent components and nodes represent connectors [HIM98]. Edges are represented as boxes while point-to-point communication nodes are white circles and broadcast communication nodes are black circles.

The context-free grammars used in the Hirsch et al. approach involve three distinct sets of production rules of the form $L \rightarrow R$ where L is the part of the graph to be rewritten and R is the new graph. The three sets are:

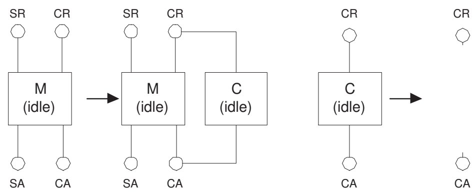
1. *Static productions:* used for the construction of the initial configuration of an architectural style.
2. *Dynamic productions:* used to create and remove architectural elements and define dynamic evolution.
3. *Communication pattern productions:* used to describe the communication between components.

This brings up another difference between the Le Métayer approach and the Hirsh et al. approach. Hirsch et al. use a uniform representation of grammars for the construction of the architectural style, the evolution of the architecture, and the behavior of the system while Le Métayer used grammar production rules for the construction, graph rewriting rules for the evolution, and a CSP-like notation to represent behavior [HIM99].

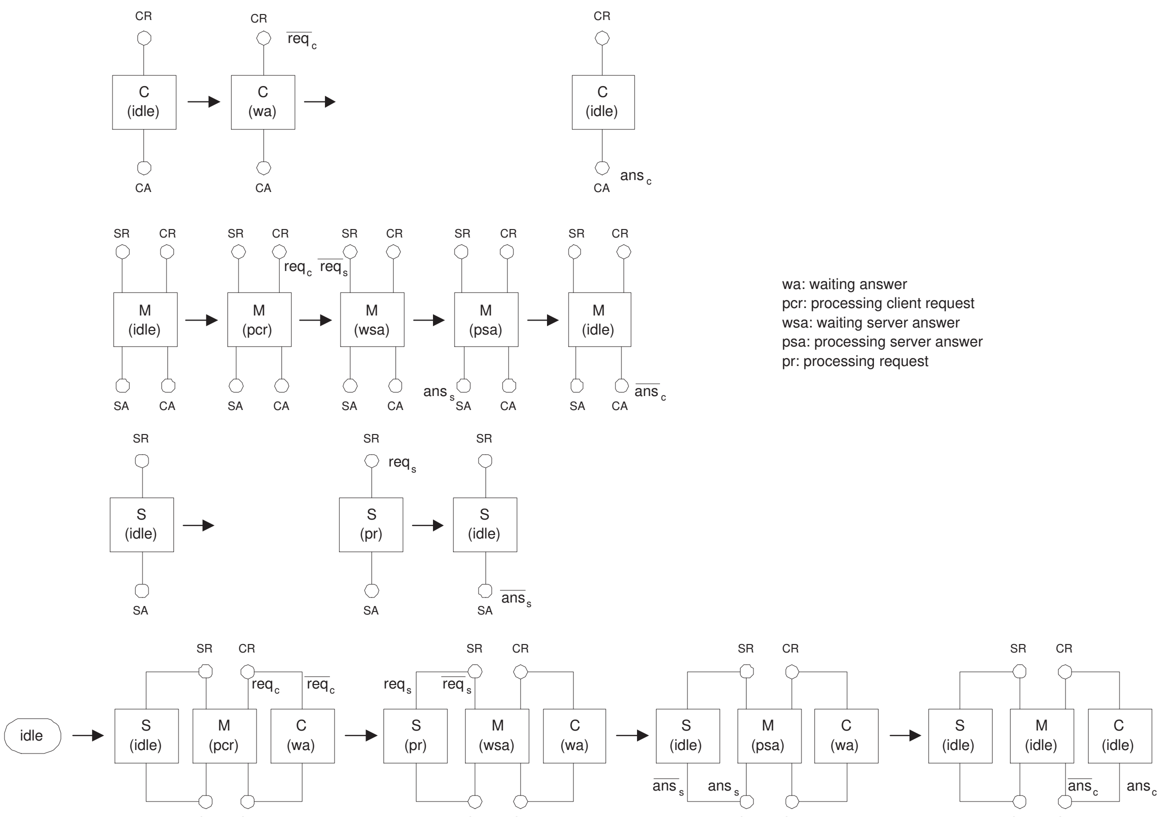
Figure 5 provides the production rules for representing a client-server example. Figure 5(a) shows the static productions. The first two productions (from left to right) show how to construct an architecture while the last three productions show how to initialize the architecture after construction is complete. Figure 5(b) shows two dynamic productions used for evolution, the first adds a client component to the architecture and the second removes a client component from the architecture. Figure 5(c) shows the communication pattern productions which describe the system behavior in terms of communication. The first three rows of productions show the state of each component and its communication ports as time moves from left to right.



(a) Static productions for client-server example



(b) Dynamic productions for client-server example



(c) Communication pattern productions for client-server example

Figure 5: Client-server example using Hirsh et al. approach [HIM98]

The last row of productions shows the same behavior but in a system view rather then a component view. The communication pattern productions are labeled with a Calculus of Communicating Systems (CCS) notation. Hirsch et al. use $\overline{event}$ to represent an initation event and *event* to represent an observed event. For example, in the client-server system, a client initiates a $\overline{req_c}$ and the manager observes the event (represented as req_c).

Dynamic productions can be applied only when the components are in an idle state. For example, in Figure 5(b), a client component can only be connected to a manager component when the manager is idle while a client component can only be removed when it itself is idle. The communication pattern productions in Figure 5(c) govern when a component will reach an idle state.

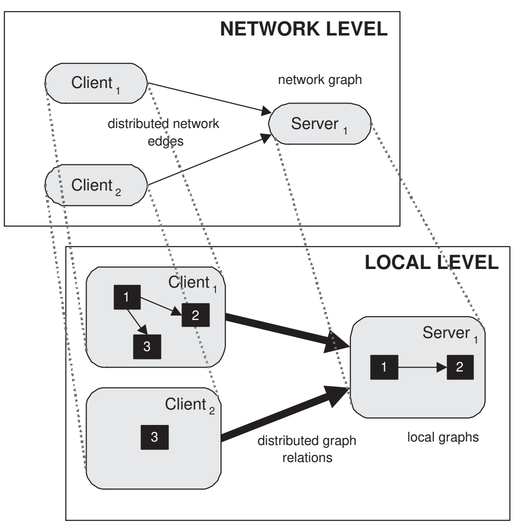
6.3 Taentzer et al. Approach

Software architectures and reconfiguration are described in the Taentzer et al. approach using distributed graphs and distributed graph transformations. Distributed graphs consist of two types of graphs [TGM98]:

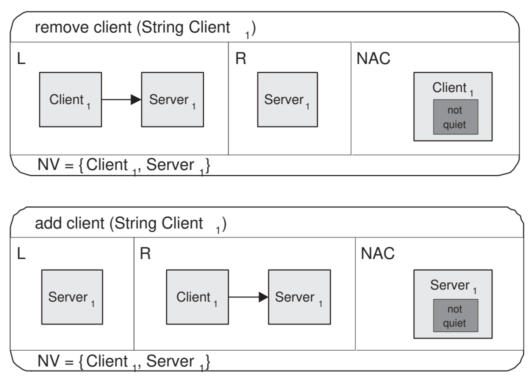
1. *network graph:* nodes represent components while edges represent communication paths (connectors).

2. *local graph:* each node in a network graph has a corresponding local graph and each edge in a network graph has a corresponding relation at the local graph level.

Figure 6(a) contains a client-server system represented as a distributed graph. At the network level we see the architecture of the software showing the relationship between client components and a server component. Each node at the network level is tied to a local graph. This graph has nodes representing: body elements



(a) Network and local level architecture



(b) Change management rules

Figure 6: Client-server example as a distributed graph

In (a), a client-server distributed graph is presented. At the network level we see the topology of the architecture as interconnected components and connectors. Each component is also related to a local level graph. In (b), we see two network level change management rules for removing and adding a client component. In these change management rules the name of the rule and any parameters appear at the top. Node attribute variables are defined at the bottom and the NAC conditions are defined on the right.

(medium gray), exported elements (light gray), imported elements (black), and imported/exported elements (light gray and black mixed) [TGM98]. Basically, the body elements represent the local data and the other elements represent the interfaces. For simplicity we have not made the distinction between the different coloured nodes in our example. Behavior and state change at the local (component) level is represented as local graph transformations.

Distributed graph transformations can be used at the network level as well as the local level. Transformations at the network level describe dynamic reconfiguration. Distributed rules are presented graphical where the left hand side represents the old or previous configuration and the right-hand side represents the new or future configuration. Only nodes that are affected by the change are included in the rule. Figure 6(b) contains two reconfiguration rules, one for client addition and one for client removal.

A rule can not take place unless the left-hand side of the rule matches part of the current system and unless the nodes are in a quiescent state . A quiescent state exists "...if the application's state is consistent and there is no communication in progress between nodes affected by a change nor betweren them and their environment" [TGM99]. Before a rule is applied, application conditions have to be checked (e.g. gluing conditions, connection conditions, network conditions, splitting conditions, and negative application conditions (NACs)). This ensures that an illegal graph structure is not obtained by applying the rule [TGM98] and thus guarantees that after the transformation we still have a distributed graph.

6.4 CommUnity

Software architecture is represented by a labeled graph using category theory semantics. Components are nodes in the graph and are written in a subset of the CommUnity program design language [FM97] and connectors are just degenerate CommUnity programs. Reconfiguration of a software architecture is done using algebraic graph rewriting. Specifically, reconfiguration uses the double-pushout graph transformation.

Recently this work has been extended and an architectural specification language has been developed to hide the underlying formalism of category theory [WLF01]. Figure 7 shows a partial specification of an architecture in this language.

```
architecture ClientServer                    connector ServerAnswer (Server, Manager)...
design Client                                variables name : nat
   in CA : bool                              constraint C
   out CR : bool                             end architecture
   do ...
design Manager...                            script Main
design Server....                            ...
connectors                                   end script
connector ClientReceive(Client, Manager)
   design                                    script AddClient
      in                                     out c: Client;
      out                                    c := create Client with data := 0;
      do ...                                 end script
   channel ...
   channel ...                               script RemoveClient
connector ClientAnswer(Manager, Client)...   remove ...
connector ServerReceive(Manager, Server)...  end script
```

Figure 7: Sample code for a client-server example using the CommUnity specification language

The structure and behavior of a system is defined through component definitions (using the keyword `design` and connector definitions which connect the components. For example, in our client-server system we have three component definitions: *Client*, *Manager*, and *Server*. We also have four connectors defining the communication of these components. The *ClientReceive* connector, for example, connects a *Client* with a *Manager*.

Reconfiguration in the CommUnity approach supports basic and composite commands. Basic commands support the notion of component and connector addition and removal. Composite commands are combinations of basic commands using operators for sequencing, choice, and iteration. The reconfiguration commands are contained in hierarchical scripts that are separate from the description of the architecture and behavior of the system.

A reconfiguration interpreter manages the coordination of the system in terms of computation and reconfiguration as follows:

1. Execution of one computation step
2. If the user wants he/she can choose and execute a top-level script. If no script is chosen then the 'Main' script, if it exists, is executed.
3. Go back to the first step.

Note that the reconfiguration scripts are implemented on the architecture only if execution of the entire script will not violate any of the constraints contained in the global conjunction C. [WF99].

6.5 Chemical Abstract Machine (CHAM)

The CHAM formalism is based on a metaphor of chemical solutions and reactions [BB92]. Reactions occur based on a set of reaction rules which are applied to molecules. This concept is very similar to using grammars and rewriting rules. The CHAM formalism has been used in the specification of static software architectures [IW95, IWY97]. More recently, Wermelinger has applied the use of the CHAM formalism to dynamic software architectures [Wer98b, Wer99]. In his approach a creation CHAM is used to formalize an architectural style while an evolution CHAM is used for the reconfiguration. If we continue the metaphor of solutions and reactions, the creation CHAM is a set of reaction rules that can generate all solutions (architectures) possible and the evolution CHAM is a set of reaction rules that can transform the solution (architecture) after it has been created [Wer98b]. The reaction rules for both types of CHAMs show how components are linked. In order to also show how components can be created and removed two commands are required: *create component* (cc) and *remove component* (rc) [Wer98b].

The CHAM approach to dynamism has been applied to ad-hoc and programmed dynamism as well as self-organising architectures. The difference between each type of change when represented in the CHAM is as follows [Wer98b]:

- *ad-hoc dynamism:* ad-hoc changes are often initated externally by the user. Therefore, the evolution CHAM that contains the reaction rules which represents these changes only consume creation and removal commands (i.e. the cc and rc commands appear only on the left side of the rules). The consumption of creation and removal commands models the consumption of ad-hoc changes by the system from the user.
- *programmed dynamism:* programmed changes are usually initated internally and have been defined prior to run-time. Therefore, the evolution CHAM that contains reaction rules which represents these changes can both consume and produce creation and removal commands (i.e. the cc and rc commands can appear on both sides of the rules). The production of creation and removal commands models the ability of changes to be produced internally in a system with programmed dynamism.

Molecule Grammar:
Molecule := Component | Link | Commmand
Component := Id: Type
Type := C | M | S
Link := Id−Id
Command := cc(Component) | rc(Id)

Creation CHAM:
cc(m:M) → c:C, c−m, cc(m:M)
cc(m:M) → s:S, m−s, cc(m:M)
s:S, cc(m:M) → s:S, m:M

Evolution CHAM:
cc(c:C), m:M → c:C, c−m, m:M
cc(s:S), m:M → s:S, m−s, m:M
rc(c), c:C, c−m → s:S, rc(s), s:S, m−s → s:S
m:M, rc(m), cc(m:M) → ms, m:M
m−s, m:M → m−s, m:M
c−m, m:M → c−m, m:M

Figure 8: Client-server example using CHAM formalism [Wer98b]

- *self-organising architectures:* self-organising architectures involve distributed reconfiguration of connectors. Therefore, the evolution CHAM for this type of dynamism contains reaction rules that neither consume nor produce reconfiguration commands. That is, the reaction rules do not involve the creation or removal of components but instead involve only the reorganization of connectors defined by the rules themselves.

We show the grammar for molecules, the creation CHAM, and the evolution CHAM for an example of an ad-hoc client-server example in Figure 8. The molecule grammar defines a molecule as a component, link, or command. The creation CHAM specifies the client-server architectural style and assumes that any system matching the style has at least one server. The evolution CHAM outlines the architectural reconfiguration. In our client-server example the evolution CHAM described client creation, server creation, client removal, server removal, and manager substitution.

7 Process Algebra Formalisms

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 [vG87]. A variety of process algebras exist including the Calculus of Communicating Systems (CCS), Communicating Sequential Processes (CSP), and the π-calculus. We will now describe four approaches to specifying dynamic software architectures with process algebra: Dynamic Wright, Darwin, LEDA, and PiLar.

7.1 Dynamic Wright

Dynamic Wright represents the structure of an architecture as a graph in which components and connectors are nodes (See Figure 9) [ADG98] and specifies the behavior and reconfiguration of a system in a variant of the process algebra CSP. "The basic idea is to treat both components and connectors as processes, which synchronize over suitably renamed alphabets" [ADG97].

Dynamic Wright supports the notion of internal choice ($\sqcap$), external choice ($\Box$), and successful termination (§) in the context of component behavior [ADG97]. If internal choice is used a component can choose whether or not to perform an event. If external choice is used a component has to perform an event from a finite list and cannot terminate until a successful termination has occurred. Additionally, if a component initiates an event it is written with an overbar (e.g. $\overline{event}$) while an event that is just observed has no overbar (e.g. *event*) [All97].

The semantics of the variant of CSP used is defined by a translation to pure CSP. However, one of the restrictions of the CSP process algebra, in the context of Dynamic Wright, is that CSP describes static configurations only. Dynamic configurations cannot be described. Thus, the dynamic behavior has to be simulated in the translation. This is achieved by [ADG97]:

- Restricting a given system to a finite set of configurations.
- "Tagging" each event with the configuration in which it occurs.
- Simulating reconfiguration by changing the tags.

We will now show how CSP is used in the definition of component and connector types (as defined in a style) and the specification of architectural configuration and reconfiguration (as defined in a configuror).

The style in a Dynamic Wright specification defines three kinds of information [ADG98]:

1. *Component types:* a component has a defined interface (port) and defined behavior (computation).
2. *Connector types:* a connector has a defined interface (role) and defined behavior (glue).
3. *Constraints:* first-order logic constraints can be specified regarding the components, connectors, and their interactions.

To demonstrate the specification of components and connectors in Dynamic Wright we consider the port p in the Client component (see Figure 10) defined as:

$$p = \overline{request} \rightarrow reply \rightarrow p \sqcap \S$$

This statement means that port p will initiate a *request* event and observe a *reply* event any number of times before deciding to terminate.

A configuror describes how the components and connector types of an architecture interact. In the configuror in Figure 10 an initial style is first described that instantiates the component and connector types and then attaches them. This initial configuration is prefaced by the keyword **style**.

In addition to an initial configuration two reconfigurations are defined. In each reconfiguration, also prefaced with the keyword **style**, *new*, *del*, *attach* and *detach* operations are used to change the way architectural elements are connected. Each reconfiguration is invoked by a condition, for example, consider the definition of *WaitForDown* in the client-server example (see Figure 10). In this example a fault can occur or the system can run correctly and terminate. In the case of the fault, the *Primary* server is detached and the *Secondary* server is attached.

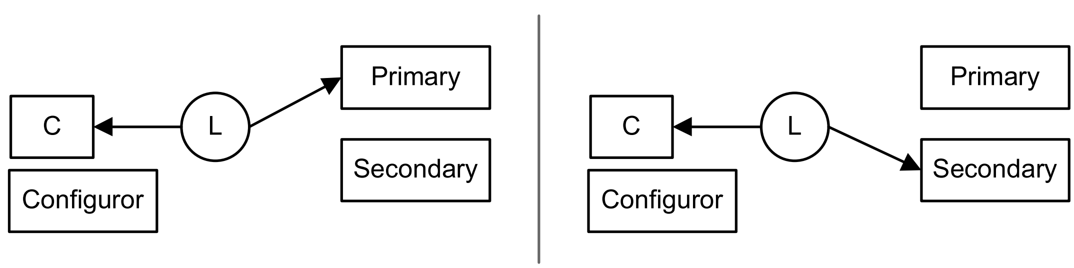


Figure 9: Alternating configurations of fault-tolerant client-server system with Dynamic Wright [ADG97]

Style Fault-Tolerant-Client-Server

Component Client

Port $p = \overline{request} \rightarrow reply \rightarrow p \sqcap \S$

Computation $= internalCompute \rightarrow p.request \rightarrow p.reply \rightarrow$ **Computation** $\sqcap \S$

Component FlakyServer

Port $p = \S \Box (request \rightarrow \overline{reply} \rightarrow p \sqcap control.down \rightarrow (\ \S \Box control.up \rightarrow p))$

Computation = $\S \Box (p.request \rightarrow internalCompute \rightarrow \overline{p.reply} \rightarrow$ **Computation**
$\sqcap control.down \rightarrow (\ \S \Box control.up \rightarrow$ **Computation**$))$

Component SlowServer

Port $p = \S \Box (control.on \rightarrow \mu Loop.(request \rightarrow \overline{reply} \rightarrow Loop \Box control.off \rightarrow p \Box\ \S))$

Computation = $\S \Box control.on \rightarrow \mu Loop.(p.request \rightarrow internalCompute \rightarrow$
$\overline{p.reply} \rightarrow Loop \Box control.off \rightarrow$ **Computation** $\Box\ \S)$

Connector FaultTolerantLink

Role $c = \overline{request} \rightarrow reply \rightarrow c \Box\ \S$

Role $s = (request \rightarrow \overline{reply} \rightarrow s \Box control.ChangeOK \rightarrow s) \Box\ \S$

Glue $c.request \rightarrow \overline{s.request} \rightarrow$ **Glue**
$\Box s.reply \rightarrow \overline{c.reply} \rightarrow$ **Glue**
$\Box\ \S$
$\Box control.ChangeOK \rightarrow$ **Glue**

Constraints

$\exists! s \in Component, \forall c \in Component : TypeServer(s) \wedge TypeClient(c) \Rightarrow connected(c,s)$

EndStyle

Configuror DynamicClientServer

Style Fault-Tolerant-Client-Server

$new.C : Client$
$\rightarrow new.Primary : FlakyServer$
$\rightarrow new.Secondary : SlowServer$
$\rightarrow new.L : FaultTolerantLink$
$\rightarrow attach.C.p.to.L.c$
$\rightarrow attach.Primary.p.to.L.s \rightarrow WaitForDown$

where

$WaitForDown = (Primary.control.down \rightarrow Secondary.control.on \rightarrow L.control.changeOk \rightarrow$
Style Fault-Tolerant-Client-Server
$detach.Primary.p.from.L.s \rightarrow attach.Secondary.p.to.L.s \rightarrow WaitForUp)$
$\Box \S$

$WaitForUp = (Primary.control.up \rightarrow Secondary.control.off \rightarrow L.control.changeOk \rightarrow$
Style Fault-Tolerant-Client-Server
$detach.Secondary.p.from.L.s \rightarrow attach.Primary.p.to.L.s \rightarrow WaitForDown)$
$\Box\ \S$

Figure 10: Client-server example using Dynamic Wright [ADG97, ADG98]

7.2 Darwin

Darwin is a declarative ADL that supports dynamic behaviour at the architectural level. Darwin was originally developed as a configuration language for Regis, a distributed programmming environment [MDK94]. The use of the configuration language Darwin has evolved and led to its use as an architecture description language. Darwin has both a graphical and text representation and an operational semantics based on the π-calculus, an extension of CCS [MDEK95]. Unlike the CSP notation used in Dynamic Wright, the π-calculus is a mobile calculus developed specifically for communicating systems with changing structure [MPW92a, MPW92b].

Systems defined using the Darwin language can be hierarchical, consisting of basic computational components and composite components. The basic components are usually defined in a programming language. For example, in the Regis environment the basic components are programmed in C++ [MDK94]. Although basic components are specified in a programming language, a basic component's interface is still represented in Darwin as a set of communication objects. The two types of communication objects are: **provide** (allow other components to interact with them) and **require** (needed to interact with other components). Communication objects can also have annotations of information such as type which can be used to generate class headers (e.g. in Regis).

Composite components, unlike basic components have no programming language implementation and are only represented in the Darwin language. A composite component can consist of instantiations of basic components and other composite components. The connectors between the instantiated components are declared in **bind** statements. It is possible for many require communication objects to be bound to the same provide object but only one require object can bind to a provide object [MDEK95].

Darwin allows for dynamic behavior at the architectural level in several different ways [MK96]:

- *Lazy instantiation:* a component is instantiated only after another component requests a service that it provides. For example, in the client-server example in Figure 11(a), 11(b) a server is instantiated only when a client requests service *p*.

- *Dynamic instantiation:* a component is instantiated arbitrarily based on a service. For example, in the client-server system in Figure 11(c), 11(d) a client is instantiated by a request for service *d*. Furthermore, the client could also be instantiated through a service in the server as well. For example:

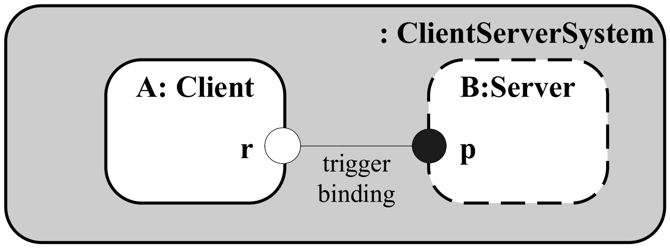
  ```
  B.createClient -- dyn client
  ```

  where `B` is an instance of a server and `createClient` is a requirement for a dynamic instantiation of a client component.

Darwin does not allow for the specification of dynamic bindings or component removal since the **dyn** operator only applies to the instantiation of components.

7.3 LEDA

LEDA, like Darwin, uses the π-calculus as the underlying formalism to specify dynamic software architectures. Components are specified in terms of an interface, composition, and attachments [CPT99]. The interface is defined using instances of roles which define the behavior of components with other components. Specifically, a behavioral representation of what is provided by the component and what is required to connect to the component is defined. If the component is a composite component, the composition or structure



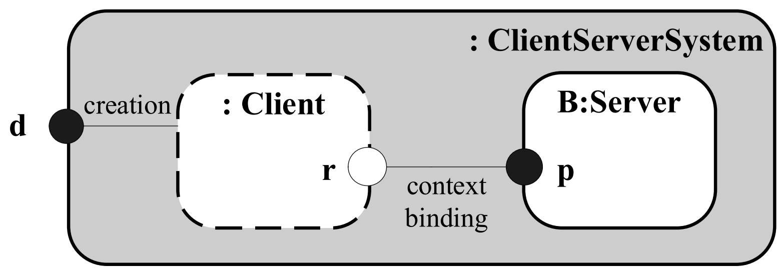
(a) lazy instantiation (graphical representation)

```
component Server {
   provide p;
}

component Client {
   require r;
}
```

```
component ClientServerSystem {
   inst
     A: Client;
     B: dyn Server;
   bind
     A.r -- B.p;
}
```

(b) lazy instantiation (text representation)



(c) dynamic instantiation (graphical representation)

```
component Server {
   provide p;
}

component Client {
   require r;
}
```

```
component ClientServerSystem {
   provide d <dyn>;
   inst
     A: Client;
     B: Server;
   bind
     d -- dyn Client;
     Client.r -- B.p;
}
```

(d) dynamic instantiation (text representation)

Figure 11: Sample code for two client-server examples using the Darwin specification language [MK96]

```
component Server{
  interface
    serve : Serve(request) {
      names
        n : Integer := 0;
      spec is
        (request?(reply).
         ( new service)reply!(service).
         n++.Serve(request);
    }
  composition
    service[] : any;
}

component Client {
  interface
    request : Request(request) {
      spec is
        (reply)request!(reply).
         reply?(service).Request(request);
    }
  composition
    service[] : any;
}

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

Figure 12: Sample code for a client-server example using the LEDA specification language [CPT99]

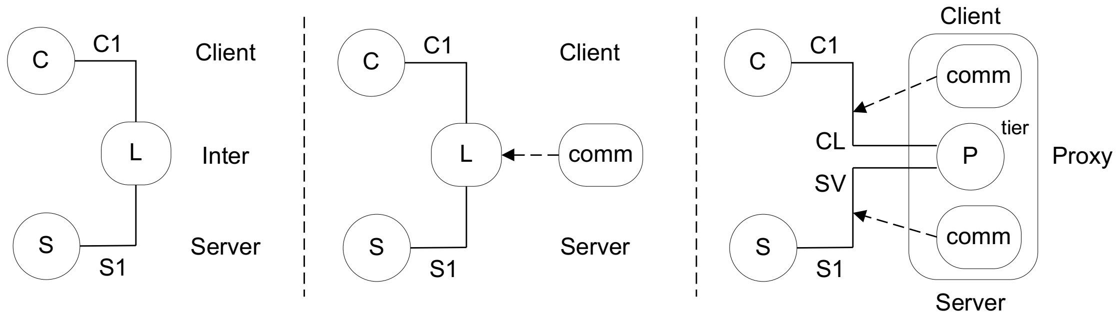
consists of instances of other components. The attachments define the connections between any component instances. The attachments at the top level component, that is the architectural level, are connectors.

Attachments in a LEDA architecture can be static or reconfigurable. An example of a static attachment would be $client.request(r) <> server[1].serve(r)$ for the architecture in Figure 12. This attachment is fixed and indicates that *client* will always be connected to $server[1]$. Note that $<>$ is the symbol for attachment. Reconfigurable attachments are used to support the notion of dynamism. The attachment in Figure 12 is an example of a simple dynamic connection. In this attachment $server[1]$ is used if its workload is less than that of $server[2]$, otherwise $server[2]$ is used. LEDA also provides validation support to ensure that attachments support the behavior specified in the roles of components being attached.

In addition to the specification of software architectures, LEDA also supports prototype execution. For example, the Mobility Workbench (MWB) [VM94], a π-calculus tool for analyzing concurrent systems can be used to execute a LEDA specification since the underlying formalism is π-calculus.

7.4 PiLar

PiLar *I*s a *L*anguage for *AR*chitectural description that allows for the representation of hierarchical systems that reconfigure dynamically. A system in PiLar can have two types of components: single components and composite components. Single components have a set of one or more interface ports which define outgoing interaction points. Composite components can contain both interface ports and instances of single or other composite components. In addition to interface definitions and component composition, a component can also have a set of constraints defined in CCS. It is within the constraints that reconfiguration is defined. PiLar has two types of primitive: operators (refer to components) and operations (perform reconfigurations) [CdlFBS01].



(a) graphical representation of client-server reconfiguration

component *client*
[**port**C_1]

component *server*
[**port**S_1]

component *system*
[$C : client \mid S : server \mid$
$L : inter[C.C_1 \mid S.S_1]$]

component *comm*
[**port**A | **port**B]
constraint
$[Q_1 \triangleq \quad \alpha.A(x).\overline{\alpha.B}(x).Q_1 \mid$
$\alpha.B(x).\overline{\alpha.A}(x).Q_1]$

component *inter*
[**port**A | **port**B]
constraint
$[Q_2 \triangleq \quad \rho R_1(\alpha : comm[\alpha.A \mid \alpha.B]).\alpha.B(H).$
$\delta R_1.\rho R_2(\alpha : tier[\alpha.A \mid \alpha.B])]$

component *tier*
[**port**A | **port**B]
$[P : proxy \mid L1 : comm[A \mid P.CL] \mid$
$L2 : comm[P.SV \mid B]]$

component *proxy*
[**port**CL | **port**SV]

Base-level: [*system*]

(b) representation of client-server in PiLar

Figure 13: Client-server example using the PiLar language [CdlFBS01]

To demonstrate the use of PiLar we will show the specification of a client-server example (see Figure 13). Notice that there are two constraints specified in this example: the *inter* component constraint and the *comm* component constraint. The *inter* component constraint is as follows:

$$Q_2 \triangleq \quad \rho R_1(\alpha : comm[\alpha.A \mid \alpha.B]).\alpha.B(H).\delta R_1.\rho R_2(\alpha : tier[\alpha.A \mid \alpha.B])$$

In this constraint we see that an operator primitive (α) is used to refer to the current component while two operation primitives are used to refer to the action of creation (ρ) and removal (δ). Understanding these primitives and basic CCS we can now read the constraint as: "Reify (ρ) the avatar (α) with the archtype *comm* connecting it with the required ports ($\alpha.A, \alpha.B$). Then wait for a message H to come from the server side ($\alpha.B$). When such a message arrives, destroy (δ) the previous reification (R_1), and reify it again with the archtype *tier* ($\rho R_2(\alpha : tier[\alpha.A \mid \alpha.B])$)" [CdlFBS01].

Recent extensions to the initial definition of PiLar ([CdlFBSB02a, CdlFBSB02b]) have redefined the semantics in π-calculus and have developed a programming language style constraint language that is more usable. We do not consider these extensions in our survey and classification.

8 Logic-Based Formalisms

In the section we outline approaches that use logic as a formal basis. Specifically, we discuss the Generic Reconfiguration Language (Gerel) which is based on first-order logic, the Aguirre-Maibaum approach which uses first-order logic and temporal logic, and ZCL which uses the Z specification language based on predicate logic and set theory.

8.1 Generic Reconfiguration Language (Gerel)

Gerel was originally part of the Reconfigurable and Extensible Parallel and Distributed Systems (REX) project. The description of Gerel as a *generic* language comes from its ability to represent both ad-hoc and programmed dynamism (see Section 2 for definitions). In the early 1990s when Gerel was developed, many other configuration level languages supported ad-hoc change (e.g. Conic, Polylith/MIL, Lady) or programmed dynamism (e.g. Durra, PRONET) but not both [EW92].

Gerel is an excellent example of a configuration language designed specifically to be used in conjunction with a programming language. The relationship between the programming language and the configuration language in Gerel is that the programming language is used for program components while the configuration language is used for configuration components which are composed of program components and other configuration components.

Dynamic change in Gerel is specified in a configuration component as a change script using the keyword **change**. A Gerel change script can have the following commands [End94]:

- *Basic commands:* define reconfiguration actions.

  create *component* **of** *comp_type* ["(" *arguments* ")"][**at** *location*]

  delete *component*

  link *port port*

  unlink *port port*

- *Structured commands:* define control flow.

  select *genericsymbol* ":" *formula* **do** *commands* **end**

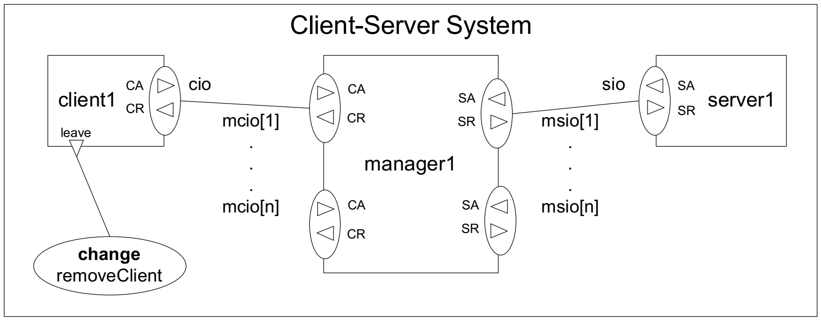
  forall *genericsymbol* ":" *formula* **do** *commands* **end**

  iterate *i* **in** "[" *low* ":" *high* "]" **do** *commands* **end**

The *genericsymbol*s used in the **select** and **forall** commands can be component instances, component types, sets of ports or type of port [EW92].

Figure 14(b) shows a client-server system in the Gerel language. In the configuration component, *client-server_system*, a script *removeclient* is provided. This script unbinds *client1* from a manager and then removes it from the system using the **delete** operation. The script is invoked by a boolean port *leave* in the client component. This connection can be seen in the graphical representation of the system in Figure 14(a).

As previously mentioned Gerel was designed for use with a programming language. For example, the Gerel scripts have been used inside a configuration management tool, built on top of Conic, called `gerel`. Conic is an environment for constructing distributed systems [MKS89]. The configuration manager performs reconfigurations based on Gerel scripts. A script defines the change and can be invoked by a user or by any application process [End94]. If a script is invoked by a user it is ad-hoc change and by an application process it is a programmed change. The manager also contains commands which allow for new components types to be used.



(a) graphical representation

```
unit client-server_interf_types
  type clientio =
    {CA: inport signal;
     CR: output signal result signal};
  type serverio =
    {SR: inport signal;
     SA: output signal result signal};
end.

component client
  use client-server_interf_types: clientio
  interface
       cio: clientio;
    leave: outport signal result bool;
  end;
  /*Component body in prog. language */
end. /*client*/

component server
  use client-server_interf_types: serverio
  interface
    sio: serverio;
  end;
  /*Component body in prog. language */
end. /*server*/

component manager
  use client-server_interf_types: serverio, cli-
entio
  interface
    mcio[1..n]: invert clientio;
    msio[1..n]: invert serverio;
  end;
  /*Component body in prog. language */
end. /*manager*/

component client-server_system
  interface
  /*any interface ports go here*/    end;
  use client, server, manager;
  create client1 of client;
  create server1 of server;
  create manager1 of manager;
  bind client1.cio manager1.mcio[1];
  bind server1.sio manager1.msio[1];
  change removeclient;
    symbol
      j,k portset clientio;
     condition c_type(invoker())=client;
    execute
      /*unlink client connectors*/
      forall j: manager_portset(j) do
        select k: s_linked(k,j) do
          unbind j k
      end;
      delete k;
    end.
  link client1.leave removeclient;
end. /*client-server system*/
```

(b) text representation

Figure 14: Client-server example using Gerel

8.2 Aguirre-Maibaum Approach

Temporal logic is a variant of modal logic. The primary benefit of temporal logic is that it can be used to describe event ordering without explicitly introducing time [CGP99]. Aguirre and Maibuam develop a specification language for reconfiguration of dynamic software architecture with a semantics based on first-order and temporal logic. The language represents component types as classes, connector types as associations and the overall architecture as a subsystem. One unfortunate aspect to this specification approach is that hierarchical composition of components is not possible since subsystems are only composed of component types not other subsystems.

A component type is represented as a class which consists of a class signature C and a finite set of axioms over C. Specifically, the class signature is [AM02b]:

- a set of read variables that allow components to access environment information
- a set of attributes of the component
- a set of actions, possibly with arguments, that take place in the component

The vocabulary of variables, attributes, and actions that are defined in the class signature can all be used in the axioms. The axioms indicate how the possible actions will affect the attributes of the component. An example of a temporal logic axiom is axiom 3 of the *Printer* class in Figure 15. The axiom

$$\forall s \in string : load(s) \rightarrow \bigcirc(job = s)$$

states that if a $load(s)$ action occurs then in the next state s becomes the current *job*. The temporal logic symbols used in this approach are: in the next state ($\bigcirc$), until($\mathcal{U}$), and in some future state ($\Diamond$). In our example we can see the $\bigcirc$ used to represent the next state.

Connector types are represented by associations which identify the participant component types in an interaction as well as the connection between the components. The connection is specified as an action synchronization definition. For example, in Figure 15 the definition

$$SRV.p\text{-}ready \leftrightsquigarrow PR.ready$$

means that the *p-ready* attribute of *SRV* must synchronize with *ready* in the *PR* participant. A synchronization can occur for both attributes and actions of classes.

Finally, subsystems are used to define the initial state of the topology and the reconfiguration operations that can be applied to an already created state. The initial state consists of instantiations of both components and connectors. The connectors accept component instances as arguments which allows for the system to be linked together. Axioms are again used in the subsystem but instead of governing action behavior they govern operation behavior involving the creation and deletion of architectural elements and the reconfiguration of the architectural topology.

The Aguirre-Maibuam approach takes advantage of temporal logic for verification as well as specification. In fact, both class and subsystem properties can be proved using Kripke structures. For example, the property [AM02b]:

$$\forall x, y : USES(x,y) \rightarrow [USES(x,y)\,\mathcal{U}\,(y.job = [])]$$

can be verified to see if a printer that is used by a server will continue to be used by the server until the end of the print job.

Class Printer
Exports print(), load(string), ready
Attributes
ready : boolean
job : string
Actions
print()
load(string)
print-el(char)
Axioms
1. **BEG** $\rightarrow job = []$
2. $\forall s \in string : load(s) \rightarrow job = []$
3. $\forall s \in string : load(s) \rightarrow \bigcirc(job = s)$
4. $print() \rightarrow job \neq []$
5. $\forall j \in string : print() \wedge job = j$
$\rightarrow print\text{-}el(head(j))$
6. $\forall j \in string : print() \wedge job = j$
$\rightarrow \bigcirc(job = tail(j))$
7. $job \neq [] \rightarrow \Diamond(print())$
8. $ready = T \leftrightarrow job = []$

EndofClass

Class Server
Exports enqueue(string), print()
Read Variables
p-ready: boolean
Attributes
p-queue : list(string)
Actions
enqueue(string)
print()
send(string)
Axioms
1. **BEG** $\rightarrow p\text{-}queue = []$
2. $\forall s \in string \forall q \in list(string)$
$: enqueue(s) \wedge p\text{-}queue = q$
$\rightarrow \bigcirc(p\text{-}queue = q + +s)$
3. $print() \rightarrow p\text{-}queue \neq []$
4. $\forall q \in list(string) : print() \wedge$
$p\text{-}queue = q \rightarrow [send(head(q)) \wedge$
$\bigcirc(p\text{-}queue = tail(q))]$
5. $p\text{-}queue \neq [] \rightarrow \diamond(print())$
6. $\forall s \in string : send(s) \rightarrow p\text{-}ready = T$
7. $\forall s \in string : send(s) \rightarrow print()$

EndofClass

Association USES
Participants
SRV : Server
PR : Printer
Connections
$SRV.p\text{-}ready \leftrightsquigarrow PR.ready$
$\forall j \in string : PR.load(j)$
$\leftrightsquigarrow SRV.send(j)$
EndofAssoc

Subsystem Multiple-Printers
Initial State
S : Server
P_1 : Printer
USES(S, P_1)
Operations
change(y: Printer)
add(y: Printer)
Axioms
1. $\forall s, p_1, p_2 : USES(s, p_1)$
$\wedge USES(s, p_2) \rightarrow p_1 = p_2$
2. $\forall p : change(p) \rightarrow \bigcirc(USES(S, p))$
3. $\forall p : \neg change(p) \rightarrow$
$[\forall x, y : USES(x, y) \leftrightarrow \bigcirc USES(x, y)]$
4. $\forall p : change(p) \rightarrow (S.p\text{-}ready = T)$
5. $\forall p : add(p) \leftrightarrow$
$[(\neg Printer(p)) \wedge \bigcirc Printer(p)]$
6. $\forall s : Server(s) \leftrightarrow \bigcirc Server(x)$
7. $\forall p : Printer(p) \rightarrow \bigcirc(Printer(p))$

EndofSubsystem

Figure 15: Client-server example using Aguirre-Maibaum approach [AM02b]

8.3 ZCL Framework

CL is a distributed configuration language that has a model to support the following abstractions for the representation of software architectures: modules (components), ports, instances, connections, and configurations. A system configuration, also known as a top component, is written in the CL language and can consist of task components written in a programming language and composite components consisting of a group of task components and possibly other composite components. In the CL model a dynamic software architecture with these abstractions is constructed as follows [RJC00]:

1. A set of components are stored in the component library.
2. Components used in the current system are selected from the library.
3. Instances of the selected components are created.
4. The component instances are linked together.
5. The component instances are activated.

Systems developed using the CL model also have special components called configuration components that, unlike task components, perform operations that affect the topology of the system [dP99]. Specifically

---

ExampleClientServer

$$\Xi ECL\_Exec\_State$$
$$\Xi CL\_Connection$$

$$
\begin{array}{l}
\#children(Server) \geq 1 \\
\#children(Client) = 1 \\
\exists Srv : Nodes \mid Srv \in children(Server) \bullet Srv \in ActiveInst \wedge tab\_inst(Srv) \neq unborn \wedge \\
(\forall Clt : Nodes \mid Clt \in children(Client) \bullet \exists conn : CL\_Connection \bullet \\
conn = \theta CL\_Connection[receiver := (Srv, provide), sender := (Clt, require)] \wedge conn \in connection)
\end{array}
$$

Not_ExampleClientServer

$$\Xi ECL\_Exec\_State$$
$$\Xi CL\_Connection$$

$$
\begin{array}{l}
\#children(Server) \leq 1 \vee \\
\#children(Client) \neq 1 \vee \\
\exists Srv : Nodes \mid Srv \notin children(Server) \bullet Srv \in ActiveInst \vee tab\_inst(Srv) \neq unborn \vee \\
(\forall Clt : Nodes \mid Clt \in children(Client) \bullet \exists conn : CL\_Connection \bullet \\
conn = \theta CL\_Connection[receiver := (Srv, provide), sender := (Clt, require)] \wedge conn \notin connection)
\end{array}
$$

$$CS \widehat{=} (ExampleClientServer \wedge Success) \vee (Not\_ExampleClientServer \wedge Failure)$$

Figure 16: State schema for the top component of a client-server example in ZCL [dP99]

---

configuration components can build the initial system, monitor the status of an existing configuration (using configuration operations), and reconfigure an existing system (using reconfiguration and synchronized reconfiguration operations).

The representation of a system and the method of composing and reconfiguring a system form the CL model. ZCL is based on this model and is specified in the Z language [dPJC98]. Z is a formal specification language that is based on predicate logic and set theory [Dil94]. In the ZCL framework, Z schemas are used to specify the CL model abstractions such as modules (components), ports, instances, connections, and configurations. That is, programming languages are no longer used for the specification of the task components and the CL language is no longer used for the specification of configurations and operations.

ZCL distinguishes between two types of schemas in the specification of dynamic architectures [dP99]:

1. *State schemas:* represent the topology or structure of architecture.
2. *Operation schemas:* represent the configuration and reconfiguration operations that can be applied by a configuration component to the architecture.

An example of the top component in a client-server system can be seen in Figure 16. The client-server system defined in the state schema has one client instance and possibly many server instances. Examples of operation schemas can be found in [dP99].

The ZCL framework not only allows for the specification of dynamic software architectures but also the specification of an execution model based on state machines. The execution model and a set of configuration commands are used to govern the dynamic change. ZCL specifications can be used to simulate run-time behavior and analyze pre-defined conditions and invariant constraints in the context of system configurations. Currently, the simulation and analysis are done using the Z-EVES theorem prover [Saa99].

9 Other Formalisms

In this section we outline other formal specification approaches to dynamic software architectures that do not have a semantics based on graph theory, process algebra, or logic. There are two approaches that fit this category: C2SADEL and RAPIDE.

9.1 C2SADEL

The C2SADEL supports the specification of software architectures developed using the C2 architectural style [OT98]. In the C2 architectural style, components have a defined top and bottom, have multiple threads of control and encapsulate functionality and behavior (see Figure 17). Connectors also have a defined top and bottom that represent component interaction. Asynchronous message passing is used for communication. An important property of the C2 architectural style is that it supports substrate independence. That is, "a component within the hierarchy can only be aware of components 'above' it and is completely unaware of components which reside at the same level or 'beneath' it" [OT98]. Component can invoke services offered by components above via explicit request messages. Communication to components below occurs implicitly. For example, a state change in a component is announced via a message broadcast to all components and connectors joined to its bottom end.

In C2SADEL the formal specification of C2 architectures is based on four levels of abstraction [MTW96]: component functionality, component interfaces, architectural configuration (and reconfiguration), and architectural style rules. First, we will outline the specification of component interfaces and architectural configuration (and reconfiguration). Second, we will explain how all four levels of abstraction are used in the context of a tool suite, ArchStudio.

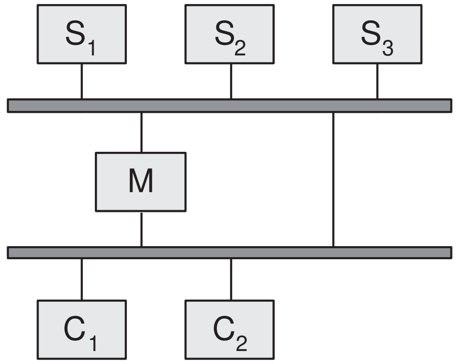


Figure 17: A C2 client-server system

The above diagram outlines the structure of an example client-server system with a C2 architectural style. Boxes represent components (C=client, M=manager, S=server), dark bars represent connectors, and the lines represent communication links. Component-connector links as well as connector-connector links are allowed provided that top-bottom pairs are connected. Component-component links are not allowed in a C2 architecture.

Component interfaces are defined in the Interface Definition Language (IDL). Figure 18 shows the definition of the *Server* component's interface in this language. Since each component has a top and a bottom, interface requests and notification are defined for each. Note that for the top interface, requests go out and notification come in while for the bottom interface, notifications go out and requests come in. The interface can also specify some of the component behavior such as method invocations and message generation. This behavior can be specified for the component at startup, cleanup and in response to received messages. In our *Server* example we see that when the *m1* message is received, *method1* is invoked and the *method1Called* event is always generated. An event can always be generated (**always_generated**) or sometimes generated (**may_generated**) in response to a message being received.

Architectural configuration in a C2 system is specified in the Architecture Description Language (ADL). This language supports the composition of component and connectors. For example, in Figure 18, we have an architecture that consists of three components and two connectors. The topology section defines how the components and connectors interconnect through top and bottom ports. Ports can also have different filter mechanisms depending on the system. In our case we use the *no_filtering* option which means that the port passes messages through directly. Other options for filtering include *notification_filtering*, *prioritized*, and *msg_sink*. In [c2s] the architecture description includes the definition of a system that implements a particular architecture. The definition uses the **is_bound_to** keyword to relate conceptual components defined in the ADL with concrete component implementations.

The definition of components and connectors and the specification of the system topology allow for the specification of static C2 systems. To specify reconfiguration the C2 architecture modification (sub)language (AML) is used. The C2 AML supports the following reconfiguration operations **addComponent**, **removeComponent**, **weld**, and **unweld** [MT00]. In the client-server example in Figure 18 we can see the specification of a new manager component being added and connected to existing connectors as well as the existing manager component being disconnected from connectors and removed from the system.

ArchStudio is a tool that allows the development of C2 architectures in Java that can be modified at runtime [MT00]. ArchStudio links an architectural model specified in the C2SADEL's IDL and ADL with a Java implementation. Change scripts are specified in the C2SADEL's AML that can be applied to the model.

IDL:

```
component Server is
  interface
    top_domain is
      in
        null;
      out
        null;
    bottom_domain is
      in
        ...
      out
        method1Called (value: some_val);
        ...
  parameters
    null;
  methods
    function method1() return some_val;
...
  behavior
     received_message m1;
      invoke_methods method1;
      always_generate method1Called
  context
    ...
end Server;

component Manager is
  ...
end Manager;

component ...
```

ADL:

```
architecture ClientServerArchitecture is {
  components {
    top_most
      Server;
    internal
      Manager;
    bottom_most
      Client;
  }
  connectors {
    Conn1;
    Conn2;
  }
  topology {
    connector Conn1 connections {
      top_ports {
        Server filter no_filtering;
      }
      bottom_ports {
        Manager filter no_filtering;
      }
    connector Conn2 connections {
      top_ports {
        Manager filter no_filtering;
      }
      bottom_ports {
        Client filter no_filtering;
      }
    }
  }
}
```

AML :

```
ClientServerArchitecture.AddComponent(Manager2);
ClientServerArchitecture.Weld(Conn1, Manager2);
ClientServerArchitecture.Weld(Manager2, Conn2);

ClientServerArchitecture.Unweld(Conn1, Manager);
ClientServerArchitecture.Unweld(Manager, Conn2);
ClientServerArchitecture.RemoveComponent(Manager);
```

Figure 18: C2SADEL client-server example (based on a stack example in [Med96, c2s])

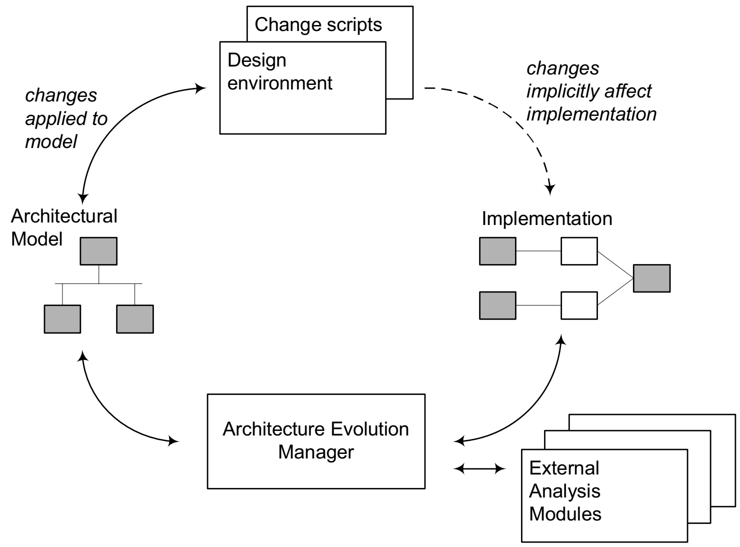


Figure 19: ArchStudio tool suite [OT98]

The ArchStudio tool suite is developed for use with the C2SADEL *and a Java implementation of a C2 system.*

The user of the tool also has the option to use interactive tools such as Argo to specify reconfiguration. When a reconfiguration is initiated, the architectural evolution manager determines if the change is valid. It checks if the change violates the C2 architectural style. The reconfiguration is not applied to the model until the evolution manager accepts the change. Once a change has been applied to the model, the change is applied to the implementation. The relationship between the conceptual model and the implementation can be specified in the C2SADEL using the **is_bound_to** keyword. Figure 19 shows the basic structure of ArchStudio.

9.2 RAPIDE

RAPIDE is an executable architecture definition language (EADL) [LV95]. In RAPIDE a system is specified using five main sublanguages:

1. **Types language.** Used for the specification of module (i.e. component) interfaces.
2. **Architecture language.** Used for the declaration of a set of components and the specification of event communication through a connection definition.
3. **Specification language.** Used to specify component behavioral constraints. Constraints can occur in an interface or an architecture. Specifically, an interface constraint constrains the execution of modules that implement the interface type while an architecture constraint is applied to the internal execution of the architecture [LKA+95].
4. **Executable language.** Used for the specification of modules. A module is concurrent and reactive - in response to an observed event a module will execute a portion of code and may also announce additional events.

```
type Server is interface
  provides
    ...
  requires
    ...
  in action SR_Receive
  out action SA_Send
  behavior
    ...
  constraint
    ...
end Server;

type Manager is interface
  provides
    function serverAvailable return Server;
    function isCurrent return Server;
  ...
end Manager;
```

```
type Client is interface
  ...
end Client;

with Client, Manager, Server; architecture ClientServer
is
    ?C: Client;
    ?M: Manager;
    ?D: Data;
    S: Server;
    ...
  connections
    ?C.CR_Send(?D) to ?M.CR_Receive(?D)
    ?M.CA_Send(?D) to ?C.CA_Receive(?D)
    ?M.SR_Send(?D) where ?M.serverAvailable(S)
          ||> S.SR_Receive(?D)
    S.SA_Send(?D) where ?M.isCurrent(S)
          ||> ?M.SA_Receive(?D)
    ...
end ClientServer;
```

Figure 20: RAPIDE client-server example using **where** operator

5. **Pattern language.** A pattern is specified to identify subsets of the partially ordered event set (poset) of computations produced from simulation.

In RAPIDE a software architecture is called a framework and it is the first step in defining a RAPIDE system. The framework consists of module interfaces and a specification of the communication in the form of connector rules and communication constraints. A framework or architecture can be instantiated into a system module-by-module [LKA+95].

If no module is currently available to instantiate the interface the module behavior can be specified in the architecture language. Thus at any point in the instantiation, analysis can occur in the form of simulation and conformance checking. The output resulting from a simulation is a poset − a record of the program behavior that reflects the causal ordering of events.

Figure 20 shows an example of a client-server system in RAPIDE. The example has three types of components (Server, Manager, Client) and defines an interface for each. The interface defines functions that are to be provided and required by the interface, event actions that are generated (**out action**) and observed (**in action**), behavior, and constraints. An architecture is also defined which connects the component instances. Basic connections are defined by connecting the in and out action events of component interfaces using the **to** operation. An instance can be explicitly named (e.g. S:Server) or a placeholder can be used to represent an instance of a component type (e.g. ?C:Client).

One way to represent dynamism in RAPIDE is to use the **where** operator, from the pattern language, in the connection section of an architecture definition [MT00]. For example in our client-server example the following dynamic connection is defined:

?M.SR_Send(?D) **where** ?M.serverAvailable(S) ||> S.SR_Receive(?D)

```
CreateModule(type : ModuleType, parent : Event, name : String);
DeleteModule(module : Event);
CreatePathway(inputs : Pattern, outputs : Pattern, name : String);
DeletePathway(pathway : Event);
ChangeParent(module : Event, parent : Event);
AddPathwayInputs(pathway : Event, inputs : Pattern);
AddPathwayOutputs(pathway : Event, outputs : Pattern);
DeletePathwayInputs(pathway : Event, inputs : Pattern);
DeletePathwayOutputs(pathway : Event, outputs : Pattern);
```

Figure 21: Execution Architecture Events in RAPIDE

This connection defines that some manager which identifies that server S is available will connect to it. The connection type is defined using a broadcast connection rule (||>) or alternatively, the connection rule could also be a pipe (=>) [LV95].

A recent extension to RAPIDE provided support for component and connector addition and removal [VPL99]. Execution architecture events (Figure 21) were added to RAPIDE as primitive events. These events are treated as being equivalent to normal events within RAPIDE. Specifically there is one event type for each of the basic reconfiguration operations. Additionally, there is an event to change the parent of a module and four other events which allow for changing the inputs and outputs of a connector without replacing the connector with a new one.

10 Evaluation of Dynamic Software Architecture Specifications

10.1 Change Type

The type of change supported focused on the reconfigurations operations and architectural element variability.

Our comparison of the change types supported by different dynamic architecture specification approaches shows that the majority of approaches support all of the basic change operations. For example, in RAPIDE there is one execution architecture event type for each of the basic reconfiguration operations. Approaches that did not support all of the basic operations include several of the process algebra approaches (Darwin, LEDA) that provide limited specification support, especially 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.

Almost all of the approaches considered provide support for composite operations. An example of support for composite operations can be found in C2SADEL where the AML can contain combinations of **AddComponent**, **RemoveComponent**, **Weld**, and **Unweld** operations (see Figure 18). Note, that C2SADEL does not explicitly support the addition and removal of connectors. The **Weld** and **Unweld** operations allow for the reconfiguration of the communication links between components and connectors. Since multiple components can be connected to the same connector we can see that connector removal, for example, occurs implicitly when all components connected to a connector are unwelded.

Although most approaches support basic and composite operations, only a few of the approaches provide full support for composite operation constructs such as sequencing, choice, and iteration. The scripts used

		Basic Reconfiguration Operations				Composite Operations				Set of Arch. Elements	
							Operation Constructs				
		Component Addition	Component Removal	Connector Addition	Connector Removal	Basic Support	Sequencing	Choice	Iteration	Fixed	Variable
Graph	Le Métayer approach	●	●	●	●	●	○	●	○	●	○
	Hirsch et al. approach	●	●	●	●	●	○	?	○	●	○
	Taentzer et al. approach	●	●	●	●	●	○	○	○	●	●?
	CommUnity	●	●	●	●	●	●	●	●	●	○
	CHAM	●	●	●	●	●	i	i	i	●	○
Process Algebra	Dynamic Wright	●	●	●	●	●	●	●	●	●	○
	Darwin	●	⦿	–	–	●	⦿?	⦿?	⦿?	●	○
	LEDA	●	○	●	○	○	○	●	○	●	○
	PiLar	●	●	●	●	●	●	●	○	●	○
Logic	Gerel	●	●	●	●	●	●	●	●	●	⦿
	Aguirre-Maibaum approach	●	●	●	●	●	?	●	?	●	○
	ZCL	●	●	●	●	●	?	?	?	●	○
Other	C2SADEL	●	●	i	i	●	●	○	○	●	?
	Rapide	●	●	●	●	●	?	●	○	●	?

KEY:

- ● = supported by specification explicitly
- i = supported by specification implicitly
- ⦿ = supported externally (i.e. tool, language, etc)
- ○ = not supported
- ? = support unknown
- – = not applicable

Table 3: Classification of formal specifications for dynamic software architectures based on type of change

in CommUnity and Gerel both provide these constructs.

In the context of architectural element variability many of the approaches did not support the use of a variable set of architectural element types. The approaches that directly support variability in the set of architectural elements are primarily distributed system configuration languages such as Gerel which were designed to be used in actual implementations. In many of the other approaches designed for formal analysis, variability of architectural elements can be simulated by expanding the set of fixed element types to include elements that will be added at run-time.

A comparison of all of the approaches with respect to change type is summarized in Table 3.

10.2 Change Process

Our classification of dynamic architecture specification approaches focused on the four primary steps in the change process: initiation, selection, implementation, and assessment.

Most approaches support internal initiation while some support external or both. In some of the approaches (e.g. LEDA) external initiation by an environment application can be simulated by abstracting one level above the top of the architecture. At that level the architecture and environment application can each exist as communicating components. An example of an approach that supported both internal and external initiation was the Le Métayer approach which provided this support through side conditions in the rewriting rules. Approaches such as CommUnity and Gerel support both types of initiation through accompanying tools instead of through the specification language. For example, a tool in the CommUnity approach can allow for a change script to be executed after each computation step.

None of the approaches classified in this paper provided support for unconstrained run-time selection.

		Initiation		Selection				Implementation	Assessment		
		Internal	External	Explicit	Pre-defined	Constrained from Pre-defined set	Unconstrained		Execution/ Simulation	Direct Formal Analysis	Formal Analysis after Translation
Graph	Le Métayer approach	●	●	○	●	●	○	graph rewriting rules	○	●	○
	Hirsch et al. approach	○	○	○	○	○	○	graph rewriting rules	○	○	○
	Taentzer et al. approach	?	?	○	●	○	○	graph rewriting rules	○	●	○
	CommUnity	●	●	◉	●	●	○	category theory	○	◉	○
	CHAM	●	●	○	●	○	○	evolution CHAM reaction rules	○	○	○
Process Algebra	Dynamic Wright	●	○	○	●	●	○	CSP	○	●	●
	Darwin	●	○	○	●	○?	○	π-calculus	●	●	●
	LEDA	●	○	○	●	●	○	π-calculus	◉	○	●
	PiLar	●	○	○	●	○	○	CCS	○	○	●
Logic	Gerel	●	◉	●	●	?	○	first order logic	◉	○	○
	Aguirre-Maibaum approach	●	○	○	●	●?	○	first order logic, temporal logic	○	○	●
	ZCL	●	○	○	●	○	○	Z operation schema	◉	◉	○
Other	C2SADEL	○	◉	●	○	○	○	AML	◉	◉	○
	Rapide	●	○	○	●	●	○	where statement, exec. arch. events	●	●	○

Table 4: Classification of formal specifications for dynamic software architectures based on change process

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 are the CHAM and graph rewriting approaches which allow for random selection of a reconfiguration if multiple possibilities exist, 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 can be found in LEDA. Consider the client-server system in Figure 12. In this example the client is attached to one of the two servers based on the result of a boolean condition.

The implementation of dynamic architectural reconfiguration in formal specification is primarily done using graph rewriting or process algebra. Several logic-based approaches also exist such as Gerel, the Aguirre-Maibaum approach, and ZCL (also based on set theory).

The assessment used for the formal specification approaches varies from direct formal analysis, to simulation, to formal analysis after translation, to no assessment support. Since the approaches we consider all have a formal semantics some analysis is possible and most of the approaches provide some analysis support. For example, the ZCL approach provides simulation and direct formal analysis through the Z/EVES theorem prover, while the Aguirre-Maibaum approach provides analysis of dynamic architectures using Kripke structures. Also, many of the configuration languages such as C2SADEL, Gerel, and Darwin allow for execution as a means of assessment. Examples of approaches that allow for simulation include LEDA and RAPIDE. LEDA can support prototype execution using a π-calculus analysis tool such as the Mobility Workbench. RAPIDE can produce a simulation result as a poset.

The results of the classification based on change process are summarized in Table 4.

10.3 The Infrastructure for Change

In the context of change infrastructure we summarize the type of management used, the separation of concerns between change and computation, the architectural structure, and the architectural element representations.

The management used in most of the specification approaches is centralized, not distributed. This is primarily because early types of dynamic architectural change such as ad-hoc and programmed often had centralized management. Newer definitions of change, such as self-organising architectures, require distributed management.

An example of centralized management can be found in Dynamic Wright where reconfigurations are specified in a configuror or in the Le Métayer approach where reconfiguration rewriting rules are specified in a coordinator. Some of the approaches such as C2SADEL and Gerel do not specify the management but instead allow for the management to be determined in accompanying tools. Some of the approaches such as the Hirsh et al. approach and the Distributed graph approach do not include explicit specification of management. However, centralized management is implicit in these approaches since there is a single graph on which rules can be applied. An example of distributed management can be found in the PiLar language. PiLar allows for multiple components to have constraints which may specify reconfiguration.

Many of the approaches provide separation of concerns between the specification of computation and change. Many provide some separation of the computation and change (partial separation) while others pro-

		Management		Separation of Concerns		Arch. Structure		Architectural Element Representation	
		Centralized	Distributed	Partial	Complete	Architectural Style	System Architecture	Components	Connectors
Graph	Le Métayer approach	●	○	○	●	●	○	nodes of a graph (behavior CSP like)	edges of a graph
	Hirsch et al. approach	i	○	○	●	●	○	edges of a graph with CCS labels	nodes of a graph [point-to-point communication (white nodes) and broadcast communication (black nodes)]
	Taentzer et al. approach	i	○	○	●	○	●	network graph node + local graph and local transformations	Edges of a graph
	CommUnity	●	○	○	●	○	●	a CommUnity program	a star-shaped configuration of programs
	CHAM	●	○	○	●	●	○	molecule	links between two component molecules
Process Algebra	Dynamic Wright	●	○	●	○	○	●	ports (interface) + computation (behavior)	roles (interface) + glue (behavior)
	Darwin	●	●?	○	●	○	●	programming language + component specification of comm. objects	support for simple bindings
	LEDA	●	●?	●	○	○	●	interface specification, composition and attachment specification (if composite)	attachments at top level components
	PiLar	○	●	●	○	○	●	components with ports, instances of other components and constraints	support for simple bindings
Logic	Gerel	⦿	○	●	○	○	●	programming language + Gerel component specification	defined by bind operation in configuration components
	Aguirre-Maibaum approach	●	○	●	○	○	●	class with attributes, actions and read variables	association consisting of participants and synchronization connections
	ZCL	●	○	○	●	○	●	state schema in Z	connection between ports of components
Other	c2SADEL	⦿	○	○	●	●	●	element with top and bottom interface and behavior	element with top and bottom ports and filtering mechanisms
	Rapide	●	●	●	○	○	●	types language for component interface (plus other sublanguages for behavior)	broadcast connection rule (||>) or pipe (=>)

Table 5: Classification of formal specifications for dynamic software architectures based on change infrastructure

vide separation of concerns into different distinct parts of the specification (full separation). For instance, the Aguirre-Maibaum approach features partial separation because axioms in the subsystem description describe reconfiguration and axioms in the class (component) description describe behavior. The Distributed graph approach, the Hirsh et al. approach, and the Le Métayer approach provide complete separation because the rewriting rules specifying change are separate from the specification of behavior. Another example of complete specification is C2SADEL where behavior is specified in the ADL and a programming language while reconfiguration is specified in the AML.

In the context of architectural structure, the majority of approaches only represented the system architecture and did not have any explicit representation of architectural styles. Exceptions include some of the graph rewriting approaches such as the Le Métayer approach, Hirsh et al. approach, and CHAM that provide a set of rules which define the architectural style for a class of systems. For example recall the creation CHAM for client-server systems in Figure 8.

The representation of the components and connectors varies from approach to approach. For example, the use of the COMMUNITY programming language in the COMMUNITY approach or the use of a notation similar to CSP in the Le Métayer approach are possible representations of architectural elements. Most techniques represented components and connectors as first-class entities. The complete results of the classification based on the infrastructure for change, including the representation of components and connectors, are summarized in Table 5.

11 Conclusions and Open Problems

In this paper we address the need for an evaluation of dynamic software architectures and present an evaluation approach that answers three fundamental questions related to dynamic architectural change: What type of change is supported? What kind of process implements the change? What infrastructure is available to support the change process? The paper summarizes the result of classifying 14 of the most popular approaches to dynamic architectural specification.

Our classification shows that the area of dynamic software architecture specification is well researched. There exist a lot of different sometimes conflicting notations, concepts, and definitions. Although a variety of approaches exist, there has been a lack of practical evaluation. Some exceptions include the configuration programming approaches. Practical evaluation would aide in understanding what notations, concepts, and definitions are important. For example, in Table 3 we see that a significant group of the approaches do not support operation constructs in the specification of composite operations and the majority of approaches do not support variable sets of architectural elements. More evaluation would inform us on the necessity of support for these criteria. In Table 4 the fact that most approaches only support a pre-defined selection of a reconfiguration operation may or may not be insufficient depending on how selection in practical systems occurs. Finally in Table 5, many of the current approaches provide little direct support for analysis of dynamic architectural change within a system. A practical evaluation of the approaches would help to better understand what type of analysis is needed.

Another problem related to the study of practical examples is domain-specific concerns. The current specification languages surveyed are applied to distributed systems or software architecture in general. Domain specific questions need to be addressed. For example, Are the approaches surveyed capable of specifying domain-specific architectures (e.g. dynamic component-based web applications)?

Most of the specification languages classified represent structure, behavior, and reconfiguration primarily in a text-based form. However, approximately half of the approaches also include either formal or informal graphical representations. Many of the graphical representations used rely on sequences of static diagrams. As the problem of dynamic architectural change becomes more understood we need to determine how to best represent dynamic change graphically. The importance of graphical architecture specification is evident in the additional support for static software architectures in UML 2.0. Gaining more insight into graphical representations of dynamic architectures would be beneficial in extending an existing standard such as UML.

A final area that requires future research is the relationship between architectural structure and dynamic change. Most of the current approaches focus on the relationship between a given system's architectural structure and dynamic change. The relationship between an architectural style and dynamic change is not as well studied. Specifically in Table 5 we see that 4 of the approaches classified represent the architectural structure explicitly while 11 do not. More work on this relationship between style and change could lead to a better understanding of the limitations placed on change types by a given style. Furthermore, understanding these limitations could provide opportunities for analysis that do not rely on a particular implementation. Thus providing prescriptive change support instead of diagnostic change support.

12 Acknowledgements

This work benefited from the supervision of James Cordy and Juergen Dingel and from discussions with Michel Wermelinger (Departmento de Informática, Universidade Nova de Lisboa, Portugal). This work was funded by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[ADG97] Robert Allen, Rémi Douence, and David Garlan. Specifying dynamism in software architecture. In *Proceedings of Foundations of Component-Based Systems Workshop*, 1997.

[ADG98] Robert Allen, Rémi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. *Lecture Notes in Computer Science*, 1382, 1998.

[All97] Robert J. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, May 1997.

[AM02a] Nazareno Aguirre and Tom Maibaum. A temporal logic approach to component-based system specification and reasoning. In *Proceedings of the* 5^{th} *ICSE Workshop on Component-Based Software Engineering*, Apr. 2002.

[AM02b] Nazareno Aguirre and Tom Maibaum. A temporal logic approach to the specification of reconfigurable component-based systems. In *Proceedings of the* 17^{th} *IEEE International Conference on Automated Software Engineering (ASE 2002)*, pages 271–274, Sept. 2002.

[AM03a] Nazareno Aguirre and Tom Maibaum. A logical basis to the specification of reconfigurable component-based systems. In *Proceedings of Fundamental Approaches to Software Engineering (FASE 2003)*, Apr. 2003.

[AM03b] Nazareno Aguirre and Tom Maibaum. Some institutional requirements for temporal reasoning about dynamic reconfiguration. In *Proceedings of Symposium on Verification: Theory and Practice*, Jul. 2003.

[And00] Jesper Andersson. Issues in dynamic software architectures. In *Proceedings of the* 4^{th} *International Software Architecture Workshop (ISAW4)*, June 2000.

[BB92] Gérard Berry and Gérard Boudol. The chemical abstract machine. In *Selected papers of the Second Workshop on Concurrency and Compositionality*, pages 217–248. Elsevier Science Publishers Ltd., 1992.

[BMZ+04] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. Towards a taxonomy of software change. *Journal of Software Maintenance and Evolution: Research and Practice*, 2004. To appear.

[c2s] C2 software architecture description language. Web page: `http://www.isr.uci.edu/architecture/adl/SADL.html`.

[CdlFBS01] Carlos E. Cuesta, Pablo de la Fuente, and Manuel Barrio-Solórzano. Dynamic coordination architecture through the use of reflection. In *Proceedings of* 16*th* *ACM Symposium on Applied Computing (SAC 2001)*, pages 134–140, Mar. 2001.

[CdlFBSB02a] Carlos E. Cuesta, Pablo de la Fuente, Manuel Barrio-Solórzano, and M. Encarnación Beato. Coordination in a reflective architecture description language. In *Proceedings of the* 5*th* *International Conference on Coordination Models and Languages (COORDINATION 2002)*, pages 141–148. Springer-Verlag, 2002.

[CdlFBSB02b] Carlos E. Cuesta, Pablo de la Fuente, Manuel Barrio-Solórzano, and M. Encarnación Beato. Introducing reflection in architecture description languages. In *Proceedings of Working IEEE/IFIP Conference on Software Architecture (WICSA 2002)*, pages 143–156, 2002.

[CGP99] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, 1999.

[CHK+01] Ned Chapin, Joanne E. Hale, Khaled Md. Khan, Juan F. Ramil, and Wui-Gee Tan. Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13(1):3–30, Jan./Feb. 2001.

[Cle96] Paul Clements. A survey of architecture description languages. In *Proceedings of the* 8*th* *International Workshop on Software Specification and Design (IWSSD'96)*, Mar. 1996.

[CPT98a] Carlos Canal, Ernesto Pimentel, and Jos M. Troya. Compatibility, inheritance and extension of π-calculus agents. Technical Report LCC-ITI-98-13, Universidad de Málaga, June 1998.

[CPT98b] Carlos Canal, Ernesto Pimentel, and Jos M. Troya. A π-calculus semantics for an architecture description language. Technical Report LCC-ITI-98-17, Universidad de Málaga, April 1998.

[CPT99] C. Canal, E. Pimentel, and J. M. Troya. Specification and refinement of dynamic software architectures. In *Software Architecture*, pages 107–126. Kluwer Academic Publishing, February 1999.

[Dar] The Darwin compiler v1.0.2. Web page: `http://www.doc.ic.ac.uk/~igeozg/Project/Darwin/DarwinSyntax.pdf`.

[Dar97] The Darwin language, version 3d. Web page: `http://www-dse.doc.ic.ac.uk/Software/Darwin/darwin-lang.pdf`, Sept. 1997.

[DC95] Thomas R. Dean and James R. Cordy. A syntactic theory of software architecture. *Special Issue on Software Architecture, IEEE Transactions on Software Engineering*, 21(4):302–313, January 1995.

[Dil94] Antoni Diller. *Z: An Introduction to Formal Methods*. John Wiley and Sons, 1994.

[dP99] Virginia Carvalho Carneiro de Paula. *ZCL: A Formal Framework for Specifying Dynamic Software Architectures*. PhD thesis, Federal University of Pernambuco, 1999.

[dPJC98] Virginia C. de Paula, G. R. Ribeiro Justo, and P. R. F Cunha. Specifying dynamic distributed software architectures. In *Proceedings of the XII Software Engineering Brazilian Symposium*, pages 7–22. IEEE Computer Society Press, Oct. 1998.

[dPJC00] Virginia C. de Paula, G. R. Ribeiro Justo, and P. R. F Cunha. Specifying and verifying reconfigurable software architectures. In *Proceedings of the* 5^{th} *International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE 2000)*, pages 21–31, Jun. 2000.

[End94] M. Endler. A language for implementing generic dynamic reconfigurations of distributed programs. In *Proceedings of* 12^{th} *Brazilian Symposium on Computer Networks*, pages 175–187, 1994.

[EW92] M. Endler and J. Wei. Programming generic dynamic reconfigurations for distributed applications. In *Proceedings of the International Workshop Configurable Distributed Systems*, pages 68–79, 1992.

[FM97] Jose Luiz Fiadeiro and T. S. E. Maibaum. Categorical semantics of parallel program design. *Science of Computer Programming*, 28(2-3):111–138, 1997.

[FWM99] José Luiz Fiadeiro, Michel Wermelinger, and José Meseguer. Semantics of transient connectors in rewriting logic. In *Proceedings of the* 1^{st} *Working International Conference on Software Architecture*, Feb. 1999.

[GMK02] Ioannis Georgiadis, Jeff Magee, and Jeff Kramer. Self-organising software architectures for distributed systems. In *Proceedings of the* 1^{st} *Workshop on Self-Healing Systems*, pages 33–38, 2002.

[GMW97] David Garlan, Robert T. Monroe, and David Wile. Acme: An architecture description interchange language. In *Proc. of CASCON'97*, pages 169–183, Nov. 1997.

[HIM98] Dan Hirsch, Paola Inverardi, and Ugo Montanari. Graph grammars and constraint solving for software architecture styles. In *Proceedings of the* 3^{rd} *International Software Architecture Workshop (ISAW3)*, pages 69–72, November 1998.

[HIM99] Dan Hirsch, Paola Inverardi, and Ugo Montanari. Modeling software architectures and styles with graph grammars and constraint solving. In *Proceedings of the* 1^{st} *Working IFIP Conference on Software Architecture (WICSA1)*, pages 127–142, February 1999.

[Hir03] Dan Hirsch. *Graph Transformation Models for Software Architecture Styles*. PhD thesis, Universidad de Buenos Aires, 2003.

[HM99] Dan Hirsch and Ugo Montanari. Consistent transformations for software architecture styles of distributed systems. *Electronic Notes in Theoretical Computer Science*, 28:23–40, September 1999.

[HM00] Dan Hirsch and Ugo Montanari. Higher-order hyperedge replacement systems and their transformations: Specifying software architecture reconfigurations. In H. Ehrig and G. Taentzer, editors, *Proceedings of the Joint APPLIGRAPH/GETGRATS Workshop on Graph Transformation Systems (GRATRA 2000). Satellite Event of the European Joint Conference on Theory and Practice of Software (ETAPS 2000)*, March 2000.

[HNS99] C. Hofmeister, R. Nord, and D. Soni. Describing software architecture with UML. In P. Donohoe, editor, *Proceedings of Working IFIP Conference on Software Architecture*, pages 145–160. Kluwer Academic Publishers, Feb. 1999.

[IW95] Paola Inverardi and Alexander L. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine model. *IEEE Transactions on Software Engineering*, 21(4), 1995.

[IWY97] P. Inverardi, A.L. Wolf, and Daniel Yankelevich. Checking assumptions in component dynamics at the architectural level. In *Proceedings of the 2nd International Conference on Coordination Models and Languages (COORD '97)*, pages 46–63, 1997.

[KM98] J. Kramer and J. Magee. Analysing dynamic change in software architectures: A case study. In *Proceedings of the* 4^{th} *International Conference on Configurable Distributed Systems*, pages 91–100, 1998.

[Kra90] Jeff Kramer. Configuration programming - a framework for the development of distributable systems. In *Proceedings of the IEEE Internatinal Conference on Computer Systems and Software Engineering (COMPEURO 90)*, pages 374–384, May 1990.

[LKA$^+$95] David C. Luckham, John L. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, Apr. 1995.

[LS80] B. P. Lientz and E. B. Swanson. *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. Addison-Wesley, 1980.

[LV95] David C. Luckham and James Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, Sept. 1995.

[MBZR03] Tom Mens, Jim Buckley, Matthias Zenger, and Awais Rashid. Towards a taxonomy of software evolution. In *Proceedings of* 2^{nd} *International Workshop on Unanticipated Software Evolution (USE 2003)*, Apr. 2003.

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In W. Schafer and P. Botella, editors, *Proceedings* 5^{th} *European Software Engineering Conference (ESEC 95)*, volume 989, pages 137–153, Sitges, Spain, 1995. Springer-Verlag, Berlin.

[MDK93] Jeff Magee, Naranker Dulay, and Jeffrey Kramer. Structuring parallel and distributed programs. 8(2):73–82, Mar. 1993.

[MDK94] Jeff Magee, Naranker Dulay, and Jeff Kramer. A constructive development environment for parallel and distributed programs. *IEE/IOP/BCS Distributed Systems Engineering*, 1(5):304–312, September 1994.

[Med96] Nenad Medvidovic. ADLs and dynamic architecture changes. In *Joint Proceedings of the* 2^{nd} *International Software Architecture Workshop (ISAW-2) and International Workshop on Multiple Perspectives in Software Development (Viewpoints '96) on SIGSOFT '96 Workshops*, pages 24–27. ACM Press, 1996.

[Mét96] Daniel Le Métayer. Software architecture styles as graph grammars. In *Proceedings of the* 4^{th} *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 15–23. ACM Press, 1996.

[MK96] Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. In *Proceedings of the* 4^{th} *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 3–14. ACM Press, 1996.

[MKS89] Jeff Magee, Jeff Kramer, and Morris Sloman. Constructing distributed systems in Conic. *IEEE Transactions on Software Engineering*, 15(6), 1989.

[MORT96] Nenad Medvidovic, Peyman Oreizy, Jason E. Robbins, and Richard N. Taylor. Using object-oriented typing to support architectural design in the C2 style. In *Proceedings of the* 4^{th} *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 24–32. ACM Press, 1996.

[MPW92a] Robin Milner, Joachin Parrow, and David Walker. A calculus of mobile processes, part I. *Journal of Information and Computation*, 100:1–40, 1992.

[MPW92b] Robin Milner, Joachin Parrow, and David Walker. A calculus of mobile processes, part II. *Journal of Information and Computation*, 100:41–77, 1992.

[MT00] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering*, 26(1):70–93, 2000.

[MTW96] N. Medvidovic, R. N. Taylor, and E. J. Whitehead, Jr. Formal modeling of software architectures at multiple levels of abstraction. In *Proceedings of the California Sofivare Symposium (CSS'96)*, April 1996.

[OGT$^+$99] P. Oreizy, M.M. Gorlick, R.N. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum, and A.L. Wolf. An architecture-based approach to self-adaptive software. *Intelligent Systems, IEEE*, 14(3):54–62, May/Jun. 1999.

[OMT98] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the International Conference on Software Engineering*, pages 177–186, Apr. 1998.

[Ore96] Peyman Oreizy. Issues in the runtime modification of software architectures. Technical Report UCI-ICS-96-35, University of California, Aug. 1996.

[Ore98] Peyman Oreizy. Issues in modeling and analyzing dynamic software architectures. In *Proceedings of the International Workshop on the Role of Software Architecture in Testing and Analysis*, June-July 1998.

[OT98] Peyman Oreizy and Richard N. Taylor. On the role of software architectures in runtime system reconfiguration. In *Proceedings of the International Conference on Configurable Distributed Systems*, page 61. IEEE Computer Society, 1998.

[Rap96] The Rapide 1.0 full syntax reference manual. Web page: `http://pavg.stanford.edu/rapide/language.html`, Sept. 1996.

[RJC00] Nelson S. Rosa, George R. R. Justo, and P. R. F. Cunha. Incorporating non-functional requirements into software architectures. In *Proceedings of the IPDPS Workshop on Formal Methods for Parallel Programming (FMPPTA 2000)*, May 2000.

[Saa99] Mark Saaltink. The Z/EVES 2.0 user's guide. Technical Report TR-99-5493-06a, ORA Canada, Oct. 1999.

[SG95] Mary Shaw and David Garlan. Formulations and formalisms in software architecture. In *Computer Science Today*, pages 307–323. 1995.

[SG02] Bradley Schmerl and David Garlan. Exploiting architectural design knowledge to support self-repairing systems. In *Proceedings of the* 14*th* *International Conference on Software Engineering and Knowledge Engineering*, pages 241–248, 2002.

[SNH95] Dilip Soni, Robert L. Nord, and Christine Hofmeister. Software architecture in industrial applications. In *Proceedings of the* 17*th* *International Conference on Software Engineering*, pages 196–207, 1995.

[Szy03] Clemens Szyperski. Component technology: what, where, and how? In *Proceedings of the* 25*th* *International Conference on Software Engineering*, pages 684–693, 2003.

[TGM98] Gabriele Taentzer, Michael Goedicke, and Torsten Meyer. Dynamic change management by distributed graph transformation: Towards configurable distributed systems. In *Proceedings of the* 6*th* *International Workshop on Theory and Application of Graph Transformation (TAGT'98)*. Springer, Oct. 1998.

[TGM99] Gabriele Taentzer, Michael Goedicke, and Torsten Meyer. Dynamic accommodation of change: Automated architecture configuration of distributed systems. In *Proceedings of the* 14*th* *IEEE International Conference on Automated Software Engineering*, Oct. 1999.

[vG87] R.J. van Glabbeek. Bounded nondeterminism and the approximation induction principle in process algebra. In *Proceedings 4th Annual Symposium on Theoretical Aspects of Computer Science (STACS 87)*, Feb. 1987.

[vL00] Axel van Lamsweerde. Formal specification: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, pages 147–159. ACM Press, 2000.

[VM94] Björn Victor and Faron Moller. The Mobility Workbench — a tool for the π-calculus. In *Computer Aided Verification (CAV'94)*, volume 818 of *Lecture Notes in Computer Science*, pages 428–440. Springer-Verlag, 1994.

[VPL99] James Vera, Louis Perrochon, and David C. Luckham. Event-based execution architectures for dynamic software systems. In *Proceedings of the* 1*st* *Working IFIP Conference on Software Architecture*, pages 22–24, San Antonio, Texas, 1999. IEEE.

[Wer98a] Michel Wermelinger. A simple description language for dynamic architectures. In *Proceedings of the 3rd International Software Architecture Workshop*, pages 159–162, 1998.

[Wer98b] Michel Wermelinger. Towards a chemical model for software architecture reconfiguration. *IEE Proceedings - Software*, 145(5):130–136, October 1998.

[Wer99] Michel Wermelinger. *Specification of Software Architecture Reconfiguration*. PhD thesis, Universidade Nova de Lisboa, September 1999.

[WF98] Michel Wermelinger and José Luiz Fiadeiro. Connectors for mobile programs. *IEEE Transactions on Software Engineering*, 24(5):331–341, May 1998.

[WF99] Michel Wermelinger and José Luiz Fiadeiro. Algebraic software architecture reconfiguration. In *Proceedings of the* 7^{th} *European Software Engineering Conference and* 7^{th} *Symposium on Foundations of Software Engineering (ESEC/FSE'99)*, pages 393–409, 1999.

[WF02] Michel Wermelinger and José Luiz Fiadeiro. A graph transformation approach to software architecture reconfiguration. *Science of Computer Programming*, 44(2):133–155, August 2002.

[WLF01] Michel Wermelinger, Antnia Lopes, and José Luiz Fiadeiro. A graph based architectural (re)configuration language. In *Proceedings of the Joint* 8^{th} *European Software Engineering Conference and* 9^{th} *Symposium on the Foundations of Software Engineering*, pages 21–32. ACM Press, 2001.

A A Proposed Evolution Taxonomy

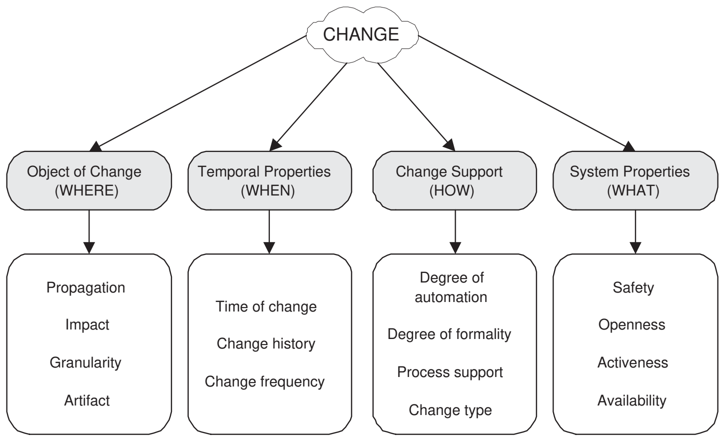


Figure 22: Dimensions of software change

The above diagram outlines the proposed evolution taxonomy in [BMZ$^+$04]. The approach is based on the mechanism of changes and organizes properties of change into four dimensions: the where, when, how and what of change. The properties for each dimension are listed in the boxes at the bottom of the diagram.

Buckley et al. approach developing a taxonomy for change from a mechanisms perspective [MBZR03, BMZ$^+$04]. Their proposed taxonomy does not concentrate on *why* a change occurs but instead focuses on the *how*, *what*, *when*, and *where* of an evolutionary change (see Figure 22).

Temporal properties such as time of change, change history, and change frequency define *when* a change occurs. Object of change properties such as artifacts, granularity, impact, and change propagation define *where* a change occurs. System properties define *what* kinds of changes can occur. A system is defined by availability, activeness, openness, and safety properties. Finally, change support properties such as degree of automation, degree of formalism, process support, and change type define *how* a change occurs.

This proposed taxonomy was designed for evolution in a general sense and thus is able to identify differences between a variety of different evolution systems and tools. For example, in [BMZ$^+$04] it is used to compare a version control system, a browser that supports refactoring transformations, and self-managing servers. However, when the scope of comparison is restricted to run-time evolution at the architectural level, the differences become more subtle and harder to distinguish. In fact, many of the key properties in the proposed taxonomy such as the time of change, the artifacts, and the granularity become fixed (see Table 6). Overall, many properties related to the *when* and *where* of architectural changes differ very little from one dynamic architecture to another. Furthermore, it remains difficult to answer the three fundamental questions presented in Section 1 using the remaining properties that do vary. Therefore, in order to evaluate dynamic architectures a refinement of this taxonomy would be required.

	General Evolutionary Change	Dynamic Architectural Evolutionary Change
Temporal Properties *(when)*		
time of change	compile-time, load-time, run-time	run-time
change history	none, sequential, parallel	changes may be executed in parallel to minimize disruption time
change frequency	continuously, periodically, arbitrarily	self-adaptive systems tend to change continually, while ad-hoc changes occur arbitrarily
Object of Change *(where)*		
artifact	source code, file, executable code, etc.	executable code (or components, connectors)
granularity	several methods, several classes, file, system, etc.	systems, components, connectors
impact	local source code changes, global changes to any artifacts, etc.	possibly global changes to other component and connectors
System Properties *(what)*		
availability	permanently available or not	often permanently available
activeness	reactive, proactive	some reactive systems, all proactive
openness	built in support for extensions	run-time infrastructure required for dynamic change means most systems at least partially open - not closed
Change Support *(how)*		
automation	automated, partially automated, manual	automated, partially automated
formality	ad-hoc or formal mechanism	often formal
process support	activities in change process supported by automated tools	configuration
change type	structural, semantics-modifying, semantics-preserving	the same possibilities, but within structural changes one must distinguish which operators (addition, etc.) and operands (architectural elements) are allowed

Table 6: Relationship between the proposed evolution taxonomy described in [BMZ$^+$04] when applied to evolutionary change in general and evolutionary change in dynamic software architectures.