Exploiting Architectural Prescriptions for Self-Managing, Self-Adaptive Systems: A Position Paper

Matthew J. Hawthorne
Department of Electrical and Computer Engineering
The University of Texas at Austin
hawthorn@ece.utexas.edu

Dewayne E. Perry
Department of Electrical and Computer Engineering
The University of Texas at Austin
perry@elgar.ece.utexas.edu

ABSTRACT

We propose a high-level approach to software architecture that bridges the gap between system requirements (in the problem space) and the architectural design (in the solution space). We use abstract *constraint-* and *intent*-based architectural prescriptions to enable architectural reflection, reification, and distributed configuration discovery as the basis for designing adaptive, self-configuring software systems. We discuss some key architectural properties and patterns that facilitate the design and implementation of self-configuring systems, and use these as the basis for an example prototype architecture for self-evolving systems called *Distributed Configuration Routing (DCR)*. Finally, we propose the development of *architectural prescription languages (APLs)* and enhanced system design environments to provide better support for intent-based architectures.

1. INTRODUCTION

Adaptive, self-evolving systems are systems that are able to evolve dynamically in response to changes in their environment. While systems may adapt at any level, for maximum flexibility, the software architecture [17] itself should be self-evolving. Architectures defined at high levels of abstraction using *problem domain* terminology expressed in terms of system *goals* and *constraints* tend to be relatively flexible and easy to evolve because they derive implementation configurations directly from system requirements, enhancing the ability of the system to reason about its level of conformance to the requirements. Most existing approaches to software architecture focus on implementation design. This often results in architectures that offer little support for reasoning about conformance to system requirements, and subsequent system evolution usually only exacerbates this disconnection between requirements and architecture.

To avoid these limitations, and expand the theoretical and practical foundation for self-evolving system architectures, we present an architectural approach that builds a more direct connection between problem-domain approaches like goal-oriented *requirements engineering (RE)* [12] and the implementation architecture. We start with the goal-oriented RE entities used to specify system requirements: *goals* (functionality requirements) and *constraints* (prescriptions regarding functional and non-functional properties of the system). To these, we add *activities* to model application-domain tasks and processes, and *roles* to specify any constraints that govern the use of a given solution-domain component in a given problem-domain context. A robust *intent* model enables the system to reify implementation architectures from incomplete abstract problem-domain specifications by providing a semantic framework for the system to reason about the behavioral aspects of all components. Defining the architecture at this level of abstraction allows adaptive systems to select the most suitable component to satisfy a given system role based on real-time system and environmental conditions. We discuss some basic requirements for adaptive systems, along with a few key architecture styles, patterns and techniques that enhance those properties. We present an example prototype architecture for self-configuring systems called *Distributed Configuration Routing (DCR).* And lastly, we propose extensions to existing system design tools, common frameworks to support *role* and *inten*t abstractions, and development of *architectural prescription languages (APLs).*

2. A PRESCRIPTIVE APPROACH

Perhaps the most useful property adaptive system architectures can have is the ability to ensure conformance to system requirements specifications during self-configuration. Prescriptive architectures [1,2] "naturally" exhibit this property because they directly utilize the requirements to derive implementation architectures. Practical prescriptive architectural approaches are an important step toward conformant self-configuring systems.

This section describes a conceptual framework that builds on recent goal-oriented requirements engineering research [12], and recent research on transforming requirements to architecture specifications [1,2,11,13,21]. We use semantically rich problem and solution domain abstraction models to effectively bridge the gap between requirements engineering and the implementation domain. Our approach enforces problem domain goals and constraints, while decreasing the coupling between goal-driven architectures and implementation object types, enabling multiple subsequent prescriptive architectures to be derived from a given requirements specification. This framework provides the foundation for the self-configuring prototype architecture described later in this paper.

Our prescriptive architecture approach starts with goal-oriented requirements engineering, a modeling and refinement process in which we model system requirements as a set of *goals*, and refine these goals into *functional goals* and *constraints*. *Functional goals* describe the functionality of the system, while *constraints* prescribe quantitative or qualitative properties of specific functional goals or of the system as a whole. Examples of constraints include performance and quality requirements.

Once the functional goals and constraints are specified, we model the system as the set of *activities* the system will perform to fulfill the *functional* goals. *Activities* model what the system will do, using problem domain terms. Examples of activities are "Financial Data Analysis", "Pumping Station Control", or any other functionality that makes sense for the application domain. High-level activities that model coarse-grained units of functionality are analogous to application subsystems. We decompose high-level activities into lower-level activities until they are "atomic" enough to be fulfilled by one, or only a few, implementation object *roles*. We assign a given constraint to any activity to which the constraint applies, decomposing constraints into lower-level (more specific) constraints as needed to apply them to lower-level activities.

Roles are abstractions of the role an implementation object will play in a certain context in the application. Roles enable the system to model constraints and other information that affects the way an application *uses* a particular object type in a certain application context. This allows different roles to use a given implementation object type without requiring implementation objects to carry application-specific context. Context-specific behavior is specified by the *role* and the *constraints* associated with that role. We want to avoid tight coupling between application *activities* and implementation object types to enable system architectures to be defined at a high level of abstraction. This allows a given application *role* to be fulfilled by any number of different implementation object types, as long as they implement equivalent behavior (have the same functional *intent*).

To enable dependable functionality mappings from requirements to implementation objects, we need a *solution domain* model that unambiguously specifies the *intended use* and *functionality* of implementation objects. We use an *intent* framework to classify and model implementation object functionality. *Intents* capture the essence of what component types are intended to accomplish (their *purpose*), and also specify their behavioral intent (their *function*). Used with *activities* and *roles*, *intents* enable us to reason about the ability of implementation objects to fulfill system goals by formally expressing the kind of information about how to *use* a component that many architectural description languages (ADLs) may express informally, if at all. Unlike typical ADL object interface specifications, *intents* do not directly prescribe the object types required or produced by a given object, nor do they specify object organization or object-object relations. Instead, *intents* are *behavioral* abstractions that specify the *purpose* and *functionality* of objects. In other words, *intents* specify object interfaces in the *purpose and functionality domains*, making them orthogonal to traditional interface definitions, which are concerned with the *object relationship domain*. Both types of models are needed to fully specify and reason about an implementation object's use in a system design.

Intents specify an object type's behavior using a *state change* model to describe functions the object can perform, specified using sets of {*from state*, *to state*}). State changes can range from low-level data transformations to system-level tasks or states. *Intents* fully specify object behavior, so any two objects with the same *intent* can be used interchangeably to accomplish the same implementation domain purpose [16].

Since the *intent* model specifies verifiable behavior, the system has the information necessary to instantiate component and runtime monitor instances. Configuration involves selecting from among available implementations with a given *intent*, using the application context information in the *role* for which an implementation is needed both to aid the selection process and to apply any role-specific *constraints* to the selected implementation object. The role's constraint context determines the number and nature of any runtime monitor components that must be reified to implement self-reflection required to test runtime constraints.

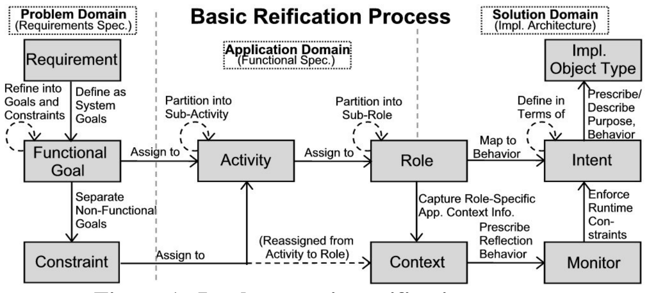

Basic Reification Process
Problem Domain (Requirements Spec.)
Application Domain (Functional Spec.)
Solution Domain (Impl. Architecture)
Requirement
Refine into Goals and Constraints
Define as System Goals
Functional Goal
Assign to
Partition into Sub-Activity
Activity
Assign to
Partition into Sub-Role
Role
Map to Behavior
Define in Terms of
Intent
Prescribe/ Describe Purpose, Behavior
Impl. Object Type
Separate Non-Functional Goals
Constraint
Assign to
(Reassigned from Activity to Role)
Capture Role-Specific App. Context Info.
Context
Prescribe Reflection Behavior
Monitor
Enforce Runtime Constraints

Figure 1. Implementation reification process

Figure 1 depicts the basic conceptual reification process from the original requirements specification through selection of implementation and runtime monitoring components to reify a functioning architectural configuration.

3. ADAPTIVE SYSTEM REQUIREMENTS

In this section we briefly discuss some of the basic requirements for self-evolving systems and suggest design features and techniques that should be useful for designing such systems.

Self-managing systems must support *automatic reconfiguration*, the ability to reconfigure themselves in response to external events and administrative directives. Techniques that support automatic reconfiguration include *reflection* (self-awareness), and *loosely coupled component* designs such as *component-connector* and *service provider* architectures.

Any system that reconfigures itself should include a *learning and planning* facility to enable it reason about empirical results and evolve its reconfiguration practices based on experience, or much of its reconfiguration activity is likely to be counterproductive. Performance, error and failure rates, and conformance statistics are useful data, among many others. To establish useful cause-and-effect relations to guide reconfigurations, observed effects must be correlated with the many aspects of the system that may influence them, including component types, versions and configurations, as well as system information including specifications, system type (platform, OS, etc.), and others. Ideally, a self-configuring system should also be able to correlate runtime results with architectural patterns [3,5]. This information can then be used to guide component selection, arrangement and configuration. Algorithms and patterns that support this type of learning and planning include *correlation coefficient* and *Bayesian inference* algorithms, and *neural nets*.

Since requirements specifications are usually incomplete, it is desirable for an adaptive architecture to support building system configurations from *incomplete specifications*. Techniques that support reification from incomplete specifications include *default implementations* and *intent-based implementation taxonomies*.

While not necessarily an essential property of self-evolving systems, *scalability* is certainly a requirement for any practical implementation. Techniques that support scalability include *distributed architectures, decentralized configuration*, and *distributed service discovery*. And although it is possible to design an adaptive system that does not support heterogeneous platforms and software frameworks, support for *heterogeneity* enables a system to utilize a richer mix of technologies and off-the-shelf components, and is an important aspect of design diversity for dependable systems [9]. Some of the techniques that support heterogeneity are *platform-independent frameworks and protocols*, and *adapters*.

And lastly, *security* is an important concern for all serious systems, especially self-configuring systems. Adaptive systems must maintain any prescribed security properties through reconfigurations and other adaptive changes. Self-configuring systems should perform authentication and integrity checking on components when they register with the system, and distributed systems are likely to require runtime integrity checking of events and data. Depending on the application, systems may also need to perform authorization checks before allowing certain operations. Some applications will also require confidentiality and non-repudiation. Techniques that simplify support for security include *component-connector* designs (security functionality can be included in smart connectors), and *opt-in component registration* (provides a convenient common point for performing component authentication and integrity checking, as well as basic authorization, if applicable).

4. AN EXAMPLE SELF-MANAGING ARCHITECTURE

This section describes *Distributed Configuration Routing (DCR)*, a prototype architecture for self-configuring systems based on the architectural approach and requirements previously discussed. To enhance scalability and gain the flexibility of loosely coupled system components, DCR uses a *state-change-based distributed service provider* configuration discovery architecture with an *intelligent connector network* topology. Connectors handle all component interactions, and also implement an extensible component monitor/data adapter framework. The basic topology and instance cardinality are shown in *Figure 2*.

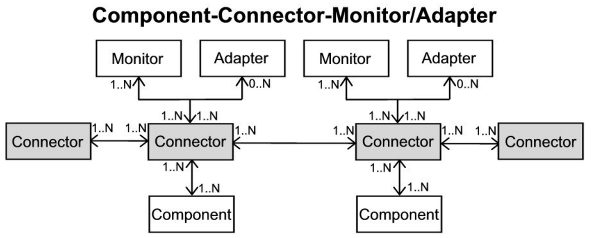


Figure 2. Basic connector model topology and cardinality

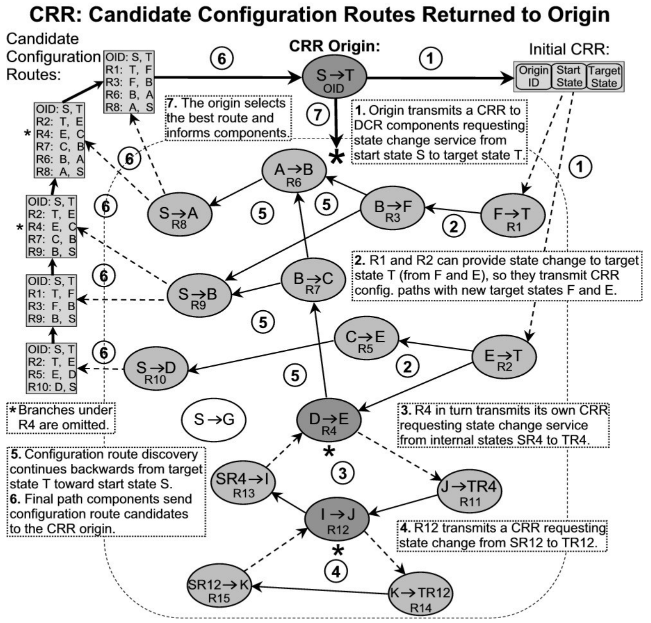
4.1 Configuration Route Discovery

To enable self-configuring systems to respond to unforeseen conditions, the DCR configuration architecture implements distributed *state-change-based configuration route discovery* at arbitrary levels of granularity, where all services are defined as *state changes*. Unlike typical service provider networks, where available services are limited to the set of functionality explicitly programmed into the current set of service providers, DCR components can request state changes for which *no* provider exists. Utilizing the information in the *role* and *intent* models to reason about constraints and behavior, the configuration route discovery architecture enables the network of service provider components to compose conformant system and subsystem configurations from arbitrary numbers of intermediate partial solutions, without requiring the requesting component to know anything about the set of services currently available. Adapted in part from wireless network routing protocols, DCR configuration route discovery utilizes agent-components that independently respond to service requests and incrementally build configurations by agreeing to participate in candidate configuration routes for which they can provide a partial or complete solution. The algorithm uses only information locally available or contained in the service request.

DCR configuration route discovery begins when a component uses its associated *role* information to generate a *Configuration Route Request (CRR)* for state change service it requires (defined as a {*start state*, *target state*} pair), and multicasts the CRR to the registered service provider components. Starting from the *target state* and working backwards toward the *start state*, candidate configuration routes grow incrementally as components add entries for themselves to CRRs whose *current* target state (the *original* target state, or the *from* state of the last component added to the CRR) is a state *to* which the component offers service.

A component that can provide state changes *to* the requested *target state* (i.e., a *to state* it provides matches the CRR *target state*), from any other *from state* not yet used in the CRR path, appends its component ID and state change information to the CRR, making its own *from state* the new CRR *target state*, and retransmits. To prevent loops, components are usually prohibited from responding to requests that already include a response with their *intent*, except when the requester indicates *a priori* that multiple instances of a given state change are needed to complete a given path.

When a component responds to a CRR, it reserves whatever processing capability is required to fulfill the functional and performance constraints of the request. The responder maintains a soft state reservation that includes request and originator IDs, and any applicable information about the level of service reserved (QoS or performance constraints, etc.). This reservation times out after a certain interval if the component does not hear a *Configuration Route Notification (CRN)* message from the originator. While a reservation is active, it may affect the component's ability to answer other CRRs because of reduced residual capacity, but unless the request is for multiple *parallel* configuration routes, it is safe for a component to add itself to multiple *non-duplicate* configuration routes *for the same original CRR*, since at most one route will be selected.



CRR: Candidate Configuration Routes Returned to Origin
Candidate Configuration Routes:
OID: S, T
R1: T, F
R3: F, B
R6: B, A
R8: A, S
OID: S, T
R2: T, E
* R4: E, C
R7: C, B
R6: B, A
R8: A, S
OID: S, T
R2: T, E
* R4: E, C
R7: C, B
R9: B, S
OID: S, T
R1: T, F
R3: F, B
R9: B, S
OID: S, T
R2: T, E
R5: E, D
R10: D, S
CRR Origin:
S→T
OID
Initial CRR:
Origin ID
Start State
Target State
7. The origin selects the best route and informs components.
1. Origin transmits a CRR to DCR components requesting state change service from start state S to target state T.
A→B
R6
S→A
R8
B→F
R3
F→T
R1
2. R1 and R2 can provide state change to target state T (from F and E), so they transmit CRR config. paths with new target states F and E.
S→B
R9
B→C
R7
C→E
R5
E→T
R2
S→D
R10
* Branches under R4 are omitted.
S→G
D→E
R4
3. R4 in turn transmits its own CRR requesting state change service from internal states SR4 to TR4.
5. Configuration route discovery continues backwards from target state T toward start state S.
6. Final path components send configuration route candidates to the CRR origin.
SR4→I
R13
J→TR4
R11
I→J
R12
4. R12 transmits a CRR requesting state change from SR12 to TR12.
SR12→K
R15
K→TR12
R14

Figure 3. CRR showing multiple configuration candidates; note branching from cascading CRRs at R4 and R12

A configuration route is finished when a component whose *from state* matches the *original CRR start state* responds. The terminal responder adds itself to the route, and unicasts the new candidate configuration route to the CRR originator. *Figure 3* depicts CRR route building after an origin component transmits a CRR, showing how components add themselves to route requests to build configurations from the target state to the start state. *Figure 3* also shows how configurations branch when components in a configuration make subsequent route requests to fulfil their functionality requirements. Finally, *Figure* 3 shows the terminal components in discovered routes returning candidate configuration routes to the CRR origin.

4.2 Route Reservation and Confirmation

The CRR originator selects the best configuration route, either by using a shortest-path or least-cost metric, or by reasoning about the configurations using any number of constraint and optimization metrics. *Figure 4* illustrates the use of a cost metric to select an optimal configuration route. *Figure 5* shows the hierarchical dependency view of the selected configuration.

Having selected the optimal configuration, the originator then multicasts a *CRN* message to all the components in the configuration route. Components participating in the new configuration complete the handshake by sending a Configuration Route Confirmation (CRC) message back to the originator, and the component connectors form the prescribed structure. After the selected route is confirmed, the originating component also sends a *Configuration Route Cancellation (CRL)* message to all components that are part of unused configuration routes.

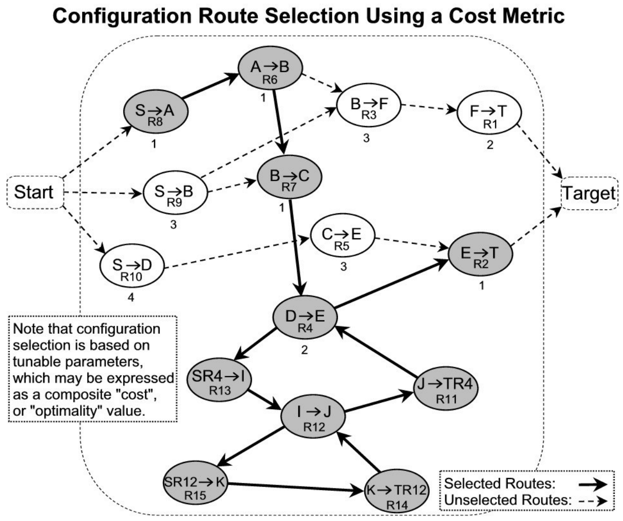

Configuration Route Selection Using a Cost Metric
Start
Target
S→A R8 1
A→B R6 1
B→F R3 3
F→T R1 2
S→B R9 3
B→C R7 1
C→E R5 3
E→T R2 1
S→D R10 4
D→E R4 2
SR4→I R13
J→TR4 R11
I→J R12
SR12→K R15
K→TR12 R14
Note that configuration selection is based on tunable parameters, which may be expressed as a composite "cost", or "optimality" value.
Selected Routes:
Unselected Routes:

Figure 4. Origin selects configuration based on metrics

4.3 Monitors, Adapters and Reconfiguration

A default monitor monitors each component for aliveness and basic performance. The connector framework includes a flexible plug-in mechanism allowing additional monitors and adapters to be added as needed to ensure system runtime constraints (performance, etc.), or to resolve interface data type incompatibilities. These plug-in components are regular DCR service provider components. Connectors use the CRR mechanism to request monitoring or adaptation services. Monitors that detect system constraint violations or other failures can perform local repairs using the CRR mechanism to find a more suitable replacement component, or if that fails, send a *System Reconfiguration Request (SRR)* event to a higher-level component that can perform reconfiguration. Eventually, the SRR may reach the top system-level component, which can reconfigure the entire system if necessary.

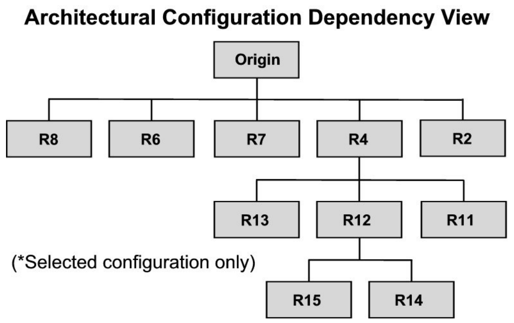

Architectural Configuration Dependency View
Origin
R8
R6
R7
R4
R2
R13
R12
R11
(*Selected configuration only)
R15
R14

Figure 5. Dependency view of selected configuration

4.4 Distributed Conformance Reasoning

The CRR framework includes a rich set of functionality to push constraint checking out into the distributed component network by specifying configuration constraints as part of CRRs generated for a given role. Distributed nodes can reason about the conformance of unfinished configurations at route creation time, and halt processing of non-conforming routes. A flexible multi-dimensional cost mechanism enables routing components to discard routes that exceed specified cost metrics. Cost may include actual costs, or more abstract "costs" such as route performance, system and network load factors, etc., enabling weighted apples-and-oranges comparisons between widely divergent factors. A CRR may also include architectural pattern prescriptions and other non-functional constraints.

5. DISCUSSION

The architectural prescription approach we propose extends recent research in requirements engineering [12,13] and architectural prescriptions [1,2,11]. In particular, our prescriptive reification approach is very closely related to [13] and extends their model by overlaying first-class *role* and *intent* abstractions at the requirements-implementation interface to enable us to model the *implementation* architecture in terms of abstractions that make sense in the requirements domain, without needing to know anything more about the actual implementation objects than their *intent*. This also serves to decouple the architecture from implementation object types and instances, allowing us to delay the binding of application *roles* to actual implementation objects until configuration time (e.g., during adaptive DCR reconfiguration). Overall, our approach builds on, and requires, the kind of requirements domain model analysis described in [13].

The DCR architecture enables self-configuring system behavior without the scalability bottleneck of a centralized controller that must reason about the universe of possible configurations, or the need for distributed components to maintain a global system configuration view. The distributed route discovery protocol can discover unplanned "composite" services that may be required to solve unforeseen application domain or system problems. Broken components can even be replaced by confederations of lower-level components that are able to self-organize in new ways to perform the required service. The system implements self-reflection using a federation of loosely coupled components, with an architecture *functionally* derived from the problem domain, enabling high-level reasoning about conformance to system goals and constraints. And since the DCR system organization is based on high-level abstractions of component *roles* and *intents*, it will allow for diverse ecosystems of components to be built over time, which should enhance system dependability [9].

The self-organizing system techniques discussed here can also be used with architectural prescriptions to instantiate new members of a product family architecture [18], for example, by constraining component types and their associations, and prescribing other architectural style constraints. In this case, the essential properties of the product family architecture are specified as requirements domain goals and constraints.

While the DCR architecture shows promise as a framework for developing self-managing systems, there are still several outstanding technical issues related to aspects of the system design. We have identified learning and planning as a desirable capability for self-evolving systems. But regardless of which learning algorithm we use, *causes* are complex, and so are *effects*. Determining true causes, and eliminating useless "false positive" correlations between multiple effects of the same cause will be challenges. Another issue is how much support the system should include for component configuration. Much of our work to date has focused on architectures for system and subsystem-level reconfiguration. While our current design supports architectural pattern or style constraints [3,5,17], we are also researching more formal ways to specify architectural constraints, as well as partially self-configuring systems. At the same time, we're not completely convinced that we should constrain the result space at all in every case, since this may negate many of the potential benefits of allowing the network to discover unforeseen configurations. The best (or only) solution may not conform to any given prescribed architectural pattern.

6. RELATED RESEARCH

As previously discussed, our architectural prescription approach builds on the body of requirements engineering research [12,13]. [10,15] model adaptable architectures as component-connection graphs, and represent architectural styles as classes of graphs, enabling graph-rewriting rules to be used for reconfiguration. [8,19] describe path-based adaptive architectural approaches used for adapter-based data stream modification. Their strongly typed data-oriented approaches are less general than our state-change-based approach, which can be used for general application purposes as well as for data transformations.

More recently, [6] discusses externalized model-based adaptation, in which architectural models are used as the basis for problem diagnosis and repair, and architectural styles can be tailored to support desired system qualities, including partial self-configurability. [7] uses architectural constraints as the basis for specification, design, and implementation of self-organizing architectures for distributed systems. However, their approach requires each component to have a global view of the system configuration, so they require reliable and totally ordered broadcast, and cannot tolerate network partitions. Our distributed approach is much less centralized; no component has a global view of the system configuration. [14] proposes a 3-tier architecture that separates computation, coordination and configuration to enable fault treatment using redundant components. [20] discusses extending existing design tools to provide better support for runtime architectural adaptation, while [4] describes a self-management coordination architecture and infrastructure to enable components in self-healing systems to work together, incorporating consistent system access, non-conflicting decision-making, and a consistent system model.

7. CONCLUSIONS

We extend the rigorous problem domain modeling of goal-oriented requirements engineering by developing a reification process that bridges the gap between the problem and solution domains. We use *activities* and *roles* to effectively extend the requirements model, and a robust component *intent* framework to extend the implementation model. This provides an abstract but semantically rich way to use requirements specifications to prescribe implementation architectures without knowing implementation details. This approach lays the groundwork for developing automated systems that will be able to generate implementation architectures directly from system specifications, a key step toward developing conformant self-managing system architectures. We discuss some basic requirements for adaptive systems, and architecture and design patterns that can aid in their design and development, and we provide an example architecture for self-managing systems we are currently developing called *Distributed Configuration Routing (DCR)*.

We propose several developments to make prescriptive architectural approaches more practical. First, extend existing *architectural description languages (ADLs)* and *requirements description languages (RDLs)* to enhance their modeling capabilities in complementary domains. Second, common *role* and *intent* frameworks for software components would give architects a common language to use for communicating and reasoning about families of requirements domain prescriptions and the functional intent of implementation domain objects, similar to the way architecture and design patterns have enhanced communication about implementation models. Third, create an *architectural prescription language (APL)* that combines the best features of both ADLs and RDLs, including tool support for requirements specification, prescriptive architectural design, and system configuration. Such an APL-based environment would enable system specification and design to be accomplished as a single process within a single framework, and enable reasoning about system conformance and design at any stage of completion.

8. REFERENCES

[1] Brandozzi, M. and Perry, D. Architectural Prescriptions for Dependable Systems. *ICSE WADS 2002*, May 2002.

[2] Brandozzi, M. and Perry, D. From Goal-Oriented Requirements to Architectural Prescriptions: The Preskriptor Process. *Intl. Workshop From Software Requirements to Architectures*, May 2003, 107-113.

[3] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1996.

[4] Cheng, S., Huang, A., Garlan, D., Schmerl, B., and Steenkiste, P. An Architecture for Coordinating Multiple Self-Management Systems. *WICSA-4*, 2004.

[5] Garlan, D., Allen, R.J. and Ockerbloom, J. Exploiting Style in Architectural Design. *Proc. SIGSOFT '94 FSE-2*, 175-188.

[6] Garlan, D., Schmerl, B. Model-Based Adaptation for Self-Healing Systems. *WOSS'02*, 2002, 27-32.

[7] Georgiadis I., Magee J. and Kramer J. Self-Organising Software Architectures for Distributed Systems. *WOSS'02*, 2002.

[8] Gribble, S., Welsh, M., von Behren, R., Brewer, E., Culler, D., Borisov, N., Czerwinski, S., Gummadi, R., Hill, J., Joseph, A., Katz, R., Mao, Z., Ross, S., and Zhao, B. The Ninja Architecture for Robust Internet-Scale Systems and Services. *IEEE Comp. Networks, Special Issue on Pervasive Computing*, Vol. 35, No. 4, Mar. 2001.

[9] Hawthorne, M. and Perry, D. Applying Design Diversity to Aspects of System Architectures and Deployment Configurations to Enhance System Dependability. *WADS'04*, June 30, 2004.

[10] Hirsch, D, Inverardi, P. and Montanari, U. Graph Grammars and Constraint Solving for Software Architecture Styles. *3rd Intl. Workshop on Software Arch.*, 1998, 69-72.

[11] Jani, D. Deriving Architecture Specifications from Goal Oriented Requirements Specifications. *Master's Thesis, Dept. Electrical and Comp. Eng., The Univ. of Texas at Austin*, May 2003. Supr.: Dewayne E. Perry.

[12] van Lamsweerde, A. Requirements Engineering in the Year 00: A Research Perspective. *ICSE 2000*, June 2000, 5-19.

[13] van Lamsweerde, A. From System Goals to Software Architecture. *SFM 2003*, 25-43.

[14] de Lemos, R., Fiadeiro, J. An Architectural Support for Self-Adaptive Software for Treating Faults. *WOSS'02*, 2002.

[15] Metayer, D. Software Architecture Styles as Graph Grammars. *Proc. SIGSOFT'96 FSE-4*, Nov. 1996, 15-23.

[16] Perry, D. The Inscape Environment. *ICSE 1989*, May 1989.

[17] Perry, D. Wolf, A. Foundations for the Study of Software Architecture. *ACM SEN*, Vol. 17 No. 4, 1992, 40–52.

[18] Perry, D. Generic Architecture Descriptions for Product Lines. *ARES II: S/W Arch. Prod. Families 1998*, 1998.

[19] Reiher, P., Guy, R., Yarvis, M., and Rudenko, A. Automated Planning for Open Architectures. *Proc. OPENARCH 2000 – Short Paper Session*, Mar. 2000, 17-20.

[20] Schmerl, B. and Garlan D. Exploiting Architectural Design Knowledge to Support Self-repairing Systems. *14th Intl. Conf. Software Eng. and Knowledge Eng.*, 2002.

[21] Vanderveken, D. Deriving Architectural Descriptions from Goal-Oriented Requirements. *Master's Thesis, Dept. d'Ingenierie Informatique, Univ. Catholique dl Louvain*, June 2004. Suprs.: Axel van Lamsweerde and Dewayne E. Perry.