An Approach to Manage Reconfiguration in Fault–Tolerant Distributed Systems

Stefano Porcarelli[1], Marco Castaldi[2], Felicita Di Giandomenico[1], Andrea Bondavalli[3], Paola Inverardi[2]

[1]Italian National Research Council, ISTI Dept., via Moruzzi 1, I-56134, Italy
stefano.porcarelli@guest.cnuce.cnr.it, digiandomenico@iei.pi.cnr.it
[2]University of L'Aquila, Dip. Informatica, via Vetoio 1, I-67100, Italy
{castaldi, inverard}@di.univaq.it
[3]University of Florence, Dip. Sistemi e Informatica, via Lombroso 67/A, I-50134, Italy
a.bondavalli@dsi.unifi.it

Abstract

This paper deals with dynamic resource management for real–time dependability–critical distributed systems. Requirements for such kind of systems span many domains such as time, survivability, and scalability and point out formidable challenges in terms of their fulfillment. An architecture is proposed, based on the agent distributed infrastructure Lira, and enriched with statistical models for decision-making capabilities. The aim of the proposed architecture is to provide adaptive system reconfiguration, resorting to a hierarchy of resource managers to cope with fault tolerance and scalability issues.

1 Introduction

Dependability has become a crucial requirement of current computer and information systems, and it is foreseeable that its importance will increase in the future at a fast pace. We are witnessing the construction of complex distributed systems, which are the result of the integration of a large number of low–cost, relatively unstable COTS (Commercial Off–The–Shelf) components, as well as previously isolated legacy systems. The resulting systems are being used to provide services, which have become critical in our everyday life. Since COTS and legacy components are not designed to achieve high dependability by themselves, their behavior with respect to faults can be the most disparate. Thus, it is paramount for these kinds of system to be able to survive failures of individual components, as well as attacks and intrusions, although with degraded functionalities. This paper contributes to fault tolerance in such framework, by focusing on fault handling strategies [1], particularly on system's reconfiguration.

The aim of system reconfiguration is to provide control capabilities over unanticipated events in order to maintain the system in a certain desirable state. An effective reconfiguration policy highly depends on an accurate diagnosis of the nature of the unanticipated event, namely if a hard, physical fault is affecting the system, or environmental adverse conditions are causing a soft fault which will naturally disappear in some time. It is out of the scope of this paper to address diagnosis issues; we concentrate on reconfiguration only, which is triggered on the basis of information on the healthy status of system components, assumed accessible to our management architecture.

The rest of the paper is organized as follows. Section 2 describes our architectural approach towards a fault tolerant resource management policy. Section 3 details the overall system management architecture and Section 4 introduces a simple case study to illustrate our approach. Section 5 briefly concludes the paper and points out some future research directions.

2 Our Approach for Adaptive Resource Management

A basic characteristic of our resource management architecture is its ability to adapt to dynamic system conditions. Different degrees of adaptivity may be (theoretically) possible, ranging from picking up the reconfiguration policy from a look-up table where policies have been pre-stored on the basis of off-line analysis, to completely dynamic definition of the best reconfiguration to perform. An intermediate solution is that where a number of strategies are pre-planned, but the choice of which one is the most appropriate given certain run-time system conditions is decided dynamically. The degree of dynamic decision making has to be carefully harmonised with other possible system constraints. In fact, the design of a resource management infrastructure for distributed systems is influenced by several factors, among which the nature of the event that triggers a reconfiguration action, the size of the computational effort for decision making, and timeliness requirements.

Adaptive resource management is very demanding nowadays; the approach we propose is mainly characterized by two aspects. Firstly, we introduce a model-based activity, which provides on-line quantitative evaluation of the impact of different reconfiguration strategies and thus helps in the selection of the most appropriate one. Secondly, the decision making process is decomposed in a hierarchical fashion, each level differing in the visibility of, and the ability to act on, the portion of the system under its control. Resorting to a hierarchical approach brings benefits under several aspects, among which: i) favoring fault tolerance by distribution of control; ii) avoiding heavy computation and coordination activities whenever faults can be managed at local level; iii) facilitating the construction and on-line solution of analytical models; iv) favoring scalability.

In our framework, the Light–weight Infrastructure for Reconfiguring Applications (Lira) [2] is used to perform remote control and dynamic reconfigurations over single components or applications. Lira does not formally describe the procedure of decision making, it only assumes that a proper subsystem is in place, in charge of deciding when and in which way the system needs to be reconfigured. In this paper, we are enriching the Lira framework with a hierarchical Decision Maker (DM) in charge of on-line selection of the reconfiguration policy to apply. The DM exploits a model-based support to guide its decision. Lira monitors system and environment conditions passing the state of components and applications to the DM as input for taking decisions.

Upon receiving inputs requiring a reconfiguration, the DM activates the model-based evaluator to devise the most appropriate configuration and behavior to face the current situation [4]. For example, it permits to evaluate the dependability of a new architecture of the system achieved re-arranging resources due to faults, or to carry out cost-benefit tradeoff choices. The output provided by the decision maker is a new system configuration; such output is then managed by Lira, to put into action the selected reconfiguration.

3 The Lira Management Infrastructure and the Decision Maker

Lira is inspired to the Network Management [9] in terms of reconfiguration model and basic architecture. The reconfiguration model of the Network Management is quite simple: a network device, such as a router or a switch, exports some reconfiguration variables and functions through an Agent, which is implemented by the device's producer. These variables and functions exported by the agent are defined in the MIB (Management Information Base) and can be modified using the *set* and *get* messages of SNMP (Simple Network Management Protocol) [9]. For software components, a reconfiguration is any allowed change in the component behavior, while an application reconfiguration is any change in the application's topology in terms of number and location of components [6][5][11].

In the next sections we will first provide an overview of the Lira infrastructure, and then integrate in it the Decision Maker.

3.1 The Lira Infrastructure

The Lira architecture specifies three elements: (i) the **Agent**, which acts on the managed components, implements the reconfiguration logic and communicates with other Agents, (ii) the **MIB** which contains the list of variables and functions exported by the agent, and (iii) the **Management Protocol**, which allows the communication among the agents.

A Lira Agent is a program that runs on its own thread of control and communicates with other Agents in an asynchronous way, using the Management Protocol. There are four kinds of agents organized in a hierarchical way [10] (see Figure 1) and specialized in different tasks: (i) **Component Agents** (CompAgents), associated to software components, (ii) **Host Agents**, associated to the host where the components are deployed, (iii) **Application Agents** (AppAgent), which control part of the system (different components and hosts), (iv) and the **Manager Agent** (Manager), on the top of the hierarchy, that controls the whole system. Accordingly to the hierarchical structure, an agent has manager capabilities on the portion of the system under its own control (helped in the decisions by the Decision Maker), while it is a simple actuator with respect to the higher level agents. In this paper we are discussing a logical architecture without addressing the deployment of the agents on the hosts. Of course, it is a delicate issue for fault tolerance, since it is necessary to have separate containment regions for the monitoring units and the monitored ones.

The *Component Agent* (CompAgent) directly controls and manages the component. Lira does not specify how the component is attached to the agent, but it only assumes that the agent is able to act on the component. The logical model of communication between CompAgent and component is through *shared memory*; in fact, the Component shares a part of its state with the Agent and explicitly allows the reconfiguration. To avoid synchronization problems, the component has to provide atomic access to the shared state. The CompAgent manages the component's life-cycle by exporting the functions *start, stop, suspend, resume* and *shutdown*. The function *shutdown* stops the component and kills the agent. For monitoring purpose, the CompAgent exports a predefined read-only variable *STATUS*, which maintains the current state of the component (starting, started, stopping, stopped, suspending, suspended, resuming). Each agent is able to notify the value of a variable to its manager, addressed by the variable *NOTIFYTO* also defined in the MIB.

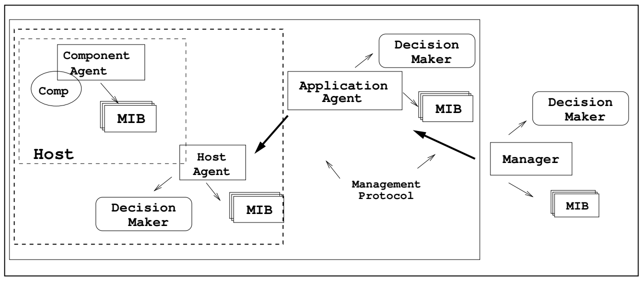

Figure 1. Lira general architecture

The *Host Agent* runs on the host where components and agents must be deployed. It dynamically installs and activates components and agents by exporting the functions *install, uninstall, activate, deactivate*. Moreover, this agent maintains the lists of both installed and activated components on the host: these lists are exported in the variables *ACTIVEAGENTS* and *INSTALLEDAGENTS*. Note that the Host Agent does not manage components, but it monitors and controls host's resources and parameters. All the variables and functions are specified in the MIB.

The *Application Agent* (AppAgent) is a higher level agent which controls a set of components through the associated CompAgents. These agents manage a subsystem as an atomic component, hiding the reconfiguration's complexity and increasing infrastructure scalability.

The *Manager* is the highest level agent which has the global knowledge to control the whole system. AppAgents communicate with the controlled components through the Management Protocol, so they are independent from the specific components and can be programmed by means of an interpreted language: the *Lira Reconfiguration Language*. By means of this language, the agent's developer defines a reconfiguration as a function: the function is invoked internally when the agent is acting in a manager role, or it is exported in the MIB and called by a manager when the agent acts in the actuator role.

The *Management Protocol* is inspired to SNMP, with some modifications and extensions. Each message is either a *request* or a *response*, as shown in the following table:

request	*response*
SET(*var_name, var_value*)	ACK(*msg_text*)
GET(*var_name*)	REPLY(*var_name, var_val*)
CALL(*func_name, par_list*)	RETURN(*ret_value*)

Requests are sent by higher level agents to lower level ones and responses are sent backwards. There is one additional message, which is sent from agents to communicate an alert at upper level (even if in the absence of any request):

NOTIFY(*variable_name, variable_value, agent_name*)

As in the Network Management, the *MIB* represents the agreement among agents that allows them to communicate in a consistent way. The MIB provides the list of variables and functions exported by the agent which can be remotely managed. A function is usually a reconfiguration process that the agent makes available for its manager. Note that also the predefined variables and functions that characterize the different agents (for example, the variable *STATUS* or the function *stop* in the CompAgent) are defined in the MIB. In [3] the MIB is presented in detail.

The Lira infrastructure described here was not created with dependability requirements, so we are investigating how to modify the infrastructure providing new features which guarantee continuous monitoring and distributed management of components and hosts, making Lira dependable itself.

3.2 Decision Maker

The decision maker takes decisions about system's reconfiguration. Decisions can be taken at any level of the agents hierarchy as proposed by Lira (four levels) and, consequently, the power of the reconfiguration is different.

The first, bottom level is that of a Component Agent. At this level, the Decision Maker can only autonomously decide to perform preventive maintenance on the controlled component. At the second level, that of the Host Agent, the DM can decide about installation and de-installation of such components. The third level concerns the Application Agent; at this level, the DM's reconfiguration capabilities span all software and hardware resources under its responsibility. At the highest level there is the Manager agent, which has a "global" vision of the system; therefore the DM at this level can perform an overall reconfiguration. After taking the decision on reconfiguration at a certain level, it is sent to the lower level agents which act as actuators on the controlled portion of the system.

For the sake of simplicity, the status of each monitored system unit may assume three values: *Up* indicating that the component is well working; *Degraded* indicating that the component is working in a degraded manner (e.g., in the case the component is hit by a transient fault which reduces its functionalities); and *Down* indicating that the component is definitely wrongly working (e.g., it is hit by a permanent fault). At each level of the decision making hierarchy, the DM perceives the behavior of each system unit visible at the one-step lower level in terms of up, degraded or down. To make an example, at the manager agent level, the DM has knowledge of the behavior of each application running in the system, each one seen as single system unit; in turn, at application agent level, the DM has knowledge of the behavior of each host involved in that application, again each one seen as a single system unit, and so on.

According to the depicted hierarchical reconfiguration process, when an event triggering a reconfiguration action at a certain level occurs, the DM at that level attempts the reconfiguration, if possible. In case it cannot manage the reconfiguration, it notifies the upper level DM about both detected problem and its healthy status. In turn, the upper level DM receiving such request to trigger a reconfiguration, uses such heathy status information, together with those of the other system units under its control.

As introduced in Section 2, the way to make decisions may be different. If it is possible to assume stochastic independence among failure and repair processes of various components, the new reconfiguration scheme can be simply retrieved from a look-up table where pre–evaluated policies (e.g. by means of combinatorial models, like fault tree) have been stored. If some environmental parameters may change at the moment of the reconfiguration, combinatorial models must be solved each time. In case the failure of a component may affect other related components, space–state models are necessary. These are solved by the DM on the basis of the information collected from the subordinated agents. In this case, each unit component is modeled with a simple Petri net which describes its forecasted behavior given its initial state. It is in charge of the Decision Maker to solve such overall composed model as quickly as possible to take appropriate decisions online identifying the most rewarding reconfiguration action among a pool of pre–defined options.

Therefore, the status of any controlled component (provided by Lira) is used as *input* for the appropriate decision maker, that reasons, decides and gives back as *output* (to Lira) the *optimal reconfiguration action*. Decisions are taken resolving analytical models (not shown for brevity), essentially based on Deterministic and Stochastic Petri Nets (DSPN) [7], and solved by means of the tool DEEM [8].

Obviously, the correctness of the decisions depends both on the accuracy of the models and on its input parameters.

4 A Simple Example: Path Availability of a Communication Network

To better motivate our methodology a possible scenario is presented. A simple, but meaningful scenario, is the case of distributed computing where two peer–to–peer clients on the network are communicating. To prevent service's interruption, we need to provide an adequate level of paths redundancy among the clients involved in the communication. We suppose to have a network topology where six hosts are physically (wired) connected as shown in Figure 2. For management purpose, we consider the network divided in two subnetworks Net_1 and Net_2, which contain the hosts H_1, H_2 and H_3, H_4 respectively. The hosts H_5 and H_6, where the clients are deployed, are not included in the managed network.

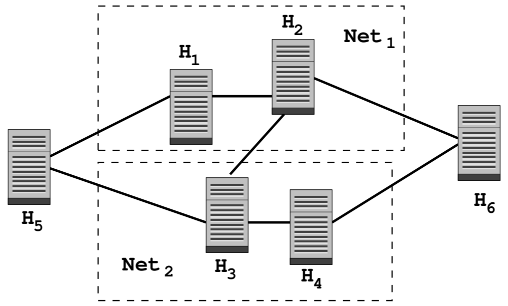

Figure 2. Hosts physical connection

A logical communication network composed by logical nodes connected through logical channels is installed on the managed hosts. The nodes N_1, N_2, N_3, N_4 connected through the channels *a, b, c, d, e, f, g* are deployed on the subnetwork Net_1, as shown in Figure 3. These channels provide different choices for establishing the communication among the clients, as listed in Table 1.

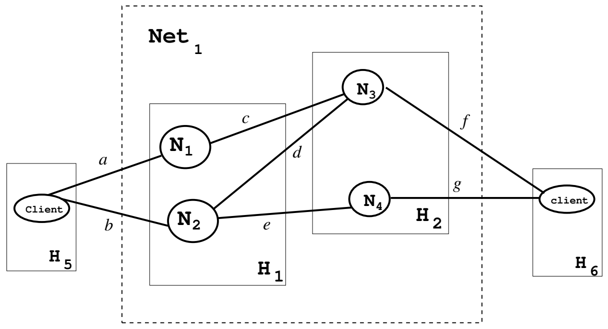

Figure 3. Logical infrastructure topology

Path	Route
1	a–N_1–c–N_3–f
2	a–N_1–c–N_3–d–N_2–e–N_4–g
3	b–N_2–e–N_4–g
4	b–N_2–d–N_3–f

Table 1. Available paths

These path options can be used to improve the overall availability of the logical network by providing redundancy. The goal of the management infrastructure is to keep *at least* two paths available between the clients involved in the

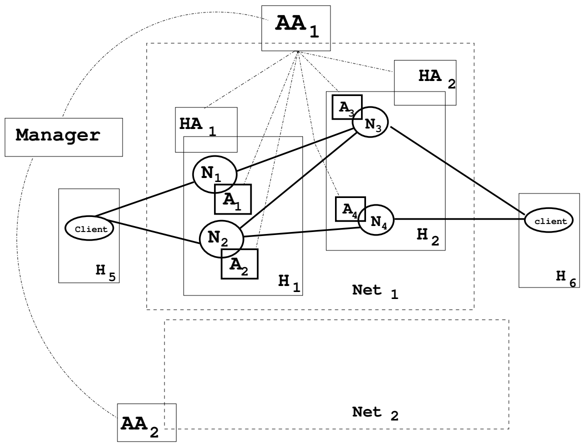

Figure 4. Lira infrastructure for the controlled network

communication. When a hardware or software fault causes paths failure, a reconfiguration is triggered to re-establish path's redundancy. In this example, we consider that manifestation of both a hardware fault (such as a wired connection's interruption or a damage in the physical machine) and a software fault (at operating system, application and logical communication level) have a fail-stop semantics, that is the component stops working.

We are interested in monitoring paths availability, so for a path to be available, all the nodes and links in the corresponding route must be available. Note that failures of a particular link or node may result in unavailability of more than one path. For example, if node N_3 fails, path 1, 2, and 4 become unavailable.

4.1 Lira infrastructure

In this section we describe how the Lira infrastructure is instantiated to manage the network previously described (see Figure 4). Each host H_i is controlled by a Host Agent HA_i, each subnetwork Net_i is controlled by an Application Agent AA_i, while the whole network is controlled by the Manager. The hosts H_5 and H_6 are considered outside the network, so they are not controlled by host agents.

The logical network is also controlled by the Lira agents. The Component Agents A_i control the logical nodes N_i, and they are managed by AA_1. AA_1 may decide to perform a reconfiguration (following the policies specified by the Decision Maker) if it has the necessary information, while it has to ask the general Manager when a global reconfiguration is needed and the local information is not enough. Figure 4 details the Lira management infrastructure.

Each *CompAgent* associated to a logical node exports the enumerated variable *HEALTH_STATE*, which can assume the values *Up, Degraded,* and *Down*. In addition to the defined variables and functions, each CompAgent A_i exports the variable *CONNECTED_NODES*, i.e. the address list of connected nodes, and the function *connectTo(Node nextNode)*, able to connect the local node with the (remote) node specified as parameter. The CompAgent manages also the life cycle of the logical node, by exporting the functions *start, stop, suspend, resume, shutdown*, as defined in Section 3.

Each *Host Agent* exports the functions *install, uninstall, activate, deactivate* and the enumerated variable *HEALTH_STATE*, whose possible values are *Up*, *Degraded* or *Down*. The result of diagnosis over each component is accessible by the agent, and it is notified through a Lira NOTIFY message before a complete crash of the machine. Moreover, the host agent exports the read-only variable *CONNECTED_HOSTS*, which contains the hosts physically connected with the variable's owner. For the host H_2, this variable contains the list H_1, H_3. Note that a host agent can install and activate new logical nodes, creating new routing paths, increasing redundancy and repairing software faults.

The *Application Agent* monitors the subsystem's state and makes it available by exporting the read-only variables *AVAILABLE_PATHS*, *ACTIVE_NODES* and *WORKING_HOSTS*. The first one contains the number of available paths between the clients: for the subsystem controlled by AA_1, the value is 4 (see Table 1). The second one maintains the list of active nodes in the controlled network: in the situation depicted in Figure 4, this variable for AA_1 is $\{N_1, N_2, N_3, N_4\}$. The third one contains the list of still working hosts in controlled network: when HA_i notifies that H_i is down, this variable is modified by the application agent. To change the network topology, the application agent exports the function *connect(Node source, Node dest)*, which is able to connect a source node to a destination node.

The *Manager Agent* controls the subnetworks Net_1 and Net_2 by checking the *WORKING_HOSTS* variable exported by the AppAgents. Thus, it can arrange reconfigurations on the two networks.

4.2 Performing reconfigurations

Reconfigurations can be triggered both at AppAgent and Manager Agent levels by their associate Decision Makers. Decisions are taken when a lower level agent notifies that its controlled component is faulty. Moreover, to prevent faults of the agent itself, higher level agents proactively ask to the controlled agents for the value *HEALTH_STATE* with a frequency T.

As an example of reconfiguration at the ApplAgent level, let's suppose that the node N_3 is starting to work in a degraded manner: the associated agent A_3 notifies the variable

HEALTH_STATE with the value *Degraded*. AA_1 receives the NOTIFY message, and it checks the path availability on the controlled network. There are still more than 2 paths between the clients (see Figure 3) even if one is degraded. In this case three different solutions can be pursued. The first is continuing in the same degraded configuration. Another is to temporarily bypass the node N_3 creating a new logical channel between N_1 and N_4 and waiting for restarting N_3. In this case, the redundancy in terms of paths is reduced because only the first link of the paths is replicated. The third can be to activate a new node N_5 on the host H_2, and to connect it to the client and to the nodes N_1 and N_2, creating new paths. Obviously the different solutions have different costs in terms of time, CPU consuming and paths redundancy. It is responsibility of the DM to select the best one.

We suppose to be in a case where there are not failure/repair dependencies among components and transient phenomena tied to reconfiguration are negligible; then, simple combinatorial models can be evaluated to take the appropriate decision. Assuming that, at given time, the failure probability of a link or component is 10^{-3} when in the Up state, and 10^{-2} when in the $Degraded$ state; for a component which undergoes restart the failure probability becomes $5 * 10^{-3}$ and links and components belonging to the new path have probability $5*10^{-3}$, then the three configurations options can be compared in terms of failure probability P_F. Table 2 summarizes the results of a fault tree analysis, and points out that the best choice is to restart the node N_3.

Policy options	P_F
Working in degraded manner	$1.73848 * 10^{-8}$
Restart node N_3	$5.19695 * 10^{-9}$
Set–up a new path	$4.77510 * 10^{-8}$

Table 2. Policy comparison

In this simple example, the decision can be based on combinatorial models computed a priori. Relaxing the above assumptions makes the analysis more complex by requiring dynamic resolution of state–based models.

5 Conclusions

This work presents an architecture for dependability provisioning which integrates Lira, a light-weight infrastructure for reconfiguring applications, with a model–based Decision Maker. In particular, our goal is to provide a distributed real–time systems with fault–tolerance capabilities. We concentrate only on system reconfiguration as consequence of both faults of software components and host computers that can affect the system.

The work presented in this paper is still ongoing activity and several extensions are currently under investigation to improve and validate our approach. Firstly, Lira infrastructure has to be fault–tolerant itself and different solutions are currently under investigation in this direction. Secondly, for validation purposes a prototype of the case study and a performability evaluation campaign have been planned. Another possible research direction is to improve error diagnosis capabilities of each agent, to better calibrate system reconfiguration.

Acknowledgements This work has been partially supported by the Italian MIUR in the framework of the project “High Quality Software Architectures for Global Computing on Cooperative Wide Area Networks”.

References

[1] J. C. Laprie A. Avizienis and B. Randell. Fundamental concepts of dependability. Technical Report 01-145, LAAS, April 2001.

[2] M. Castaldi, A. Carzaniga, P. Inverardi, and A.L. Wolf. A light-weight infrastructure for reconfiguring applications. In *Proceedings of 11th Software Configuration Management Workshop (to appear), Portland, Oregon (USA)*, May 2003.

[3] M. Castaldi and N. D. Ryan. Supporting Component-based Development by Enriching the Traditional API. In *Proc. of Workshop on Generative and Component-based Software Engineering, Erfurt, Germany*, October 2002.

[4] S. Porcarelli F. Di Giandomenico A. Chohra and A. Bondavalli. Tuning of database audits to improve scheduled maintenance in communication systems. In *Proc. of 20th SAFECOMP*, Budapest, Hungary, 2001.

[5] J. Kramer and J. Magee. Analysing dynamic change in software architectures: A case study. *Proc. 4th Int. Conf. on Configurable Distributed Architecture*, pages 91–100, 1998.

[6] J. Magee. Configuration of Distributed Systems. In M. Sloman, editor, *Network and Distributed Systems Management*. Addison-Wesley, 1994.

[7] M. A. Marsan and G. Chiola. On petri nets with deterministic and exponentially distribuited firing times. In *LNCS*, volume 266, pages 132–145. Springer Verlang, 1987.

[8] A. Bondavalli I. Mura S. Chiaradonna R. Filippini S. Poli and F. Sandrini. DEEM: a tool for the dependability modeling and evaluation of multiple phased systems. In *Proc. of Dependable Systems and Networks*, New York, USA, 2000.

[9] Marshall T. Rose. *The Simple Book: An Introduction to Networking Management*. Prentice Hall, April 1996.

[10] Michel Wermelinger. A Hierarchic Architecture Model for Dynamic Reconfiguration. In *2nd Int. Workshop on Software Engineering for Parallel and Distributed Systems*, 1997.

[11] A. J. Young and J. N. Magee. A flexible approach to evolution of reconfigurable systems. *Proc. of IEE/IFIP Int. Workshop on Configurable Distributed Systems*, March 1992.