Enabling Self-Management by Using Model-Based Design Space Exploration

Tripti Saxena, Abhishek Dubey, Daniel Balasubramanian, Gabor Karsai
Institute for Software Integrated Systems
Vanderbilt University, Nashville, TN

***Abstract*—Reconfiguration and self-management are important properties for systems that operate in hazardous and uncontrolled environments, such as inter-planetary space. These systems need a reconfiguration mechanism that provides recovery from individual component failures as well as the ability to dynamically adapt to evolving mission goals. One way to provide this functionality is to define a model of alternative system configurations and allow the system to choose the current configuration based on its current state, including environmental parameters and goals. The primary difficulties with this approach are (1) the state space of configurations can grow very large, which can make explicit enumeration infeasible, and (2) the component failures and evolving system goals must be somehow encoded in the system configuration model. This paper describes an online reconfiguration method based on model-based design-space exploration. We symbolically encode the set of valid system configurations and assert the current system state and goals as symbolic constraints. Our initial work indicates that this method scales and is capable of providing effective online dynamic reconfiguration.**

I. Introduction

Complex, mission-critical systems, such as space vehicles, are often built up from individual components. The component-based approach is employed for several reasons. Ideally, if each component is individually verified with regard to a safety specification, then after the entire system is assembled, its safety can be inferred from the verification of the individual components. Systems meeting this requirement are said to be *compositional*. While this sort of verification can provide guarantees about safety properties, it is not enough to ensure the success of mission critical systems. The hazardous and uncontrolled environments in which these systems operate can cause entire component failures, which the system must be able to handle. Additionally, the system must also cope with evolving mission requirements in the face of these component failures.

Fortunately, the component based nature of these systems provides a way to deal with both of these issues. In the event of either a component failure or an evolved mission goal, the system can reconfigure itself by swapping faulty components for working ones. This reconfiguration approach is similar to the one used by autonomic computing frameworks. Autonomic computing employs a subsystem that is able to (1) detect occurrences of discrepancies that signify component failures, (2) diagnose and isolate the probable fault sources, and (3) take actions to contain the faults. In this paper, we focus on the last point above: taking actions to contain faults and mitigate their effects. Given a faulty set of components and/or a set of mission goals, we desire to find a valid system reconfiguration that meets these constraints and requirements.

In order for a system to reconfigure itself, it needs at least two things: (1) a list of valid system configurations, and (2) a set of constraints listing the faulty components and mission goals. Given these, the list of valid configurations can be traversed until one is found that satisfies all of the current constraints. The primary problem with this approach is that the state space describing all valid system configurations can be exponentially large in the number of components, making explicit enumeration infeasible. In addition, the set of faulty components and mission goals must be taken into consideration when selecting a configuration. This requires a language for describing constraints that is both intuitive and expressive.

This paper presents our initial work on a framework to provide online reconfiguration that addresses these issues. It is based on a constraint driven approach. The basic method is the following. First, the state-space of valid system configurations is symbolically encoded. When a fault occurs or a mission goal changes, a set of constraints describing these faults and goals is constructed. The state space of valid configurations is then recomputed using these constraints, along with any existing constraints describing system faults and goals. This is done using our design space exploration tool called DESERT [1]. Once a final design space is computed, a single configuration can chosen based on one of several metrics.

Outline: This paper is organized as follows: Section II provides a brief overview of background material. Section III discusses the related works. Sections IV , V and VI provide an overview of our approach. We present a case study in section VII, and we conclude in section VIII.

II. Background

A. Model Integrated Computing

Model Integrated Computing (MIC) [2] has proven itself as a sound method of applying computer based modeling approaches to a variety of problem domains. MIC incorporates the creation of domain-specific, model-based abstractions which serve to capture relevant aspects of a target system. These models can then be programmatically traversed and transformed to produce a variety of domain-specific artifacts. These models are often transformed into alternate but equivalent representations which can be used by external analysis

and simulation tools to verify certain properties of the system [3] [4]. Examples of MIC uses are the generation of real-time schedules from a software model [5], the creation of configuration files to integrate distributed systems [6], or the generation of source code that can be integrated into an existing framework [7].

MIC relies heavily on the use of domain-specific modeling languages to capture relevant characteristics of an object or system of objects. A domain-specific modeling language (DSML) allows a designer to describe objects in terms of the domain rather than in terms of traditional computer languages. The Generic Modeling Environment (GME) [8], [9] is a freely available tool which provides a platform for Model Integrated Computing design and development. Specifically, GME is a configurable and domain-independent modeling environment that supports the creation and instantiation of multiple user defined (domain-specific) modeling languages.

B. Reflex and Healing Approach for Self-Management

One of the approaches to building a self-managing system is based on Reflex and Healing (RH) [10], [11], [12], which is a biologically inspired two stage mechanism for recovering from faults in large distributed real-time systems. The first stage consists of primary building blocks of fault management called reflex engines that are arranged in a hierarchical management structure. They offer pre-specified reactive responses called reflexes to faults as they are discovered.

The next stage of RH architectures is associated with system healing. This refers to a planned reconfiguration of the system at the global level, which is required if no suitable reflex exists for a fault-event or when system performance needs to be optimized after several reflex actions. This step is typically multi-objective in nature and is dependent on several factors such as system goals, resilience to future faults and performance. This architecture has been successfully demonstrated for the BTeV real-time embedded systems project (http://www-btev.fnal.gov/public/hep/detector/rtes/) [13].

The online reconfiguration technique presented in this paper is one possible way of implementing healing in autonomic systems.

C. Design Space Exploration

Complex systems have a large number of choices in terms of the selection of software components and hardware architectures for implementation. A set of all the possible design alternatives forms a *design space*. Each choice can have an impact on the functional and/or para-functional properties of the final implementation.

Design Space Exploration (DSE) is the exploration of design alternatives before implementation of the system. The goal is to search through the large design space to find design alternatives that satisfy a given set of constraints and that are optimal with respect to one or more objective functions. The selection process is further complicated because the objective functions might conflict with each other (e.g. area vs. latency). The exploration process can be performed in steps by first reducing the size of the design space by rejecting unsatisfactory designs, followed by the identification of designs which are best in terms of certain user-defined metrics of fitness. A wide variety of automated and semi-automated DSE techniques like simulation, analytical approaches and combinatorial search algorithms exist.

DESERT : DESERT [1] is a DSE tool that works on a finite set problem where a design space is a finite set and constraints are relations on the elements in that set. Finite set manipulation requires symbolic representation of both the design space and well as the constraints. DESERT encodes the elements of design space set using binary vectors and thus all operations over the set can be represented as Boolean functions, which are then represented as Ordered Binary Decision Diagrams (OBDD)s.

Design spaces are represented hierarchically as an AND-OR-LEAF tree in DESERT. Each design configuration is essentially a well-formed path in the tree representation which originates from the root and consists of a unique trail branching from an OR-node and multiple simultaneous trails branching from an AND-node. This tree ensures assignment of a unique binary code to each configuration. Additionally, the encoding scheme also encodes the attributes such as latency, WCET (Worst Case Execution Time) and constraints. DESERT can efficiently compute two categories of constraints, namely: compatibility constraints and performance constraints. Compatibility constraints specify relations between elements of the design space. These constraints are symbolically represented as Boolean expressions over the Boolean representation of the elements of the design-space. Performance constraints specify bounds on the composite properties of elements in the composed system.

The design space pruning is a process that involves conjuncting the OBDD representing the design space with the OBDD representing the constraints. The primary advantage of this symbolic design space pruning approach is that it is exhaustive. The resulting pruned space consists of all designs which meet the applied design constraints.

DESERT FD: In [14], Neema documents the scalability issues of the OBDD representation in the presence of continuous finite domain variables. To overcome these limitations, a design space exploration tool named DESERT-FD was developed [15]. It contains a hybrid solver which combines the symbolic constraint satisfaction of DESERT with finite domain constraint satisfaction. DESERT-FD also supports a property composition language that allows the modelers to write custom property composition functions.

III. Related Work

One technique of enabling self-management and reconfiguration is to build the fault-protection behavior as an integrated part of the control system during the design process. This is the basis of goal-based control paradigm [16] that supports a deductive controller which is responsible for observing the plant's state (mode estimation) and issuing commands to move

the plant through a sequence of states that achieves the specified goal. Their technique lies in the use of a Reactive Model-based programming language (RMPL) [17] for specifying correct and faulty behavior of the software components.

In principle, this approach is similar to the supervisory control of Discrete Event Systems (DES)[1]. In such systems, one seeks to restrict the behavior of a plant by disabling the events that leads to undesirable behavior such that the supervised system (closed loop of the supervisor and plant) meets the required specifications, which might include "safety" specifications (e.g. prohibit the behavior that can lead to a catastrophe) and "liveness" specifications (e.g. guarantee the eventuality of a specified goal). The seminal work on supervisory control in DES was pioneered by Ramadge and Wonham [18].

Garlan et al. [19] and Dashofy et al. [20] proposed the use of architectural models to represent the system as a composition of several components. They argued that the models capture the component interconnection and the properties of interest and can be used as the formalism upon which system adaptation can be based. Our research is based on similar principles of model driven architecture.

Garlan used reactive rule based strategies for implementing decisions in his work. A typical strategy looked like an if-then-else clause present in several high-level languages. His strategies are similar to reflexes (see section II). Our approach in this work does not use a reactive strategy language. On the contrary, we rely on the constraints specified during design to evaluate system configuration and replace the sub-systems with possible alternatives such that all the constraints are met and the reconfigured system is not faulty.

DESERT has been used earlier for dynamic software reconfiguration in sensor networks [21]. The main difference in this work and [21] is that we allow new constraints to be dynamically added and evaluated during runtime. In [22], Eames et.al. presented an evaluation of the use of DesertFD as a runtime reconfiguration engine for embedded systems. In their application, DesertFD executed on a host machine, connected to the embedded processing platform. Measurement points across the embedded system gather runtime performance metadata and system state, which is fed to the design space explorer on the host. The design space explorer then evaluates the design space and returns a configuration which satisfies the constraints. This configuration is then separately deployed.

IV. Overview of the Approach

This work focuses on self-adaptive component-based systems in which adaptation is limited to activating/deactivating the components and changing their interconnections. Self-adaptive systems typically involve a feedback process with four activities: collect, analyze, decide and act [23]. The process starts by collecting relevant information that reflects the current state of the system. This information is analyzed to diagnose potential performance problems or detect failures. The system then decides how to adapt itself in order to reach a desirable state and the action is taken according to result of the decision activity.

Our set of system architectures is symbolically encoded using a model that consists of alternatives and constraints: the alternatives specify the different ways that a particular piece of functionality may be implemented, while the constraints place further restrictions on which architectures are considered valid. These constraints can be functions over the components and their attributes, and are described in more detail in Section V-C.

Our self-adaptation approach is illustrated in Figure 1. The process starts by monitoring the run-time system for component failures. When a failure is observed, it is translated into a constraint (by the constraint generator) that is then added to the system architecture model; this constraint indicates that the component that failed is no longer a valid choice in system architecture. After all of the constraints describing failed components have been propagated to the architecture model, a search takes place for a new, valid configuration. This search takes place over the new architecture space, which consists of the previously pruned design space (generated when previous components failed) augmented with the new constraints describing the most recent component failures. A set of valid configurations is then obtained as a result of search through this new architecture space, and a particular configuration can be chosen using one of several metrics.

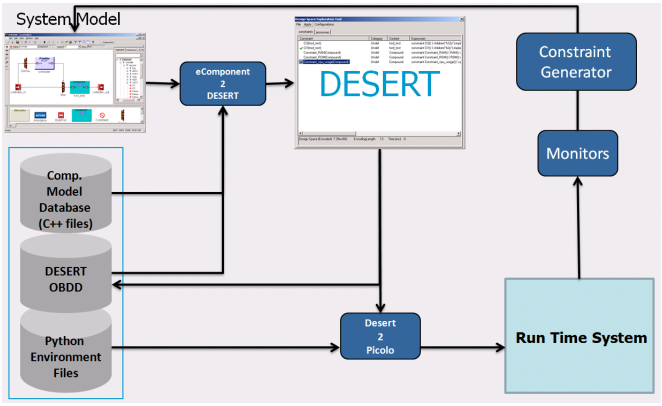

Fig. 1. Adaptation framework

V. Architecture Models

The framework shown in Figure 1 is built on top of a MIC framework. In particular, we use the Generic Modeling Environment (GME) as our main modeling tool because of its support for domain-specific modeling languages (see section II).

In order to obtain the system architecture model that describes alternative configurations and constraints, we follow a three step process as follows.

[1]A Discrete event system (DES) is a discrete state, event-driven system, in which, the state evolution depends entirely on occurrence of discrete events over time.

1) Define a domain-specific modeling language for the system architecture models. This language allows concrete system architecture models to be defined.
2) Compose the language developed in the previous step with a meta-architecture template that we have developed. This composition yields a new domain specific modeling language that enables a designer to model design alternatives and constraints, yielding a design space.
3) Specify any initial design constraints on the system using a combination of graphical and textual constraint specification languages.

The first step above involves capturing the relevant concepts and relationships found in a domain (e.g. subsection V-A), and as such is specific to each individual domain; for more information, see [2]. The following subsections (V-B and V-C) describe the other two steps in more detail with a representative example.

A. ComponentML

ComponentML is a DSML used to capture the basic concepts of component-based systems. Fig. 2 shows the metamodel for ComponentML. A *SoftwareComponent* in ComponentML is the primary unit of implementation, reuse, and composition that exposes a set of *ComponentPorts*. A software component can contain ports that receive (input) or emit (output) requests. Each port is connected to a port on another software component. *InPort*s provide a set of operations to clients of a component instance. *Outport*s provide connection points that allow a component instance to interact with other component instances. The operational parameters like power consumption, latency and other quality-of-service (QoS) properties are captured using attributes of the *SoftwareComponent*. Each component has an attribute `Path` which identifies the path of the executable that is invoked when the *SoftwareComponent* is started. The *HasFailed* attribute tells whether the component is still working or has failed. Components are connected (composed) in order to build an *Assembly* which represents a part-of or a whole application. The communication links between the components are clearly expressed using the *InPort*, *Outport* and *SignalFlow* objects.

The initial goal with ComponentML was to have a visual interface to Picolo [24], a rapid prototyping framework aimed at easing the introduction of components and component-based applications. The component models and properties are graphically specified and then an interpreter generates the corresponding representation in *Picolo*.

B. Meta-architecture

ComponentML needs to also contains elements that allow the user to model more than one implementation of an application in a compact and scalable representation, along with the structural constraints that govern the selection of the appropriate implementation. This is done by metamodel composition with Meta-architecture Template (MT) (see figure 3).

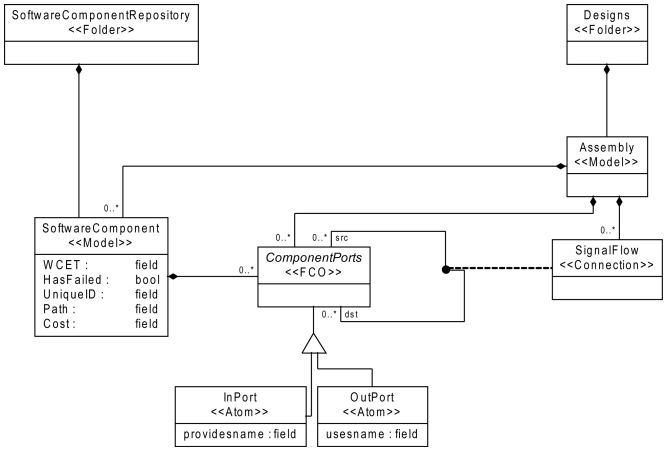

Fig. 2. Original ComponentML

MT is an abstract metamodel template that captures the common patterns in design space exploration. The resulting extended-ComponentML (eComponentML) language can instantiate (concretize and replicate) concepts in MT while still using elements from the original ComponentML. Figure 3 shows part of the MT metamodel. The design space can be structured using *GDSELPrimitive*s, and container objects like *Mandatory*, *Alternative* and *Option*. The *Mandatory* class models composition, which means all the objects contained in a *Mandatory* object must be included in a valid configuration. The *Alternative* class models a choice point where each child object represents one alternative, and exactly one of these alternatives is selected in a valid configuration. This allows mutually exclusive design choices to be modeled. The *Option* class describes configurations in which the child object may or not be selected. A *Primitive* is a basic element representing a fundamental unit of composition.

Data of interest in a design space is captured as properties of the *Primitives*. The value of a property of a Container (Alternative, Mandatory, Option) is calculated as a function of the property values of the contained objects. This function is called the composition function. DESERT supports a limited set of composition functions (e.g. Add, Multiply).

C. Constraints

In addition to specifying design alternatives, our language allows the user to define additional constraints using the attributes of individual components. These constraints further restrict the set of valid architectures. The system requirements are expressed as formal constraints on operational parameters such as power, latency and other QoS properties. Constraints are modeled using graphical or textual constraint objects. The *Graphical Constraints* are used to model dual context (i.e., two objects are involved) constraints that can be instantiated at the model level. The *Textual Constraints* are used to model single context constraints and may or may not be instantiated at the model level. The 'expression' attribute of the constraint objects contains the actual constraint definition expressed in the Constraint Specification Language (CSL), a simple

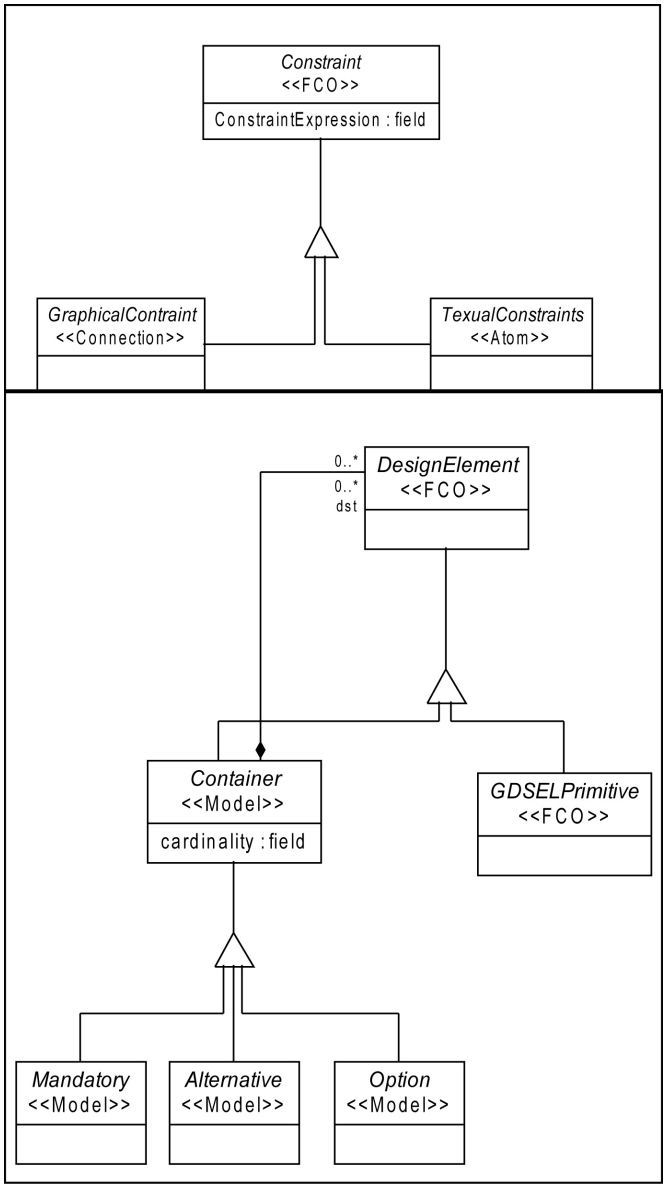

Fig. 3. Meta-Architecture Template

scripting language we developed to facilitate user friendly syntax for writing constraints on elements of the design space.

While the Object Constraint Language (OCL) [25] is the constraint language used and supported by GME, we decided to develop our own language for a number of reasons. Most constraints in the design space are single context (on a single component) or dual context (relating two components). OCL expressions use a single context and become bulky when used in multiple contexts. For example, a constraint between two nodes in the design space tree uses their least common ancestor as the context. OCL requires a number of indirections to access the required nodes. Moreover, there is a need to filter out the model specific details while writing constraints on the design space.

In order to address these limitations, a scripting language

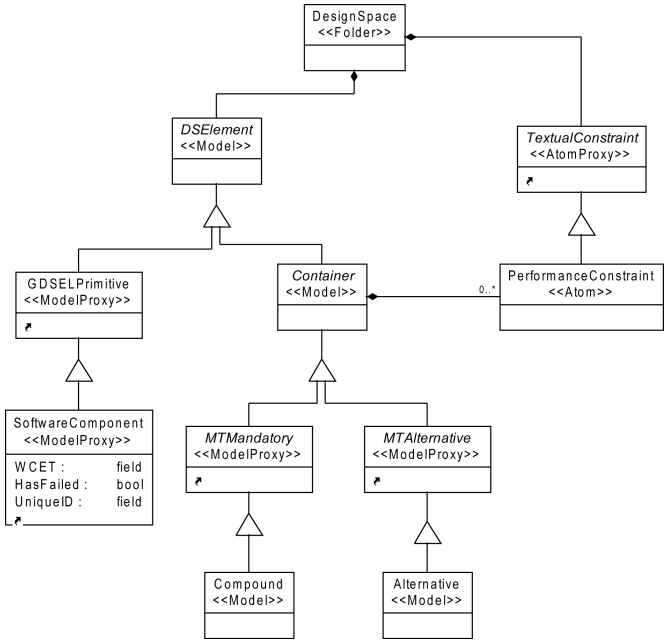

Fig. 4. Extended ComponentML Metamodel

called the Constraint Specification Language (CSL) has been developed to provide user-friendly syntax for writing constraints. Currently, CSL consists of the most essential elements needed to express constraints, which includes two basic data types: property variables and constants. Each property variable refers to an attribute of a model element with the same name. The property variables and constants can have type integer, string or Boolean. CSL supports arithmetic, logical and relational operators. The supported logical operators include conjunction, implication and negation. Operations are used to create expressions in CSL. At the highest level an expression is a logical expression. CSL also supports an "if-then-else" construct which corresponds to the "implies" operator of OCL.

Fig. 4 shows a part of the extended componentML (eComponentML metamodel) language generated after the composition operation. Notice that two new elements - *Compound* and *Alternative* have been added to eComponent to create alternative design spaces. The *SoftwareComponent* belonging to the original language is derived from *Primitive*. The performance constraints are modeled using instances of *PerformanceConstraint*, which is derived from *TextualConstraint* and models single context constraints.

VI. Runtime reconfiguration

Once the architecture model describing the alternative system configurations has been defined and deployed, the monitoring activities shown in Figure 1 begin: the run-time system is monitored, failures are translated into constraints, the design-space exploration tool recomputes the valid architecture and a selected configuration is deployed. The model translators and the DESERT execution engine both run on a host machine connected to the embedded platform.

During the configuration process, the design space model in eComponentML is converted to a format acceptable to DESERT by the `eComponent2DESERT` Interpreter. The converted data is fed to DESERT as an XML file. DESERT applies the constraints present in the model and generates another XML file which contains the valid design configurations in the pruned space.

The DESERT to Picolo interpreter (Execution Engine) is the most important component in our software reconfiguration infrastructure. It is responsible for reading the DESERT output file that contains the set of valid configurations and choosing a single configuration. Once a configuration is selected, the interpreter reads the component model database and creates a component assembly using the Picolo framework. The current configuration selection algorithm works by maintaining a list of components that are currently executing and choosing the configuration that requires the least number of changes from the current configuration.

Once the system is initialized, the Execution Engine goes into an infinite loop in which it listens for reconfiguration data. Once it receives configuration information, it performs the reconfiguration and then goes back to listening for reconfiguration data message. Reconfiguration duties include introducing new components to- and removing old components from the current component assembly. Next, the reconfiguration dynamically changes the wiring (function calls) of the components. The Picolo framework provides libraries for the creation and destruction of the components. The Execution Engine first stops all components that need to be disabled, and then selects (as described above) a single configuration from the configuration set output by DESERT.

The monitor executes on the host machine and receives messages with the state of each component. As soon as it detects the failure of one of the components, it signals the `ConstraintGenerator`, which then updates the eComponentML design space model with additional constraints reflecting the exclusion of the faulty component from all valid configurations. This updating of the design space model invokes the reconfiguration process.

VII. Illustrative Example

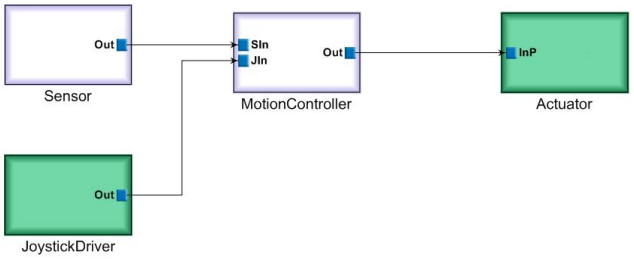

Fig. 5. Motion Controller

Figure 5 shows a simple example of the component-based design of a motion controller use case of an intelligent wheelchair. There are two main functions of the motion controller: sensing and actuating. The motion controller gets data from the wheel sensors and converts it to speed and angle. Actuation is performed by reading the values from the joystick and writing them to the actuator. There are four main components: the *Sensor*, *MovementController*, *JoystickDriver* and the *Actuator*. The *Sensor* component is triggered periodically to publish an event with data. The *MovementController* component is an asynchronous consumer of this event, and it is triggered periodically to consume the last event produced by the sensor. When finished processing, it ends with publishing its own output event. The *Actuator* component is triggered aperiodically to process the *MovementController* event. Upon activation, the *Actuator* component uses an interface provided by the *MovementController* to retrieve the position and speed data via a synchronous call.

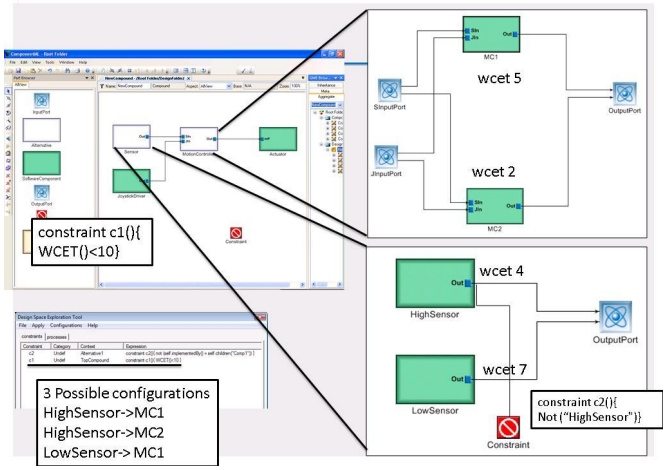

Fig. 6. Runtime Reconfiguration of Motion Controller

Design Spaces: The application has four implementations which are hierarchically organized using alternatives and primitives. The *Sensor* object in the model is an Alternative that models a choice point between two sensors of different precisions. The choices modeled using *HighSensor* and *LowSensor* have the same interface but different precisions. Similarly, the *MovementController* object models two different implementations: *MC1* and *MC2*. The first implementation, *MC1*, takes input data from the *JoystickDriver* and sends it to the *Actuator*, and the second implementation, *MC2*, is similar to *MC1* but also converts the values from the joystick to the standard form and sends them to the Display as well. The meta data (e.g., WCET) is captured as attributes. The model also captures constraints that are evaluated over the design space, at both design-time and run-time. The performance constraints in this example are:

`C1.wcet()` $< L_{bound}$

`C2.cost()` $< C_{bound}$

where C_{bound} is the upper bound on the cost of the implementation and L_{bound} is the upper bound on the wcet of the implementation (e.g. L_{bound} = 10 shown in Fig. 6).

While the *HighSensor* can be used with both implementations of the motion controller, the *LowSensor* can be used only with *MC1* implementation restricting the number of valid configurations to three. This restriction can be modeled using the following compatibility constraint:
`C3.LowSensor` $\rightarrow$ `MC1`

Runtime: We prune the design space using DESERT, which provides configurations that satisfy the constraints (e.g. overall cost and WCET). WCET is an additive property, meaning that the WCET of the implementation is the sum of WCET of the selected components. Once the initial configuration is selected, the following sequence of events illustrates the online reconfiguration in this example.

1) Let the initial selected configuration consist of *HighSensor* and *MC1*.
2) We simulate injection of a fault that indicates failure of the *HighSensor*. This failure is detected by the monitor.
3) The monitor signals the `ConstraintGenerator`, which then generates a constraint and updates the system model. Figure 6 shows the new constraints that are added to the system model. Valid configurations are restricted from including the failed sensor.
4) Updating of the system model triggers reevaluation of the design space. DESERT returns a set of valid configurations in the presence of this additional constraint.
5) The ExecutionEngine selects a configuration from the set of valid configurations and performs the steps necessary for reconfiguration of the system.

VIII. Conclusion and Future Work

Reconfiguration is an important technology for mission-critical systems. This paper presented an approach for constraint-guided software reconfiguration of component-based systems using our design-time design space exploration tool, DESERT. Our approach requires monitoring the running system for component failures, at which time the reconfiguration process is invoked. We demonstrated the applicability of this approach with a simple illustrative example.

Although the proposed reconfiguration framework is fairly straightforward, it highlights some of the challenges that need to be addressed for self-adaptation. The most important challenge is to reduce the latency of the reconfiguration process. Every invocation of the reconfiguration framework invokes DESERT, which performs the design space pruning with additional constraints. This can be very expensive, especially for dynamic applications that must be reconfigured quickly and have a high frequency of reconfigurations. DESERT uses symbolic constraint satisfaction which does not scale well in the presence of mathematical constraints and continuous finite domain variables. At present, the reconfiguration framework chooses a system configuration that is closest to the current configuration. More complex strategies of choosing the configuration can be implemented in future versions of the framework. Possibilities include implementing optimization based reconfiguration where the system configuration is chosen based on a certain cost function.

References

[1] S. Neema, J. Sztipanovits, G. Karsai, and K. Butts, "Constraint-based design-space exploration and model synthesis," in *EMSOFT*, 2003, pp. 290–305.

[2] J. Sztipanovits and G. Karsai, "Model-integrated computing," *Computer*, vol. 30, no. 4, pp. 110–111, 1997.

[3] S. Abdelwahed and W. Wonham, "Interacting des: modelling and analysis," in *Systems, Man and Cybernetics, 2003. IEEE International Conference on*, vol. 5, 5-8 Oct. 2003, pp. 4222–4229vol.5.

[4] A. Ledeczi, J. Davis, S. Neema, and A. Agrawal, "Modeling methodology for integrated simulation of embedded systems," *ACM Trans. Model. Comput. Simul.*, vol. 13, no. 1, pp. 82–103, 2003.

[5] J. Porter, G. Karsai, and J. Sztipanovits, "Towards a time-triggered schedule calculation tool to support model-based embedded software design," in *EMSOFT*, 2009, pp. 167–176.

[6] A. Gokhale, K. Balasubramanian, and T. Lu, "Cosmic: addressing crosscutting deployment and configuration concerns of distributed real-time and embedded systems," in *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. New York, NY, USA: ACM, 2004, pp. 218–219.

[7] J. Gray, J. Sztipanovits, D. Schmidt, T. Bapty, S. Neema, and A. Gokhale, "Two-level aspect weaving to support evolution of model-driven synthesis," *Aspect-Oriented Software Development*, no. Chapter 30, pp. 681–710, 2004, addison-Wesley.

[8] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi, "The Generic Modeling Environment," in *Workshop on Intelligent Signal Processing, Budapest, Hungary*, vol. 17, May 2001.

[9] The Generic Modeling Environment, http://www.escherinstitute.org/Tools/GME.asp.

[10] S. Neema, T. Bapty, S.Shetty, and S.Nordstrom, "Developing autonomic fault mitigation systems," *Journal of Engineering Applications of Artificial Intelligence Special Issue on Autonomic Computing and Grids*, 2004.

[11] R. Sterritt and D. Bantz, "Personal autonomic computing reflex reactions and self-healing," *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, vol. 36, no. 3, pp. 304–314, May 2006.

[12] A. Dubey, S. Nordstrom, T. Keskinpala, S. Neema, T. Bapty, and G. Karsai, "Towards a verifiable real-time, autonomic, fault mitigation framework for large scale real-time systems," *Innovations in Systems and Software Engineering*, vol. 3, pp. 33–52, March 2007.

[13] S. A. et al., "RTES demo system 2004," *SIGBED Rev.*, vol. 2, no. 3, pp. 1–6, 2005.

[14] S. Neema, "System-level synthesis of adaptive computing systems," Ph.D. dissertation, Vanderbilt University, May 2001.

[15] B. K. Eames, S. K. Neema, and R. Saraswat, "Desertfd: a finite-domain constraint based tool for design space exploration," *Design Automation for Embedded Systems*, 2009.

[16] B. C. Williams, M. Ingham, S. Chung, P. Elliott, M. Hofbaur, and G. T. Sullivan, "Model-based programming of fault-aware systems," *AI Magazine*, vol. 24, no. 4, pp. 61–75, 2004.

[17] B. Williams, M. Ingham, S. Chung, and P. Elliott, "Model-based programming of intelligent embedded systems and robotic space explorers," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 212–237, 2003.

[18] P. Ramadge and W. Wonham, "Supervisory control of a class of discrete event processes," *Siam J. Control and Optimization*, vol. 25, no. 1, 1987.

[19] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *Computer*, vol. 37, no. 10, pp. 46–54, 2004.

[20] E. M. Dashofy, A. van der Hoek, and R. N. Taylor, "Towards architecture-based self-healing systems," in *WOSS '02: Proceedings of the first workshop on Self-healing systems*. New York, NY, USA: ACM Press, 2002, pp. 21–26.

[21] S. Kogekar, S. Neema, and X. Koutsoukos, "Dynamic software reconfiguration in sensor networks," in *Proc. Systems Communications*, Aug. 14–17, 2005, pp. 413–420.

[22] B. Eames, "On the use of desertfd as a reconfiguration engine for embedded systems," in *Proc. IEEE Mountain Workshop on Adaptive and Learning Systems*, Jul. 24–26, 2006, pp. 127–132.
[23] J. Kephart and D. Chess, "The vision of autonomic computing," *IEEE Computer*, 2003.
[24] R. Marvie, "Picolo: A simple python framework for introducing component principles," in *In Euro Python Conference 2005*, 2005.
[25] R. S. C. et al., *Object Constraint Language Specification ver 1.1*, Sept 1997.