CONSTRAINT-GUIDED SELF-ADAPTATION

Sandeep Neema and Akos Ledeczi
Institute for Software Integrated Systems, Vanderbilt University
Nashville, TN 37235, USA
{neemask, akos}@isis.vanderbilt.edu

Abstract. *We present an approach to self-adaptive systems utilizing explicit models of the design-space of the application. The design-space is captured by allowing the specification of alternatives for any component at any level in the model hierarchy. Non-functional requirements and additional knowledge about the system are captured in the form of OCL constraints parameterized by operational parameters, such as latency, accuracy, error rate, etc, that are measured at run-time. The constraints and the models are embedded in the running system forming the operation-space of the application. When changes in the monitored parameters trigger a reconfiguration, the operation space is explored utilizing a symbolic constraint satisfaction method relying on OBDDs. Once a new configuration that satisfies all the constraints is found the reconfiguration takes place.*

INTRODUCTION

This paper presents an approach to building self-adaptive embedded systems based on Model-Integrated Computing (MIC) [2][3]. In MIC, domain-specific, multiple-aspect models represent the application, its environment and their relationships. Model interpreters translate the models into the input languages of static and dynamic analysis tools, while other model interpreters synthesize software applications running in a real-time, dynamic, macro-dataflow execution environment. Making the design-time models available at run-time by embedding them in the application makes the system *reflective*, a key requirement for self-adaptivity. [1] presents an Embedded Modeling Infrastructure (EMI) that extends MIC into the self-adaptive systems arena. This work builds on the results of that research.

Conventional practices in embedded system development involve working with single-point designs. This, in effect, implies elimination of component and system design alternatives in the early stages of the development process. Such elimination, in the absence of adequate system-level contextual information, leads to sub-optimal and inflexible designs that are difficult to maintain and evolve as system requirements change. Therefore, capturing the *design-space* of the application is advantageous even for conventional systems. For self-adaptive systems, it is mandatory. It is exactly (a subset of) this design-space that can be embedded in the running application in the form of embedded models that forms the *operation-space* of the system. The process of adaptation is the transition from one element of this space to another.

The key question in self-adaptive systems is how to decide what the new configuration should be. This can be considered a search problem in the operation-space. This paper introduces an approach to guide this search based on (1) parameterized constraints captured in the models and embedded in the running system, and (2) symbolic constraint satisfaction at run-time utilizing Ordered Binary Decision Diagrams (OBDD).

The paper is organized as follows. The next section summarizes existing results in embedded model representation as well as, design-space and constraint representation methods. Then, we describe our constraint-based self-adaptation technique in detail.

BACKGROUND

Embedded Modeling

The basic structure of embedded model-integrated systems is illustrated in Figure 1. The Embedded Modeling Infrastructure (EMI) can be best viewed as a high-level layer at the top of the architecture, while a classical embedded systems kernel is located at the bottom. The component that connects these two layers is the translator that we call the embedded interpreter. The embedded models provide a simple, uniform, paradigm-independent and extensible API (EMI API) to this interpreter [1]. These models typically contain application models using some kind of dataflow representation. They may also describe available hardware resources and contain other domain-specific information.

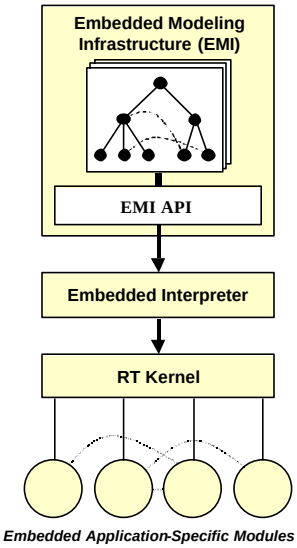

Figure 1: Embedded Modeling

Besides the kernel, the modeling system and the embedded interpreter, the fourth major component of the computing system is the set of software modules that perform the actual task of the embedded system. These are objects executable by the kernel and responsible for the core functionality of the system.

The EMI system provides capabilities for each of the three general tasks essential to self-adaptive systems: monitoring, decision-making, and reconfiguration. To facilitate the implementation of monitoring algorithms, convenient access to operational parameters such as resource utilization, performance values, error rates etc., is required. The embedded models provide uniform representation of these parameters. Objects in the model may have designated attributes (monitor attributes) set by the underlying modules: either by the embedded kernel (in case of most operation system parameters), or by any of the application specific task modules that have information on the operation of the system.

The second and most critical component of self-adaptive applications is the one making reconfiguration decisions. The constraint-based technique we are introducing in this paper provides an approach to his problem. The third task, reconfiguration, is done by the embedded interpreter utilizing the EMI API.

Design-Space Representation

Despite their numerous advantages, there is a lack of formalized methods for representing design spaces in embedded systems design research. In general, existing approaches can be grouped into two categories:

1. *Explicit Enumeration of Alternatives* – different design alternatives are explicitly enumerated. The design space is a combinatorial product of the design alternatives. Characteristically different designs may be obtained by selecting different combinations of alternatives.
2. *Parametric* – the design variations are abstracted into single or multiple parameters. The cross-product of the domains of the configuration parameters forms a parameterized design space. Physically different designs may be obtained from the parameterized design space by supplying appropriate value for the configuration parameters.

In the rest of this section, we present a review of some research and technologies where explicit representation of design space is considered.

Design Space Modeling with Alternatives

The Model-Integrated Design Environment (MIDE) for Adaptive Computing Systems (ACS) introduced in [4] targets multi-modal structurally adaptive computing systems. One of the key-features of this model-integrated framework is its support for explicit representation of design spaces for embedded adaptive systems. Representation of design spaces has special significance to multi-modal adaptive computing systems. The diverse functionality desired in the different modes of operation makes optimization decisions extremely difficult. Mode-level optimization does not imply system-level optimization as the reconfiguration cost involved in transitioning from a mode to another may offset any efficiency attained by a mode-optimized implementation. In order to address these challenges, a design flow has been developed that involves constructing large design spaces for the targeted system and then using constraints to guide the search through the large design space for system synthesis.

In this approach, an adaptive computing system is captured in multi-aspect models. The different modes of operation and the operational behavior of an adaptive system are captured as a hierarchical parallel finite state machine in a StateChart-like formalism [5]. The resources available for system execution are captured as an architecture flow diagram. The computations to be performed in the different modes of operations are captured as a hierarchical dataflow with alternatives. The basic dataflow model captures a single solution for implementing a particular set of functional requirements. In this framework the basic dataflow representation has been extended to enable representation of design alternatives. With this extension a dataflow block may be decomposed in two different ways. The first type is hierarchical decomposition in which a dataflow block can encapsulate a functionality described as a dataflow diagram. The second type is an orthogonal decomposition, in which a dataflow block contains more than one dataflow block as alternatives. In this case, the container block defines only the interface of the block and is devoid of any implementation details. The dataflow blocks contained within the container define different implementations of the interface specifications. With these extensions (i.e. hierarchy and alternatives), a dataflow model can modularly capture a large number of different computational structures together to form an exponentially large design space.

The alternatives in a dataflow may take many different forms. Alternatives may be technology alternatives that are different technology implementations of a defined functionality—e.g. TI-DSP C40 (software) implementation vs. a TI-DSP C67 (software) implementation vs. a VIRTEX® FPGA (hardware) implementation of a cross-correlation component. Technology alternatives minimize the dependency of the system design on the underlying technology, thereby enabling technology evolution. Alternatives may also be algorithmic alternatives that are different algorithms implementing a defined

functionality (e.g. spatial vs. spectral correlation of a 2D image). It is generally accepted that the best performance can be obtained by matching the algorithm to the architecture or vice-versa. When different algorithm alternatives are captured, it may be possible to optimize the system design for a range of different architectures by choosing from different algorithm alternatives. Alternatives may also be functional alternatives that are different (but related) functions obeying the same interface specifications (e.g. a 3x3-kernel convolution vs. a 5x5-kernel convolution). Often in the design cycle of a system, functional requirements change when the system is scaled up, or better precision implementations of a function are desired due to improvements in sensor fidelity, availability of more compute power, etc. Functional alternatives are valuable in accommodating a large range of functional requirements in a design in such situations.

In summary, a design space composed by capturing alternatives can encapsulate a large number of characteristically different solutions for an end-to-end system specification. While large design spaces are valuable in improving design flexibility and optimization opportunities, determining the best solution for a given set of performance requirements and hardware architecture can be a major challenge. [4] describes a constraint-based design space exploration method that meets this challenge. This paper advocates extending this approach to run-time for dynamic self-adaptation.

Generative Modeling

Modeling design alternatives explicitly provides much more flexibility than capturing a single point solution. However, it still requires the user to pre-design all the components and their possible interconnection topologies. The user (or an automatic tool) can pick and choose which alternative to select from a fixed set. A complementary approach, called generative modeling, is a combination of parametric and algorithmic modeling [1]. With this technique, the elementary components are modeled as before, but their number and interconnection topology are specified algorithmically in the form of a generator script. Generator scripts can refer to the values of architectural (numerical) parameters contained in the models. This approach is very similar to the VHDL generate statement; they both support the concise modeling of repetitive structures.

Generative modeling inherently supports dynamic reconfiguration. The generator scripts can be compiled as part of the runtime system. Runtime events can change the values of architectural parameters triggering the generator scripts. Note, however, that an extra level of indirection is needed here; the generators should not reconfigure the runtime system directly. Instead they should reconfigure a representation of the running system, the embedded models in order to be able to analyze the system before the actual reconfiguration takes place [1].

Constraint Representation

Constraints are integral to any design activity. Typically, in an embedded system design constraints express SWEPT (size, weight, energy, performance, time) requirements. Additionally, they may also express relations, complex interactions and dependencies between different elements of an embedded system viz. hardware, middleware, and application components. Ideally, a correct design must satisfy all the system constraints. In practice, however, not all constraints are considered critical. Often trade-offs have to be made and some constraints have to be relaxed in favor of others. Constraint management is a cumbersome task that has been inadequately emphasized in embedded systems research. Most embedded system design practices place very little emphasis on constraints and treat them on an ad-hoc basis, which means either testing after the implementation is complete, or an over-design with respect to critical parameters. Both of these situations can be avoided by elevating constraints to a higher level in the design process. Two important steps in that direction are a) formal representation of constraints; and b) verification/pre-verification of the system design with respect to the specified constraints.

Principally, three basic types of design constraints are common to embedded systems: (a) performance constraints, (b) resource constraints, and (c) compositional constraints. More complex constraints are typically combinations of one or more of these basic types joined by first order logic connectives.

Performance constraints – Performance constraints express non-functional requirements that a synthesized system must obey. These may be in the form of size, weight, energy, latency, throughput, frequency, jitter, noise, response-time, real-time deadlines, etc. When an embedded computational system is expressed in a dataflow description, these constraints express bounds over the composite properties of the computational structure. Following are some common examples:

· Timing – expresses end-to-end latency constraints, specified over the entire system, or may be specified over a subsystem e.g. (latency < 20).

· Power – expresses bound over the maximum power consumption of a system or a subsystem e.g. (power < 100).

Resource constraints – Resource constraints are commonly present in embedded systems in the form of dependencies of computational components over specific hardware components. These constraints may be imperative in that they may express a direct assignment directive, or they may be conditionalized with other computational components. Following is an example of a resource constraint in plain English:

· Imperative – component FFT must be assigned to resource DSP-1

· Conditional – if component FFT is assigned to resource DSP-1 then component IFFT must be assigned to resource DSP-2

Compositional constraints – Compositional constraints are logic expressions that restrict the composition of alternative computational blocks. They express relationships between alternative implementations of different components. These are essentially compatibility directives and are similar to the type equivalence specifications of a type system. Therefore, compositional constraints are also referred to as typing constraints. For example, the constraint below expresses a compatibility directive between two computational blocks FFT and IFFT that have multiple alternate implementations: {if component FFT is implemented by component FFT-HW then implement component IFFT with component IFFT-HW}.

The Object Constraint Language (OCL), a part of the Universal Modeling Language (UML) suite, forms a good basis for expressing the type of constraints shown above. OCL is a declarative language, typically used in object modeling to specify invariance over objects and object properties, pre- and post- conditions on operations, and as a navigation language [8]. [6] advocates an extended OCL to express the type of constraints specified above. The constraints are specified in the context of an object. A constraint expression can refer to the context object and to other objects associated with the context object and their properties. The OCL keyword *self* refers to the context object. Role names are used to navigate and access associated objects. For example, the expression self.parent evaluates to the parent object of the context object, similarly self.children evaluates to a set of children object of the context object.

A constraint expression can either express direct relation between the objects by using relational or logical operators, or express performance constraints by specifying bounds over object properties. Object properties can be referred to in a manner similar to associations. Property constructs supported in the derived constraint language include latency, power, resource assignment, etc.

SELF-ADAPTIVE SYSTEM ARCHITECTURE

Figure 2 shows the overall architecture of a self-adaptive system under our approach. It should be noted that this architecture builds upon the EMI discussed earlier in the background section and in [1]. We improve upon the existing architecture by introducing a new reconfiguration controller that employs a constraint-based operation space exploration for determining the next configuration. Parameterized constraints and a symbolic constraint-based operation space exploration form the essence of this new reconfiguration controller. Constraints in this approach provide a way of mapping operational requirements to reconfiguration decisions. The modeling paradigm, elaborated later in this section allows capturing a large design space for the system implementation, along with constraints. The paradigm supports capturing both design-time as well as runtime constraints. The system design space is pruned using the design-time constraints, and a small subset of the design space is retained at runtime as the operation-space of the system. It is this reduced space that is explored at runtime using the parameterized runtime constraints. In the rest of this section, we elaborate upon the modeling paradigm for the embedded model representation, the constraint-based operation space exploration method, and the functioning of the reconfiguration controller.

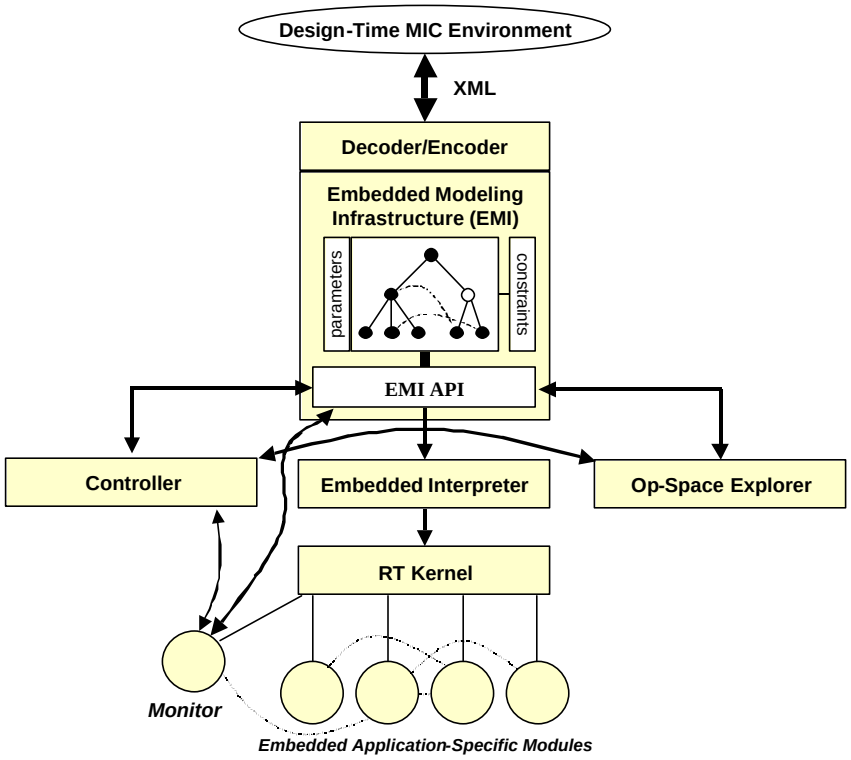

Figure 2: System Architecture

Modeling Paradigm

The modeling paradigm, i.e. the modeling language, that supports designing embedded self-adaptive systems using constraint-based adaptation methods is a derivative of the one presented in [4] described previously. It has three primary components:

- application models that represent the desired functionality of the system using hierarchical signal flow diagrams with explicit alternatives, thereby capturing the design-space of the application,
- resource models that capture the available hardware resources and their interconnection topology,
- constraints that describe non-functional requirements, resource constraints and other information about the system and are parameterized by operational variables of the running system.

The novel idea is the concept of parameterized constraints. Each constraint can be associated with one or more parameters that capture values that are continuously measured during the operation of the system. These parameters also have multiple thresholds specified. Whenever a value crosses a threshold, the controller is triggered which, in turn, starts the operation-space exploration.

Constraints are also prioritized. Constraints capturing critical requirements get the highest priority. These will always have to be satisfied. However, the system needs to have the flexibility to relax non-critical constraints in case there are no solutions in the operation space that satisfies all the constraints. Priorities specify the order of automatic constraint relaxation.

Note that currently generative modeling is not considered in our constrained-based approach to self-adaptivity. The main reason for this is the difficulty with symbolic constraint satisfaction. If the generator language is Turing complete, which is highly desirable for the expressive power, symbolic representation of the generator script is a very hard problem. A possible approach is to constrain the values of generative parameters and analyze the restricted (now finite) design space. This would diminish the advantages of generative modeling itself—the flexibility and the infinite design space. The only alternative is analyzing a single instance of the generative models that corresponds to a particular instantiation of the parameter set. However, the search for the new configuration in the parameter space needs to consider many such instances. Regenerating and analyzing every candidate would be computationally prohibitively expensive.

Operation Space Exploration

The objective of the operation space exploration is to find a single feasible solution from the space that satisfies all the critical constraints and maximally satisfies the non-critical constraints. The challenge of operation-space exploration emerges from the size of the space, complexity of the requirements and criteria expressed as constraints, and the strict resource and time bounds over the exploration process. This paper proposes the use of symbolic methods based on Ordered Binary Decision Diagrams (OBDD-s) for constraint satisfaction. The highlight of the symbolic constraint satisfaction method is the ability to *apply constraints to the entire space without enumerating point solutions*, whereas an exhaustive search by enumeration through the space is generally exponential time complexity. Symbolic analysis methods represent the problem domain implicitly as mathematical formulae and the operations over the domain are performed by symbolic manipulation of mathematical formulae.

The symbolic constraint satisfaction problem considered here is a finite set manipulation problem. The operation-space is a finite set. Constraints specify relations in this space. Constraint satisfaction is a restriction of the solution space with the constraints. Solving this finite set manipulation problem symbolically requires the solution of two key problems: (a) symbolic representation of the space, and (b) symbolic representation of the constraints.

Symbolic constraint satisfaction is simply the logical conjunction of the symbolic representation of the space with the symbolic representation of the constraints. Figure 3 illustrates the process of symbolic constraint satisfaction.

Symbolic Representation of the Operation Space

The key to exploit the power of symbolic Boolean manipulation is to express a problem in a form where all of the objects are represented as Boolean functions [7]. By introducing a binary encoding of the elements in a finite set all operations involving the set and its subsets can be represented as Boolean functions. In order to represent the operation space symbolically, the elements of the operation space have to be encoded as binary vectors. The choice of encoding scheme has a strong impact on the scalability of the symbolic manipulation algorithms [7]. An encoding scheme has been developed in [6] after a careful analysis of the problem domain, taking into consideration the hierarchical structure of the solution space.

The operation space captures feasible configurations for implementing the system functionality, and is represented as a hierarchical dataflow graph with alternatives, as described earlier. The dataflow is associated with a network of resources in defining the system configurations. This representation can modularly define a very large space. The complete operation space is a set of possible system configurations. The encoding scheme assigns encoding values to each node in the hierarchy such that each configuration receives a unique encoding value. Additionally, the encoding scheme must also encode the resource assignments of components along with performance attributes such as latency, throughput, power, etc. The performance attributes take numeric values from a continuous finite domain. However, for the purpose of encoding the domains of the attributes are discretized. The total number of binary variables required to encode the operational space is primarily dependent upon on the domain size and the quantization levels in the domain [6]. With this encoding the operation space is symbolically composed as a Boolean function from the symbolic Boolean representation of components. After deciding the variable ordering this Boolean representation is mapped to an OBDD representation in a straightforward manner.

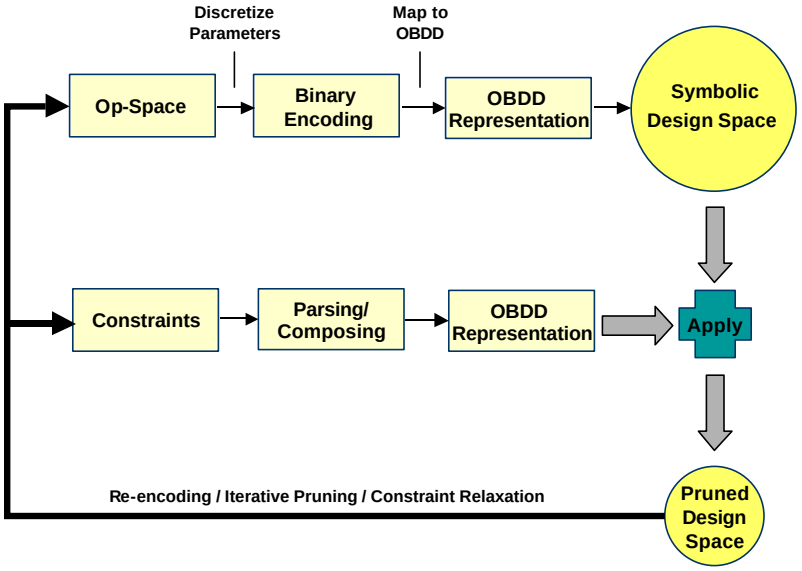

Figure 3: Symbolic Constraint Satisfaction

Symbolic Representation of Constraints

Three basic categories of constraints are considered. Symbolic representation of each of these categories of constraints is summarized below.

Compositional constraints – Compositional constraints express logical relations between processing blocks in the hierarchical dataflow representation. Symbolically the constraint can be represented as a logical relation over the OBDD representation of the processing blocks trivially.

Resource constraint – Resource constraints relate processing blocks to resources. Symbolic representation of resource constraints is accomplished by expressing the relation over the OBDD representation of the processing block and resource.

Performance constraints – Performance constraints are more challenging to solve symbolically than the previously specified categories of constraints. There are two primary drivers of the complexity: 1) A system-level property has to be composed from component-level properties in a large design space, and 2) The property being composed is numeric, and may admit a potentially very large domain. Representing a large numeric domain symbolically as a Boolean function and performing arithmetic operations symbolically is a challenging problem with serious scalability concerns. In general different performance attributes compose differently. An approach for expressing constraints over additive attribute symbolically has been detailed in [6]. The basic approach involves expressing linear arithmetic constraints over a group of binary vectors, each binary vector representing an integer variable, as a Boolean function. This function is then conjuncted with the representation of the operation space that encodes performance attributes of components as integer values for different binary vectors. The binary vectors are then quantified out from the resulting function. Thus, in effect a relation over the attributes of components is composed into a relation over the components of the operation space. Building on this basic approach more complex composition of system-level properties, and symbolic representation of different performance constraints has been shown in [6].

In addition to these basic categories of constraints, complex constraints may be expressed by combining one or more of these constraints with first order logic connectives. The symbolic representation of the complex constraints can be accomplished by composing the symbolic representation of the basic constraints.

Embedding Symbolic Constraint Satisfaction

The key issues in embedding the symbolic constraint satisfaction methodology outlined above are to manage the memory footprint of the OBDD data structures, and to control the potential non-deterministic exponential blow-up of OBDD-s. In the presented approach we have a rudimentary technique to manage both the memory footprint and the exponential blow-up by limiting the maximum node count in the OBDD data structure. Doing this forces the OBDD algorithms to throw an exception whenever the data-structure grows beyond the limit. Such an exception is handled by asserting the target constraint unsatisfiable within the given resource and time budget.

An additional optimization for embedding symbolic constraint satisfaction enables incremental constraint satisfaction. Thus, when a single or a small set of constraints is activated by a change in an operation parameter, and the change further constrains the operation space, then it does not necessitates re-exploration/re-application of all the constraints. Only the affected constraints are expressed symbolically as OBDD-s and re-applied. However, this approach does not work when the parameter change relaxes the constraint. In this case, it may be possible to avoid re-application of all the constraints by caching intermediate symbolic representation of the pruned operation. Caching, however, increases the memory

requirements. Thus, a trade-off has to be made between the memory bounds and the time bounds of the exploration.

Reconfiguration Controller

The controller continuously monitors the operational parameters of the system. The embedded models capture the dependencies between the specified constraints and the operational parameters. The dependency specifications include multiple threshold levels for the operational parameters. Whenever any one of these parameters crosses any of the threshold levels, an operation space exploration is invoked with the affected constraints. Additional critical and non-critical constraints are also passed to the operation space exploration for satisfaction. The exploration process is time-bounded by specifying a maximum OBDD node count. The exploration process may result in an exception (for exceeding the maximum node count) or zero, one, or more possible configurations for the system implementation. The four scenarios are individually considered below:

Exception – In the event of an exception due to a non-critical constraint, the exploration continues with the specified non-critical constraint dropped from the constraint store. If the exception is due to a critical constraint then the system execution continues with the current system configuration, however, the system re-attempts exploration with an increased maximum OBDD node count. A number of re-tries limited by a pre-determined constant are made, until the exploration results in one or more configuration that satisfies all the critical constraints, and maximally satisfies non-critical constraints. The system continues to operate with the current configuration until a new configuration is found.

Zero – In this case the relaxation of non-critical constraints is attempted progressively. The progression continues until one or more configurations are found. If no configuration is found even after relaxing all non-critical constraints, no further attempts for exploration are made and the system continues to execute with the current configuration. It must be noted here that by progressively relaxing non-critical constraints and accepting the first configuration that emerges by relaxing constraints does not guarantee maximal satisfaction of non-critical constraints. The system sacrifices maximality in favor of finding at least one working configuration.

One – This is an ideal scenario, when the operation space exploration results in exactly one valid configuration. The system simply accepts the resulting configuration as the next configuration.

Multiple – The system can be compiled with several different strategies to handle this scenario. A simple strategy is to pick the first configuration from the result set, as they are all equally fit from a constraint satisfaction perspective. A more complex strategy attempts to evaluate the difference between the current configuration and the configurations in the result set. Configuration with the least difference is accepted as the next configuration.

Once the next configuration has been accepted, the reconfiguration controller passes the control to the system reconfiguration manager that performs the reinterpretation and reconfiguration process.

CONCLUSIONS

This paper presented an approach for constraint-guided self-adaptation of embedded systems. Embedded models of the system contain multiple potential system configurations captured as alternatives in an operation space. Parameterized constraints provide a way of capturing changing operational requirements. An embedded operation space exploration, triggered by changes in operational parameters, rapidly finds next system configuration that satisfies the current operational constraints, which is then instantiated and deployed through our Embedded Modeling Infrastructure.

The main contribution of the work is the systematic approach to the reconfiguration controller, the key component of any self-adaptive system. The OBDD-based symbolic satisfaction of constraints parameterized by monitored operational variables makes the controller *reusable* across applications and even application domains. It can potentially replace the ad-hoc, highly application-specific, hand-crafted reconfiguration controllers of the past. Another principal benefit of this approach is that the every system configuration that is deployed is *correct by design*, i.e. it already satisfies all the correctness criteria specified by the constraints.

An ideal application area for our approach is the domain of robust, fault-tolerant embedded systems. The operation space can contain various system configurations that are tailored for different system failure modes. Constraints capture the complex relationships between different failure modes and system configurations. The system can quickly find a new configuration and adapt after a component failure.

There are numerous open research issues associated with the approach and a lot of work remains to be done. The most difficult problem is the operation-space exploration within strict time bounds and utilizing possibly limited computational resources. A systematic approach to over- or under-constrained systems needs to be developed. Finally, the technique needs to be implemented and tested in real-world scenarios.

ACKNOWLEDGEMENT

The research presented here was made possible by the generous sponsorship of the Defense Advanced Research Projects Agency (DARPA) under contracts F30602-96-2-0227 and DABT63-97-C-0020.

REFERENCES

[1] Ledeczi A., Bakay A., Maroti M.: "Model-Integrated Embedded Systems," in Robertson, Shrobe, Laddaga (eds): Self Adaptive Software, Springer-Verlag Lecture Notes in Computer Science, #1936, February, 2001

[2] J. Sztipanovits, G. Karsai, "Model-Integrated Computing," IEEE Computer, pp. 110-112, April, 1997.

[3] Ledeczi A., Maroti M., Karsai G., Nordstrom G.: "Metaprogrammable Toolkit for Model-Integrated Computing," Engineering of Computer Based Systems (ECBS-99) , pp. 311-317, Nashville, TN, March, 1999.

[4] Bapty T., Neema S., Scott J., Sztipanovits J., Asaad S.: "Model-Integrated Tools for the Design of Dynamically Reconfigurable Systems", ISIS Technical Report/Vanderbilt University, 2000.

[5] Harel, D., "Statecharts: A Visual Formalism for Complex Systems", Science of Computer Programming 8, 1987, pp.231-274.

[6] Neema S., "Design Space Representation and Management for Model-Based Embedded System Synthesis", Technical Report ISIS-01-203, February 2001.

[7] Bryant R., "Symbolic Manipulation with Ordered Binary Decision Diagrams," School of Computer Science, Carnegie Mellon University, Technical Report CMU-CS-92-160, July 1992.

[8] *Object Constraint Language Specificatio*n, Version 1.1, Object Management Group, September 1997.