2012 IEEE 14th International Symposium on High-Assurance Systems Engineering

Towards Adaptable Middleware to Support Service Delivery Validation in i-DSML Execution Engines

Karl A. Morris, Jinpeng Wei and Peter J. Clarke
School of Computing and Information Sciences
Florida International University
Miami, FL 33199, USA
Email: {kmorr007, weijp, clarkep}@cis.fiu.edu

Fábio M. Costa
Institute of Informatics
Federal University of Goiás
Goiânia-GO, Brazil
Email: fmc@inf.ufg.br

***Abstract*—A developing paradigm in the area of Software Engineering is that of Model Driven Development where models are used to express operations that are thereafter interpreted and executed through the use of an execution engine. The high level of abstraction within these models present inherent challenges in guaranteeing operation that respect policies and other constraints during execution. Additionally, the domain specificity necessarily present within these execution engines make them rigid and not suited for repurposing across different domains. We propose to address these issues through the use of a middleware architecture that is responsible for the service delivery aspect of the execution engine. Our architecture will provide a separation of domain specific and domain independent concerns, resulting in a set of domain specific artifacts which possess domain knowledge, and a generalized execution platform that inherits its operations from the domain artifacts. Our design facilitates the realization of user intent through the generation, validation and execution of adaptation models at runtime constrained by policies. We show the viability of this approach in the User-Centric Communication Middleware, a layer of the Communication Virtual Machine, which is responsible for enforcing communication requirements.**

***Keywords*-Models at Runtime; Adaptable Middleware; Functional Assurance; Domain Independence;**

I. Introduction

The growth of Model Driven Engineering and Model Driven Development has increased tremendously in recent years [1]. Its growth has ushered in new research on how to interpret and execute these models. Conventional approaches generally look at model transformation through translation into programming code. This code is then transformed into executable code by a compiler. A developing trend in this area is to remove the steps involved in model translation, and to instead execute the models directly. To achieve this, one requires a semantically rich environment capable of interpreting these models. The use of an Interpreted Domain-Specific Modeling Language (i-DSML) execution engine is one such environment [2]. These engines are able to interpret and execute a model within a given domain without the need for translation to executable code. Within the i-DSML execution engine is a middleware responsible for service delivery. This middleware must be context-aware, policy-driven, and ensure full service delivery in heterogeneous environments [3].

As the declarative nature of i-DSMLs may result in syntactically similar languages across domains, it is natural that their execution engines would also be similar as this can reduce re-engineering time and effort. It is therefore desirable to standardize the mechanism for interpreting and executing models within a family of execution engines. In order to achieve this, we must utilize an architecture that is capable of achieving the necessary run time adaptation for varying domains, while still providing functional guarantees. We must ensure that a generalized execution engine is able to realize all the required functionality of a specific domain with high levels of assurance once it is specialized for that domain.

To address this issue of functional assurance, we have designed an adaptable middleware architecture that, through the use of models at runtime, performs on-the-fly validation prior to adaptation. Our architecture is separated into two distinct facets: A set of artifacts that capture the domain specific concerns, and a domain independent platform which serves as the executing machine of the middleware. A middleware instantiated for a specific domain is able to dynamically generate structural adaptation models that adhere to all relevant system constraints while realizing user intent.

We put forward the following contributions:

- Model-driven adaptation based on operation classification: We introduce an approach to middleware adaptation which utilizes runtime generated models that capture the operations of the middleware as specified by the user. The components of these *intent models* are categorized by a domain specific ontology.
- On-the-fly model-driven assurance validation: We look at functional assurance within our model driven approach by ensuring policies are adhered to in any valid model. Our middleware design incorporates the validation of models for functionality assurance as a required step in the adaptation process.
- Domain Independence in an adaptable middleware: Our architecture encloses all domain specific information in a set of classifiers and an organized set of operational units called *procedures*. It then provides a platform capable of executing those procedures based on semantics inherent to the classifiers. As a result, our architecture is not limited to a specific domain.

1530-2059/12 $26.00 © 2012 IEEE
DOI 10.1109/HASE.2012.25

We begin by giving needed background information in Section II. In Section III we take a closer look at i-DSMLs and execution engines. In Section IV, we present an overview of our proposed architecture. Section V describes our Domain Specific Classifiers, which form the high level taxonomy of a middleware's domain. In Section VI we present the structural metamodel of the middleware, which defines the constructs used to build intent models. In Section VII we detail the domain independent platform, which is responsible for the generation, validation and execution of these models. Section VIII describes an implementation of the middleware in the context of the Communication Virtual Machine (CVM). We then close with related work and conclusions in sections IX and X respectively.

II. Background

A. Adaptive Middleware

Adaptive middleware, as with adaptive systems in general, allows for the monitoring and reconfiguring of its structure and/or behavior at runtime [4]. This is achieved through various introspection mechanisms such as reflection. The ability to adapt a system based on system context gives it the ability to change based on the availability of new information and new resources.

B. Models at runtime

Models at runtime provide us with a mechanism to leverage models and the abstraction they provide during the real time operation of a system [5]. Similar to models used in the software development process, a runtime model allows us to reason about a system's environment and its behavior by presenting us with relevant information, while abstracting away superfluous information. This process opens the door to semantic based reasoning on, and modification of, a system's architecture and operations.

C. Data Privacy

Ensuring data privacy during storage and transmission is an important requirement for today's connected systems. Key-based cryptography is one mechanism used to achieve this. Cryptography deals with the encryption of plain text files using external data called *keys*. There are two types of key-based cryptography: one that uses single or symmetric keys, where a single key is shared by the encrypting and decrypting systems. Such a system gives us the encryption function E $\rightarrow$ C=E(T, K), and decryption function D $\rightarrow$ T=D(C, K) where T is the plain text, C is the cipher text and K is the key.
The other type is referred to as public or asymmetric keys, where there is a distinction between the encipherment (K_e, C=E(T, K_e)) and decipherment (K_d, T=D(C, K_d)) keys. One well known symmetric key cryptographic scheme is titled Diffie-Hellman key exchange [6]. In this scheme, a single key is iteratively generated among all parties in a communication over an unsecured connection. This approach ensures that even though all transmissions between parties can be monitored, the final key will be private, and can thereafter be used to encrypt subsequent communication.

III. I-DSMLs and Execution Engines

An Interpreted Domain-Specific Modeling Language (i-DSML) provides a declarative mechanism to express a user's requirements in a given domain through the use of models. The model generated by an i-DSML is in turn interpreted by an execution engine which is responsible for realizing the model's intent. An i-DSML execution engine is separated into 4 layers each with its own set of concerns. An instance of an i-DSML execution engine and the modeling language it interprets is presented in III-A.

A. Communication Virtual Machine and CML

The Communication Virtual Machine (CVM) is an i-DSML execution engine which provides a runtime environment for the modeling and realization of user-centric communication services (Fig. 1). Models are specified using the Communication Modeling Language (CML), an i-DSML which contains abstract representations of concepts relevant in the communication domain [7]. The CVM platform is divided into four major levels of abstraction, each layer playing a role in realizing communication services. The layers of CVM are: (1) User Communication Interface (UCI) - provides an environment for users to specify their communication requirements using CML; (2) Synthesis Engine (SE) - synthesizes and negotiates CML models with other participants in a communication and generates control scripts; (3) User-centric Communication Middleware (UCM) - executes commands found in a control script to manage and coordinate the delivery of communication services to users; (4) Network Communication Broker (NCB) - provides a network-independent API to UCM and manages underlying frameworks to deliver communication services. Our work focuses on the functions of the User-centric Communication Middleware. We will augment the functionality of the current implementation by incorporating the artifacts of our middleware design.

B. User-centric Communication Middleware

The User-centric Communication Middleware (UCM) is the layer of the Communication Virtual Machine charged with

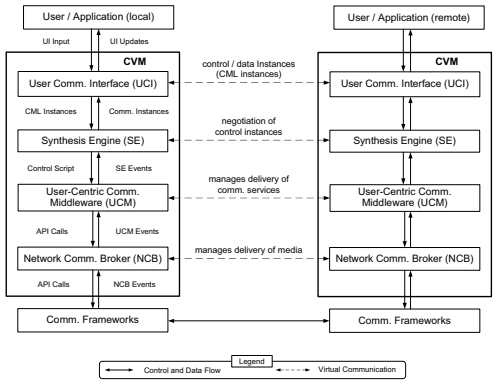

Fig. 1. Communication Virtual Machine

ensuring the delivery of services resulting from the synthesis of a communication model by the Synthesis Engine (SE) [8]. Upon completion of the model synthesis process, the SE packages and delivers a control script, which is an ordered set of commands, to the UCM. It is the job of the UCM to realize the intent of the user by performing the necessary operations described by the commands found in the control script while adhering to the non-functional requirements of the system. This may require the UCM to determine at runtime what the semantics of a particular command should be based on the current system context.

The Assurance Problem

As our modeling language is declarative, it lacks the ability to state non-functional constraints when stating requirements. Additionally, the dynamic determination of operational semantics based on system context presents us with the challenge of ensuring that one execution path matching a particular intent is operationally equivalent to another. These limitations present us with the assurance problem. Our middleware must ensure that the chosen adaptation is representative of the user's intent while at the same time ensuring adherence to all system constraints.

C. Policies for non-functional constraints

CML models give us a mechanism for stating intent. It contains the necessary artifacts for us to stipulate the requirements of a communication instance. However it lacks the facility to state how a particular communication instance should be realized, and what, if any, constraints must be adhered to. We therefore utilize policies to express these non-functional requirements.

D. Illustrative Scenario

To illustrate how the design of our middleware provides assurance of functionality during adaptation, we will take a look at a scenario in the Communication Virtual Machine.

Mr. J is a patient at a local hospital after becoming ill while on a business trip. The doctor at this hospital has contacted Mr. J's primary care physician in his home city and they are discussing his case. The physician needs to send Mr. J's records to the doctor, however since they are communicating via an unsecured Internet connection, the records must be encrypted before being sent to conform to the HIPAA privacy rule [9].

We will present an in depth overview of our design and then revisit our scenario to show how our architecture guarantees adherence to stated policies while ensuring delivery of service.

IV. Overview

Our middleware architecture achieves functional assurance through runtime validation of adaptation models. These intent models perform the necessary adaptation of our middleware based on current policies and environmental context. It generates, validates and executes intent models in response to system commands and events. Our approach, by design, ensures that any model which is selected for execution to carry out a user's intent fully conforms to all constraints the system has in place. It achieves this through the full classification of the middleware's operations, the generation of runtime models based on the classifiers, and finally validating and selecting a model for execution based on whether or not a model incorporates the features necessary to meet system constraints. This facility allows our middleware to only perform requested adaptation if it is able to do so within the current environment and with the available procedural components. If the middleware lacks the proper components to meet stated constraints, it throws an exception to the overlaying layer.

Fig. 2 gives an illustrative overview of the middleware's salient functions during the model generation and selection process.

1) Command Classification matches a command to a Domain Specific Classifier (DSC) in order to begin the model generation process. The relationship between DSCs and commands is discussed further in Section V.

2) Candidate Model Generation enumerates all possible candidate models which are able to realize the current intent based on the set of available procedures. This process is bound by the Maximum Partition Product which is discussed in Section VII-A.

3) Candidate Reduction and Selection first derives the subset of available models which conform to system policies. This process satisfies the assurance problem by ensuring that all resulting models match the user intent and fit all current system constraints imposed by policies. The resulting models are then passed to a cost function which selects the best model based on some predefined analysis.

4) Model Execution utilizes the selected intent model as an execution plan. This is discussed further in Section VII-C.

V. Domain Specific Classifiers

Domain Specific Classifiers (DSCs) form a top level taxonomy that categorizes the actions performed by the middleware (behavior) and the attributes that it is concerned with (state). Through this approach, DSCs catalog the domain specific concerns of the middleware and provide a framework on top of which all operational facilities will be built. They provide a common point of reasoning for commands, procedures and policies governing the middleware operation.

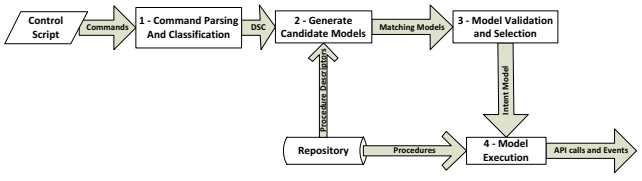

Fig. 2. Model Generation and Selection Process

ID	Name	Kind
1	CommunicationModel	attr
2	FileURI	attr
3	Send(FileURI)	oper
4	Receive(FileURI)	oper
6	plainTextFileURI	attr
7	encryptedFileURI	attr
8	Encrypt(plainTextFileURI, encryptedFileURI)	oper
9	Decrypt(encryptedFileURI, plainTextFileURI)	oper
10	homeNetwork	attr

TABLE I
A SET OF DSCS

Classifiers have a one-to-one relationship with the middleware's commands, and a one-to-many relationship with procedures. That is, a classifier may have more than one procedure that can carry out a specified command. All commands to which the middleware can respond must form a subset of the set of DSCs. An intent model for adaptation is therefore derived by building a dependency tree of procedures with the root being a procedure that matches the classifier of the command being executed. The set of DSCs are extensible. This allows the capabilities of the middleware to be expanded or reduced by manipulating classifiers and associated procedures.

DSCs that describe operations may, as a part of their definition, state parameters that are required for the completion of those operations. These parameters are themselves DSCs. For example, the DSC which describes sending a file, *Send*, would include a DSC for the name and path of the file to be sent, *FileURI*. Therefore, any procedure which conforms to the *Send* DSC must expect and handle the *FileURI* parameter. The makeup of a DSC is as follows:

- *Name* - String. Must be unique within a namespace.
- *Kind* - Operation or Attribute
- *Namespace* - A package-like system that helps provide classifier uniqueness within a domain
- *Parameters* An ordered set of DSCs

Table I presents a set of DSCs for the communication domain.

VI. STRUCTURAL METAMODEL

Here we detail the constructs of our intent models and their relationship (Fig. 3).

A. Procedures

We formally describe a procedure *P* as a 6-tuple (I, N, C, EU, EU_0, D) where

- *I* is the unique identifier
- *N* is the name of the procedure
- *C* is the classifier where C $\in$ {DSC}
- *EU* is the set of execution units
- EU_0 is the starting unit where $EU_0 \in$ EU
- *D* is the set of dependencies where D $\subset$ {DSC}

From an implementation perspective, we may view a procedure as an ordered collection of executable units, which may list a set of procedures on which it depends to perform its task. It is comprised of two parts.

- The descriptor, which provides the necessary meta-data for the procedure including name, the unique identifier, classification, starting component, and dependencies.
- The set of executable units which undertake the operations of the procedure including manipulating state information, making API calls, and calling for the execution of other units and procedures.

The set of dependencies of a given procedure is a proper subset of the set of DSCs. This is because a procedure of a given type cannot be dependent on that same type. To reduce potential complexity, we define a procedure as having only one classifier.

Dependencies: Our architecture describes procedure dependencies through DSCs (typed), as well as through their IDs (named). By utilizing DSCs, a procedure can simply declare what type of functionality must exist within the middleware for it to perform its function. In contrast, when a dependency is expressed via an ID, a specific procedure will be selected.

B. Execution Units

An execution unit is an atomically executable set of instructions that performs some aspect of the operations of its parent procedure. It may perform any number of allowed system operations; however it should be limited to making a single API call to any external interface. This constraint facilitates a high level of adaptability as the operations of the parent procedure are granulated in terms of their effect outside the middleware. An execution unit may be triggered as the initial step of a procedure, or in response to internal or external events, such as a timer or a message from a remote middleware instance respectively. It should be noted that we detail execution units here only for the completeness of the metamodel as they do not factor into the model generation and selection process.

C. Intent Models

An intent model is an acyclic directed graph where the nodes are procedures and the root of any subtree is dependent on its child procedures. The root of an intent model is a procedure whose DSC matches that of the currently executing command. The composition of an intent model is discussed further is Section VII-A. For safety and to reduce complexity, we define a well formed intent model as meeting the following criteria:

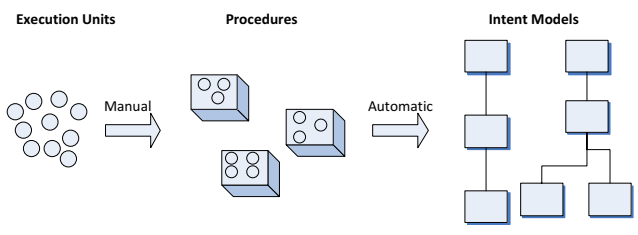

Fig. 3. Composition of an Intent Model

- Singly classified - This prevents a node from having multiple parents.
- Unique procedural dependence - A model must have only one dependence of a given type. As such, we speak of the set of dependencies to infer the nonexistence of duplicates.

VII. Domain Independent Platform

This section details the domain independent platform which is the part of the middleware that performs generation, selection and execution of intent models. Through the use of DSCs, procedures, and their execution units, we are able to abstract the domain specific considerations from our architecture. By removing this knowledge, what remains is a domain independent platform which serves as the execution mechanism for procedures in response to commands and events (Fig. 4). The platform provides a framework in which we specify a set of classifiers, and provide the procedures that perform the operations these classifiers describe.

A. Model Generation

Intent models are built using the concepts defined in Section VI, as specified in the generateModels() method of Algorithm 1. An intent model is a dependency tree which contains procedures as nodes, with the initial executing procedure as the root of that tree. A procedure with stated dependencies form the root of any subtree, and its children are procedures that match said dependencies. The leaves of the tree are procedures which have no stated dependencies. An intent model is built by matching the DSC of a command to the DSC of all available procedures. For a given procedure, we recursively walk its dependencies until no additional dependencies are required. If any dependency cannot be met, that model is eliminated from consideration.

A consequence of our method of defining and dynamically composing models based on types is model space explosion. In theory, an idealized set of procedures may produce an excessively large number of intent models that match a particular command. While this may not prove to be a limiting factor in practice, as a typical command may only relate to a small subset of the available procedures, it is still a motivating factor for addressing optimization strategies as we have done in Section VIII-C. The issue arises due to the Maximum Product Partition problem [10], where for a given set of procedures

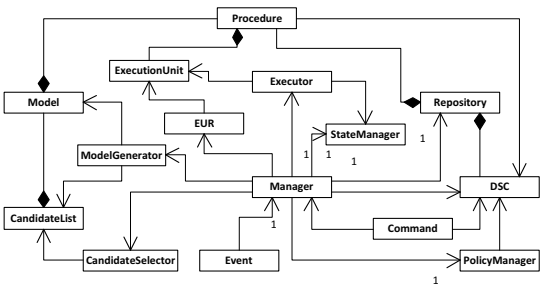

Fig. 4. Platform Metamodel

Algorithm 1 Model Generation and Selection

```
IN: Command
OUT: Intent Model
for all parameter in Command.Parameters do
    System.StateManager.set(parameter)
end for
models ← generateModels(Command.DSC)
validModels ← validateModels(models, Command.DSC)
for all model in validModels do
    cost ← costFunc(model)
    if cost < lowestCost then
        lowestCost ← cost
        bestModel ← model
    end if
end for
return bestModel
```

```
generateModels (DSC:initDSC)
procedures ← getMatchingProcedures(initDSC)
for all procedure in procedures do
    depDSCs = procedure.deps
    for all depDSC in depDSCs do
        subModels ← generateModels(depDSC)
    end for
    matchingModels.add(mergeModel(procedure, subModels))
end for
return matchingModels
```

```
validateModels (Model[]:models, DSC:modelDSC)
policies ← System.getMatchingPolicies(modelDSC)
for all model in models do
    for all policy in policies do
        if (checkModelForValidProcedure(model, policy)) then
            continue
        else
            break
        end if
    end for
    validModels.add(model)
end for
return validModels
```

P, there exists a partitioning based on DSCs that creates a maximal number of models.

B. Candidate Selection

The system can incorporate a multitude of techniques for selecting the most viable intent model through the implementation of a cost analysis mechanism. We define the function $f(M)$ such that

$$f(M) = \sum_{i=0}^{n-1} cost(M_i)$$

where n is the number of procedural nodes in the model and $cost()$ takes a procedure M_i and returns a deterministic cost associated with it. A specific mechanism for determining cost is not defined by our architecture. This ensures that we do not constrain domains which may have varied analysis requirements. We run this function on all candidate models and select the model with the minimal total cost.

C. Model Execution

We initialize execution of a model by loading the initial execution unit of the root procedure. Each unit has the responsibility of ensuring progression through a procedure by directly executing, or registering for execution, the next unit

in sequence. Execution units may also call on a dependent procedure if its capabilities are required. This call to a dependent procedure may be done indirectly (via a DSC), or directly (via the procedure's ID). If a call is made via a DSC, the middleware will execute the starting execution unit of the procedure previously identified during the model generation process.

D. State Information and Access Control

The middleware must manage state information which is used to coordinate inter and intra procedural operations. It accomplishes this through the use of the *State Manager* which manages key/value pairs which can be DSC attribute values as well as data generated by executing procedures.

E. EU Register

In order to facilitate distributed procedure execution (a procedure executing in response to continuous events from a remote instance of the middleware, e.g. during negotiation), the middleware must allow a procedure, or more specifically its execution units, to respond to events. Our architecture facilitates this by maintaining an register where execution units are registered to respond to specific events, including those generated in response to the action of remote middleware instances.

F. Repository

The repository of the platform houses the procedures, their execution units, and the DSCs which describe them.

VIII. Case Study

We now revisit the scenario introduced in Section III.

A. Mapping our Architecture

We begin by mapping our design to the current implementation of the Communication Virtual Machine, and specifically the User-Centric Middleware (UCM). Our execution units are the macros utilized in the UCM. These are minimal blocks of Java code which are executed by the system at runtime using reflection.

We formalize the concept of Domain Specific Classifiers within the UCM by subsuming the current list of commands that can be passed to the UCM through control scripts. Additionally, we formalize procedures by extending the macro definitions to associate them with specific procedures. Our architecture will allow the UCM to respond not only to commands provided through control scripts, such as establishing a connection or sending a file, but also to events received from the NCB. This is required to allow procedures to execute when they depend on responses from remote instances of the middleware.

As the current implementation of the CVM has no support for policies, we infer constraints appropriate for this domain to demonstrate the validation of models.

As stated in Section III, our scenario is set in the context of a larger communication model currently being executed where we have a multi-party connection established.

ID	Name	DSC	Init	Deps
1	SendBasic	Send	SBInit0	
2	SendSecure	Send	SSInit0	{Encrypt}
3	DHEncrypt	Encrypt	DHE0	{Aux:5}
4	PKIEncrypt	Encrypt	PKIE0	{Aux:6, Aux:7}
5	DHGenKey	Aux	DHG0	
6	PKICertAuth	Aux	PKICA0	
7	PKICertCheck	Aux	PKICC0	

TABLE II
Subset of UCM procedures

The middleware receives a command to send a patient file from the primary care physician to the local doctor. Due to HIPAA, the CVM has a policy in place (Fig. 5) which specifies that all media transfers taking place on an unsecured network must encrypt the media prior to transmission.

The middleware first generates candidate models based on the user's intent. It does so by mapping the system command to the appropriate classifier, selecting all matching procedures, and then recursively walking their dependencies to build the appropriate models. We present a subset of procedures available to the UCM in Table II.

After this process, we arrive at three models with the first containing the root node *SendBasic*, and the remaining two sharing the same root node, *SendSecure* (Fig. 6). All of these candidate models support the user intent of sending a file.

The middleware analyzes the current policies to find all contextually relevant model constraints. This ensures that our assurance guarantees are met as only models which conform to current system constraints are seen as viable candidates for execution. It finds a policy that applies an encryption constraint on all media transfers (*Send*). It then reduces the candidate list by querying the generated models to ensure that they contain a procedure which facilitates encryption (classified by the *Encrypt* DSC). As the names suggest, only two models include the ability to send a file securely resulting in the *SendBasic* model being discarded. The remaining models are then analyzed to ascertain their overall operational cost. The middleware incorporates a naive $cost()$ function which simply returns a constant value for each procedure, reducing a

if (!homeNetwork) {Send(FileURI) → Encrypt(FileURI)}

Fig. 5. Policy expressed using DSCs

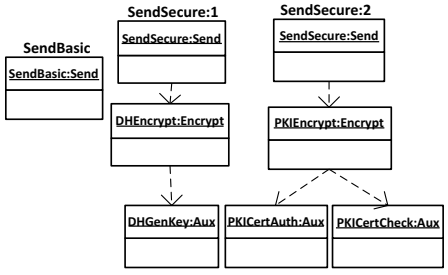

Fig. 6. Candidate models for scenario

model's overall cost to a count of the procedures it possesses. We therefore favor the model with fewer nodes and select *SendSecure:1* for execution.

Fig. 7 and 8 show the state machines depicting the execution of the procedures involved in the selected intent model. The *DHGenKey* procedure is executed in a distributed fashion on local and remote instances of the middleware.

During execution, *SendSecure* will first make a typed call to the middleware to execute the bound Encrypt procedure. It awaits completion of the called procedure by registering a macro to listen for the specific completion event. *DHEncrypt* is then called, which may, depending on whether or not a shared key is already established, make a named call to *DHGenKey* Upon execution, *DHGenKey* will attempt to create a shared key utilizing the Diffie-Hellman key exchange protocol [6]. This is then used by *DHEncrypt* to encrypt the file prior to it being sent by the *SendSecure* procedure. If at any point a procedure is unable to execute its assigned function, it throws an exception, which invalidates the operation, and the user is informed of the failure.

B. Prototype

We implemented a basic prototype to measure the overhead of the model generation process. This is the most normalizable process as it has no bearing on the relative complexity of procedures.

1) Platform: Our prototype was developed in Java 1.6 running on a 1.8 GHz Intel Core i5 MacBook Air with 8 GB of 1600 MHz DDR3 memory running Mac OS X Lion 10.7.4. Appropriate data structures were used to represent the artifacts of the architecture.

2) Setup and Results: We curated a test environment where we initialized our prototype with a set of 100 procedures to simulate a typical middleware implementation. Of the generated procedures, 10 were designed to realize the operation of the test command. Additionally, these 10 procedures were designed to realize the maximum partition sum bound (VII-A).

Our set of procedures resulted in the generation of 36 intent models ($3 \times 3 \times 2 \times 2$) which are detailed below. We validated our model against a DSC known to be present in all models, therefore ensuring that no models were eliminated from the candidate list, and passing all 36 models to the model selection phase. Finally our prototype selected a model for execution based on node count, on which our naive cost analysis implementation bases its comparisons. The evaluation results are presented in Table III:

Procedures available	100
Procedures used	10
Models generated from command	36
10,000 operation cycles	0.770 seconds

TABLE III
RESULTS OF MODEL GENERATION

As the results of our prototype evaluation show, we are able to perform the full generation, validation and selection process on commodity hardware in the order of micro seconds. Repeating the set of operations 10,000 times still comes in at under a second. We believe that these results demonstrate the appropriateness of our architecture for many domains, even those that may have a very high responsiveness requirement such as smart electrical grid management [11].

C. Performance and Optimization Considerations

Inherent to any adaptive system is the overhead resulting from the management of the transformation process. While the impact of this delay is unavoidable, we argue that our approach a) gives the implementing designer the flexibility to avoid excess overhead by carefully defining domain operations. In enumerating the operations in a given domain, a designer can limit the number and size of intent models by defining monolithic operations, and writing procedures which posses all the needed facets to realize the respective operations. This approach may, in the extreme case, result in single procedure models which are not dissimilar to traditional adaptive systems. We therefore highlight the flexibility of our architecture which gives the designer the ability to trade granularity for performance on a sliding scale by increasing or reducing the number of distinct operations present in the system. Additionally, it should be noted that even when accounting for the inherent overhead, an adaptable system such as this can introduce optimizations through the context aware selection of models which may result in better performance under certain circumstances.

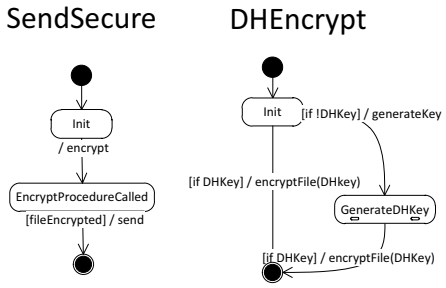

Fig. 7. State Machines for SendSecure and DHEncrypt Procedures

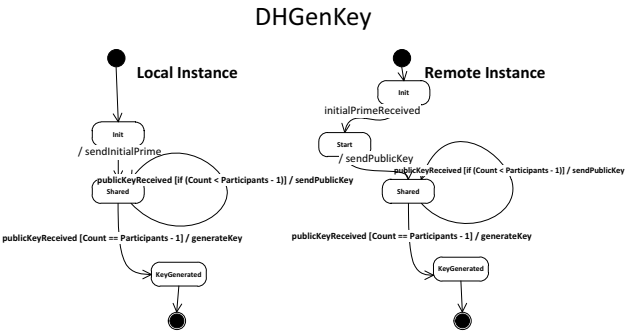

Fig. 8. State Machine for DHGenKey Procedure

Optimization

While our approach does not address optimizations in-depth, there are many strategies that could be utilized to mitigate performance degradation or slow response times. Examples of such strategies are:

- Pre-generation of models: The system generates and selects models to respond to all supported commands and keeps these models resident in memory. This can be undertaken at system start-up and the process reinitialized for specific commands/models whenever there is an applicable context change. Consequently however, this approach may leave a large number of models resident in memory that may never be utilized, but must still be managed by the system.
- Least Used Caching: The middleware may take a smarter approach to caching models by employing a strategy of Least Recently Used, Least Frequently Used, or some combination of both [12]. This addresses the issue of maintaining models which may not be needed by the middleware by replacing them over time with newer, more frequently used models.

IX. Related Work

In [13], the authors present an approach to dynamically bind middleware components for execution based on user intent and context. Their work is similar to our own in that it treats the platform as a base for execution in a programmatic way. Our work differs in the granularity of operations as their components are analogous to our procedures. As procedures are further broken down into executable units, we are able to achieve finer grain execution and adaptation. Our approach is also inherently distributed, as the platform's event registration service allows procedures to be distributed across multiple remote instances of the middleware.

Zachariadis et al. [14] demonstrate an adaptable mobile middleware that can augment its functionality based on functional component availability. While interesting, this work does not address having multiple components which meet the needs of the system, nor does it purport to be agile in its execution as components are monolithic in their composition and operation. Additionally, this approach provides no consideration of policies nor does it provide a single dialect with which to analyze component capabilities against system policies. Our architecture facilitates adaptation with a minimal resource footprint due to the use of execution units. Additionally, we provide a mechanism to analyze models when more than one are able to execute an operation. This can have a direct impact on the middleware's operational speed.

X. Conclusions

Our middleware architecture allows the separation of concerns though the bifurcation of domain specific and domain independent elements in the execution of interpreted domain-specific modeling languages. Domain specific concerns are captured through the use of procedures which perform operations relevant to the domain, and a set of domain specific classifiers which categorize them. Our domain independent platform is therefore able to execute procedures based on these classifiers in response to commands and system events. In doing so, we are able to provides assurance of functionality during adaptation though the classification and validation of required functionality in the selected adaptation model. Through our construction process, we preclude the possibility of generating inaccurate models with respect to functionality. We also show that the performance overhead is minimal for the contruction, validation and selection of models. We are able to generate models for structural adaptation that respect system policies and other constraints while ensuring service delivery per the user's intent.

Acknowledgment

This work was supported in part by a GAANN Fellowship from the US Department of Education under P200A090061. Fábio M. Costa would like to thank CAPES, Brazil, Proc. no. BEX 0759/11-2, for the support received during his sabbatical at FIU.

References

[1] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in *Future of Software Engineering, 2007. FOSE '07*, pp. 37 –54, may 2007.

[2] P. J. Clarke, Y. Wu, A. A. Allen, F. Hernandez, M. Allison, and R. France, *Towards Dynamic Semantics for Synthesizing Domain-Specific Models*, ch. 9. IGI Global, 2012.

[3] K. K. Khedo, "Requirements for next generation middleware implementations," in *Computing in the Global Information Technology, 2006. ICCGI '06. International Multi-Conference on*, p. 53, aug. 2006.

[4] F. Eliassen, A. Andersen, G. Blair, F. Costa, G. Coulson, V. Goebel, O. Hansen, T. Kristensen, T. Plagemann, H. Rafaelsen, K. Saikoski, and W. Yu, "Next generation middleware: requirements, architecture, and prototypes," in *Distributed Computing Systems, 1999. Proceedings. 7th IEEE Workshop on Future Trends of*, pp. 60 –65, 1999.

[5] N. Bencomo, "On the use of software models during software execution," in *Modeling in Software Engineering, 2009. MISE '09. ICSE Workshop on*, pp. 62 –67, may 2009.

[6] D. Coppersmith, "Cryptography," *IBM Journal of Research and Development*, vol. 31, pp. 244 –248, march 1987.

[7] Y. Wu, F. Hernandez, P. Clarke, and R. France, "A DSML for coordinating user-centric communication services," in *Computer Software and Applications Conference (COMPSAC), 2011 IEEE 35th Annual*, pp. 93 –102, july 2011.

[8] Y. Wu, A. Allan, Y. Wang, F. Hernandez, P. J. Clarke, and Y. Deng, "A user-centric communication middleware for CVM," *Software Engineering and Applications*, 2008.

[9] U. S. Congress, "Health insurance portability and accountability act." U.S. Department of Health & Human Services.

[10] T. Došlić, "Maximum product over partitions into distinct parts," *Journal of Integer Sequences*, 2005.

[11] M. Allison, A. A. Allen, Z. Yang, and P. J. Clarke, "A software engineering approach to user-driven control of the microgrid," *Software Engineering and Knowledge Engineering*, 2011.

[12] Z. sheng Li, D. wei Liu, and H. juan Bi, "CRFP: A novel adaptive replacement policy combined the lru and lfu policies," in *Computer and Information Technology Workshops, 2008. CIT Workshops 2008. IEEE 8th International Conference on*, pp. 72 –79, july 2008.

[13] U. Bellur and N. Narendra, "Towards a programming model and middleware architecture for self-configuring systems," in *Communication System Software and Middleware, 2006. Comsware 2006. First International Conference on*, pp. 1 –6, 0-0 2006.

[14] S. Zachariadis, C. Mascolo, and W. Emmerich, "The SATIN component system-a metamodel for engineering adaptable mobile systems," *Software Engineering, IEEE Transactions on*, vol. 32, pp. 910 –927, nov. 2006.