Model-Driven Domain-Specific Middleware

Fábio M. Costa*, Karl A. Morris†, Fabio Kon‡, Peter J. Clarke§
*Instituto de Informática, Universidade Federal de Goiás, Goiânia-GO, Brazil
Email: fmc@inf.ufg.br
†Department of Computer and Information Sciences, Temple University, Philadelphia-PA, USA
Email: karl.morris@temple.edu
‡Department of Computer Science, University of São Paulo, São Paulo-SP, Brazil
Email: kon@ime.usp.br
§School of Computing and Information Sciences, Florida International University, Miami-FL, USA
Email: clarkep@cis.fiu.edu

***Abstract*—Middleware was introduced to facilitate the development of sophisticated applications based on a uniform methodology and industry standards. However, early research and practice suggested that no one-size-fits-all approach was suitable for all application domains and scenarios. This gave rise to industry initiatives to standardize domain-specific middleware services and profiles, as well as research efforts on configurable, reflective, and adaptive middleware. The industry's approach led to easy deployment, although with a level of flexibility limited by the extent of existing profiles. The approach of the research community, on the other hand, enabled high flexibility, allowing any middleware configuration to be defined. Nevertheless, creating sound configurations using this approach is a challenging task, limiting the target audience to expert engineers. As a consequence, both initiatives do not scale with the current proliferation of specialized application domains. In this paper, we target this problem with an approach that leverages model-driven engineering for the construction of domain-specific middleware platforms. A set of high-level, yet expressive, building blocks is defined in the form of a metamodel, which is used to create models that specify the desired middleware configuration. We argue that this approach enables the rapid development of middleware platforms to match the proliferation of application domains, at the same time as it does not require per-application middleware construction or even highly skilled middleware engineers. We present the current state of our research and discuss research directions to fully realize the approach.**

***Index Terms*—distributed systems middleware; middleware engineering; adaptive middleware; model-driven engineering.**

I. Introduction

Although middleware facilitates the development of applications, the construction of effective, robust middleware itself is a very complex and challenging task. Engineering middleware for a specific application domain requires not only good skills in software design and implementation but also considerable experience in implementing applications for that particular domain. To take a middleware system built for one specific domain and adapt it for another domain is a very difficult task and frequently leads to non-optimized cumbersome solutions.

To address this challenge, in the past 20 years, middleware researchers have proposed more flexible and configurable ways to build such systems. Technologies such as customizable and component-based middleware [1], [2], reflective middleware [3], aspect-oriented middleware [4], [5], real-time middleware [6], and model-driven middleware [7] brought new ways of architecting middleware to cope with the variety of requirements, contexts and domains that such systems must deal with. These approaches offer considerable flexibility for the construction of middleware, but unfortunately their building blocks do not leverage the common middleware requirements found within particular application domains. Instead, they focus on building custom middleware configurations from the ground-up, using generic constructs that require significant expertise, and providing no clear path to reusing middleware across different applications within a domain. Furthermore, the proliferation of specialized application domains, such as in the Internet of Things and smart cities, requires new advances in middleware engineering to support applications on different devices with different needs with respect to QoS and other non-functional properties.

We build on the knowledge acquired by the middleware community in the past decades to propose a novel approach for middleware design and implementation: Model-Driven Domain-Specific Middleware (MD-DSM). We propose the use of Model-Driven Engineering concepts to facilitate the development of middleware systems that are specialized for particular application domains. Moreover, to facilitate application development by domain experts, or even by end-users, we propose that middleware should be capable of running model-based applications. *Thus, we overload the meaning of "model-driven" to convey the idea that middleware is built in terms of models, as well as the fact that it runs applications that are themselves defined in the form of models.* MD-DSM thus seeks to seamlessly unify three different perspectives of Software Engineering research for middleware: the use of models to build middleware, the ability to tailor middleware to specific domains, and the use of middleware as an execution engine for model-based applications. Middleware construction relies on a domain-independent middleware modeling language, which enables middleware engineers to define the structural and functional elements of the platform for a specific domain. Execution of application models, in turn, relies on a precise description of domain-specific knowledge, which is used to automatically generate the operational semantics required for the correct interpretation of application model constructs. Fur-

2017 IEEE 37th International Conference on Distributed Computing Systems

http://ieeexplore.ieee.org/abstract/document/7980137/

thermore, using models as a integral part of the development process supports easy customization, rapid development, and more comprehensive validation, starting at the design phase, potentially with a high percentage of code generation, and enabling round-trip engineering.

As a preliminary assessment of the approach, we review existing realizations of its foundation elements. We review a middleware architectural style based on application model execution, followed by its use in different application domains. We discuss an approach for building middleware using models that capture the common characteristics of a domain in the form of constructs for application development. This culminates with a description of MD-DSM as an approach to unify the three software engineering perspectives mentioned above.

In the remaining of this paper, Section II considers the motivation and challenges for MD-DSM, while Section III describes its associated methodology. Section IV reviews existing research that supports its proposal. Section V presents the foundation elements that underpin the approach, followed by Section VI, which describes the design of an MD-DSM platform. Section VII presents an evaluation of the foundational elements. Finally, Section VIII discusses related work, and Section IX concludes the paper, pointing out future research directions.

II. Motivation and Challenges

From an applications point of view, the motivation for MD-DSM can be found, for instance, in the realm of the Future Internet. The expected uses and the technologies that enable the Future Internet will result in a proliferation of new and specialized application domains. For instance, the general Healthcare domain can be specialized into more specific subdomains, each with its own merits, such as elderly care, patient monitoring, life support, epidemics management, and public policy design, to mention just a few. Similarly, the broad Smart Cities domain can be more conveniently handled in terms of more specialized application domains, among them, traffic control, public transportation, energy management, public safety, air quality, waste management, and emergency services. Specifically in the case of smart cities, Wenge et al. [8] suggest that any domain-related service can be turned into a smart-X system (e.g., smart transportation, smart traffic, smart grid). Moreover, they argue in favor of the integration of such smart systems as an essential aspect of a larger smart cities picture. In addition to that, and based on the principles of model-driven engineering, we argue that each such application domain needs its own modeling language to facilitate the creation and maintenance of new applications by utilizing constructs that are familiar to users of that particular domain.

Thus, there is a need to tackle two related challenges: (1) providing support for the development of middleware that is tailored for specific application domains, and (2) the need to develop and fully support domain-specific modeling languages for each such domain. While answering the latter challenge is a necessary step toward enabling the development of simple applications by end-users, the former is a key ingredient to efficiently support such applications.

In this paper, we target the first challenge above, namely, the provision of an approach and related infrastructure to facilitate the development of domain-specific middleware. The second challenge, which refers to the development of domain-specific application modeling languages for each domain of interest, has been addressed in our past research [9]–[12]. We assume the existence of such application modeling languages and provide the means to build model-driven middleware to support them by means of the dynamic execution of application models.

We note that more traditional methodologies, such as component-based development, can be used to produce specialized middleware for each domain [1], [2], [13]. However, these methodologies do not scale well with the growing number of application domains, as they require considerable expertise in middleware design and implementation. The use of model-driven techniques to build domain-specific middleware, as discussed above, raises the level of abstraction and makes the problem more tractable and amenable to non-expert developers, possibly domain experts, as opposed to being accessible only to middleware experts. In addition, the abstractions used to develop applications on top of the middleware are closer to the domain vocabulary, facilitating the job of domain experts. Not only this improves the speed of development but the formalization of such abstractions enables the use of automated tools to verify the consistency of the generated middleware, improving the reliability of its code.

Therefore, producing custom middleware configurations for each and every specialized application domain becomes a feasible target. Besides, middleware models may be realized on top of a component-based substrate, thus leveraging existing technology.

III. Overview of the approach

The overall approach is outlined in Figure 1, which shows the use of a common metamodel to create middleware models for different domains. These models are then used to create middleware instances that provide support for applications in each domain. Applications, in turn, are also built using a model-driven approach, noting the need for conformance between the application domain-specific modeling language (DSML) and the middleware model that supports it.

Figure 2 outlines the process of developing middleware and applications in MD-DSM. Initially, the middleware platform is generated from two input models: a model of its structural elements, and a model of the domain knowledge describing its operational semantics. Both are provided by someone taking up the role of middleware engineer (such as a domain expert).

Note that, in contrast to early approaches to customized middleware [14], [15], which add specialized components to a generic base platform, MD-DSM provides, in the form of a metamodel, the basic building blocks to construct entirely new middleware structures. Another difference is that MD-DSM explicitly focuses on the development of common middleware

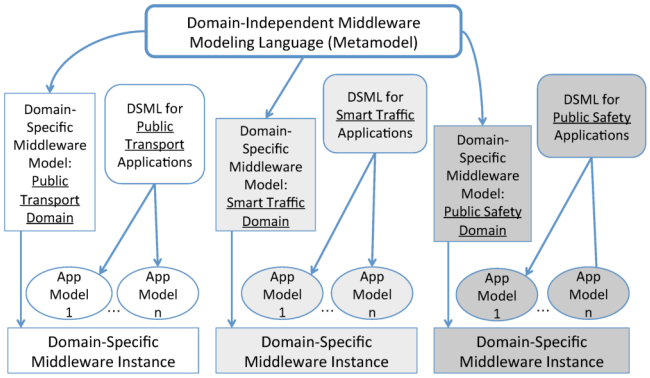

Fig. 1. Model-Driven Domain-Specific Middleware: Overall approach.

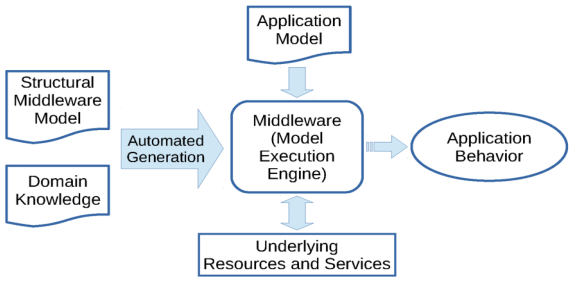

Fig. 2. Outline of the MD-DSM process.

for a variety of applications within a focused domain, instead of trying to customize middleware on a per-application basis. Once the middleware is instantiated, it can interpret application models provided by users, resulting in the orchestration of the underlying resources and services to produce the intended application behavior.

MD-DSM thus targets a specific architectural style for middleware, based on a model execution engine that runs application models created using an application-level DSML. We adopt a particular reference architecture for the model execution engine, based on layers of abstraction. This architectural choice facilitates the creation of a common middleware metamodel with well-defined execution semantics, at the same time as it does not compromise MD-DSM's ability to support the creation of middleware for different application domains.

The reference architecture is structured in four layers: (1) the **User Interface** layer provides a language environment for users to specify application models; (2) the **Synthesis** layer is responsible for transforming application models into sequences of commands; (3) the **Controller** layer is responsible for driving the execution of commands, taking into account the current context, applicable non-functional properties, and assurance of soundness and safety of the applications semantics; and (4) the **Broker** layer is responsible for interacting with the underlying resources and services for the actual execution of commands, considering systems issues such as heterogeneity and concurrency. Note that we leverage on the models@runtime approach [16], so that application models can be reflectively modified at runtime with immediate effect on how the underlying resources and services are handled.

IV. Experience in Specific Application Domains

We now review previous experience that motivates MD-DSM as a unifying approach for building domain-specific middleware. We describe four different platforms, which, despite being aimed at distinct application domains, share the same foundation: the ability to dynamically interpret domain-specific application models using a layered reference architecture. MD-DSM resulted from an effort to generalize and systematically apply the architectural principles of those platforms, contributing to streamline the construction of custom middleware for particular application domains. Note that, for historical reasons, the term 'middleware' is used in this section to refer to the specific layer that deals with the delivery of services and their associated non-functional properties, whereas in the generalized approach of MD-DSM, it refers to the entire platform that supports the development, execution, and runtime adaptation of model-driven applications.

A. Communication Domain

The *Communication Modeling Language* (CML) is a DSML for the domain of user-to-user communication [9], [10]. It enables domain experts to describe models for particular communication scenarios. Such models are fed into a model execution engine, called *Communication Virtual Machine* (CVM), which enacts the behavior intended by the user (as expressed in the model) by means of the orchestrated use of underlying communication services. It should be noted that the models used to specify the communication scenarios are based on the semantics of distributed applications and the CVM must therefore exhibit the behavior associated with such systems, including concurrency, scalability, fault tolerance and transparency, among others.

CML may be used to create two types of models: *schema* and *instance*, similar to the concepts of class and object in the OO paradigm. Schemas can be divided into *control* – that defines the configuration of the communication, and *data* – that defines the media and media structures that can be used in the communication defined by the control schema. The CVM consists of a four-layer architecture as shown in Figure 3. The purpose of each layer is as follows: (1) *user communication interface* (UCI) – provides the user with the ability to specify his/her communications needs; (2) *synthesis engine* (SE) – transforms CML models into control scripts and negotiates communication models with other parties in the communication; (3) *user-centric communication middleware* (UCM) – is responsible for the delivery of the communication services by interpreting the control scripts from the SE; and (4) *network communication broker* (NCB) – provides a network independent API to the UCM and hides the heterogeneity of the underlying communication services.

B. Smart Microgrids Domain

We developed a modeling language and execution engine for energy management in the smart microgrid domain [11],

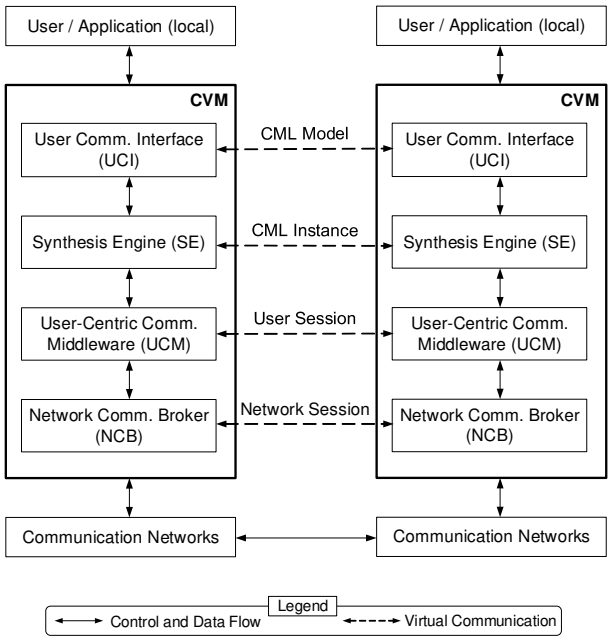

Fig. 3. Communication Virtual Machine (CVM).

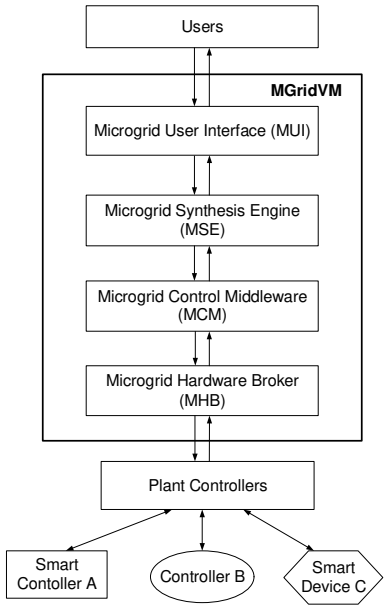

Fig. 4. Microgrid Virtual Machine (MGridVM).

based on the approach used for the communication domain. In this work, the modeling language was the *Microgrid Modeling Language* (MGridML) and the execution engine was the *Microgrid Virtual Machine* (MGridVM). The user expresses the configuration requirements of the microgrid, which maybe a home, using MGridML and the MGridVM interprets the model to realize the state of the system. Unlike the communication domain, the models for the microgrid domain exhibit the behavior associated with the semantics of a centralized application and include features such as a shared main processing unit, accessibility to all resources, and high resource utilization, among others.

MGridVM uses a four-layer architecture similar to the CVM as shown in Figure 4. The differences are mainly due to the nature of the domain. The layers of the MGridVM are: (1) *microgrid user interface* (MUI) – provides the user with the ability to create MGridML models specifying the requirements of the energy management in the microgrid; (2) *microgrid synthesis engine* (MSE) – transforms the MGridML models into control scripts and establishes causality with the plant controllers; (3) *microgrid control middleware* (MCM) – interprets control scripts, maps logical controllers to physical controllers, applies energy management algorithms, and enforces policies for various device configurations; and (4) *microgrid hardware broker* (MHB) – responsible for issuing atomic commands to the microgrid controllers (and devices) and for monitoring controller states.

C. Smart Spaces Domain

We developed another modeling language, called 2SML, and its respective execution engine, 2SVM, for the domain of smart space programming [12]. The language constructs represent the main kinds of elements that constitute smart spaces – users, smart objects, and ubiquitous applications – along with the relationships among them. The execution engine interprets a 2SML model and generates the necessary commands to configure the programmable entities of the smart space (smart objects and applications).

The execution engine has a four-layer architecture similar to CVM. The main difference is that the architecture was adapted to the characteristics of the execution environment: the instance of 2SVM that runs on the central device that controls the smart space only has the three top layers, while the instances that run on smart objects only have the two bottom layers. The rationale for this is that model synthesis only happens in the smart space controller, which dispatches the synthesized control scripts to the middleware layer on the smart objects. The broker layer, in turn, is only necessary in the smart objects, which is where 2SVM actually interacts with the controlled resources. Another important difference is that the generated control scripts are not immediately executed by the middleware layer. Instead, they are installed at the layer and their execution is triggered by asynchronous events, such as when smart objects enter or leave the environment.

D. Mobile Crowdsensing Domain

Finally, we also developed a modeling language and its corresponding execution engine for the domain of participatory sensing using smartphones. They are called CSML and CSVM, respectively, and allow the user to specify models that represent crowdsensing queries, which in turn are dynamically interpreted to drive the acquisition of sensing data (from participating devices) and the subsequent processing to produce the query results [17]. For long running queries, CSVM also allows on-the-fly changes to the user's model, which dynamically reflect on the execution of the query.

Similarly to 2SVM, CSVM adopts an adaptation of the four-layer architecture, according to the execution environment. A crowdsensing environment is composed of a logically centralized provider and a potentially large set of devices. The CSVM configuration that runs on mobile devices has all the four layers, whereas the configuration that runs on the provider only has the three bottom layers, since creation and modification of user models only happens in the mobile devices.

E. Discussion

A common characteristic of the above platforms is that they share the same architectural style and reference architecture. The architectural style is that of a model execution engine, which takes application models as input and produces application behavior based on the model-driven orchestrated use of underlying resources. For instance, in the case of CVM, the resources are the communication services, whereas in MGridVM they are the microgrid controllers and smart devices. As for the reference architecture, the platforms adopt a four-layer architecture with *user interface*, *synthesis engine*, *middleware* (henceforth called *controller*), and *broker* layers.

The reference architecture (and the architectural style it implements) has proven adequate for the design of middleware in the four very different application domains reviewed in this section. It is flexible enough to be adapted to the processes, concepts and execution environment that characterize each domain, and yet is expressive enough to fully enable the realization of applications in each case. For instance, while the CVM adopts a synchronous model of execution, based on the control of streaming and data communication among a handful of interacting parties, the other three platforms follow a mostly asynchronous mode of execution, in environments with potentially large numbers of objects that present important scalability requirements. This enables us to propose it as the architectural basis for the middleware engine that underlies the MD-DSM approach. In comparison with more traditional middleware, this reference architecture raises the level of abstraction of the middleware interface used by application developers: from the traditional API level to the level of user-defined models, which is precisely the interface provided by the Synthesis and UI layers.

V. Elements of the Approach

The experience gained with building these platforms enabled us to identify the common structures that must be present in model-driven middleware independently of any particular application domain. Based on this, we have identified two foundational principles to build domain-specific middleware using the layered reference architecture described above, together with their corresponding domain-independent building blocks and supporting mechanisms.

A. Model-based construction of middleware

As demonstrated in Section IV, the four-layer architecture for model execution engines can be applied to build middleware for multiple, diverse application domains. Based on this experience, we are currently developing a metamodel that captures the principles and constructs used to build such platforms and which are common across domains. A middleware model, which is created as an instance of this metamodel, defines the mechanisms and structures needed to interpret user-defined application models. In other words, it describes the *model of execution* (MoE) for an application-level DSML.

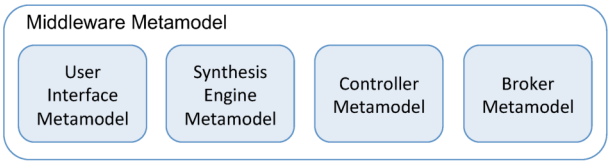

Fig. 5. High level structure of the metamodel.

The middleware metamodel is complemented by a generic, domain-independent, *runtime environment* responsible for loading and executing middleware models to realize their respective MoE [18]. The runtime environment is used to generate and execute the appropriate middleware components defined in the model. It does so with a component factory that generates each middleware component based on code templates that are parameterized with metadata from the middleware model. It also provides threads (and the underlying concurrency model) to run the middleware components.

The actual semantics of the application domain, in turn, is specified together with the middleware model, in terms of *actions* that convey the behavior of each of the constructs defined in the application-level DSML. Importantly, alternative actions can be defined for the same construct, and the choice of action to use in a particular execution of an application model element is based on policies and context variables defined in the middleware model.

This approach has been demonstrated in two application domains: communication [18] and microgrids [19]. The macro structure of the middleware metamodel is in accordance with the layered architecture described in Section IV. Each layer is defined by its own (sub-)metamodel, as shown in Figure 5.

We currently have fully functional metamodels for the Broker and Controller layers, as described next. The metamodel for the Synthesis layer is currently under development. It will be a generalization of the computational model for the synthesis process defined in [11], which is based on the use of labeled transition systems to encode the domain-specific semantics of model synthesis. The main components in the synthesis engine are: (1) *model comparator* - compares the new user-defined model and the current runtime model to produce a change list; (2) *change interpreter* - processes the change list to generate control scripts (using the current state of the labeled transition system) and handles events from the Controller layer; and (3) *dispatcher* - dispatches a new runtime model to the UI and updates the currently executing model. As for the UI layer, we leverage on existing tooling, such as those that compose the Eclipse Modeling Framework [20], which provide model editors and other model manipulation

tools generated from the metamodel of the given DSML.

A simplified view of the metamodel for the Broker layer is shown in Figure 6. It defines the constructs that are needed to build different configurations of the layer. Among such constructs are the main `Manager`, which is responsible for exposing the layer's interface and handling calls received from the upper layer and events received from the underlying resources. Calls and events are handled by selecting and dispatching appropriate actions, as discussed below. Besides the main manager, the metamodel enables the definition of specialized managers for other aspects of the layer's functionality: *state management* (to store and manipulate the layer's runtime model), *policy management* (to evaluate and execute policies that govern the layer's behavior), *autonomic management* (for self-configuration), and *resource management* (to interface with the underlying resources).

A middleware engineer thus models a configuration of the Broker layer by instantiating and appropriately initializing the elements of this metamodel, according to the requirements of the application domain. For instance, for the `Autonomic Manager`, different symptoms, change requests and change plans may be defined to specify the different situations in which autonomic behavior is triggered and how to handle each such occurrence. In addition, the middleware engineer also needs to specify the actions to be executed in response to calls and events received by the Broker layer. These are specified in the model as instances of `Action` and `Handler`, respectively, which define the mechanisms to select the appropriate action in each case.

The metamodel for the Controller layer is similar. The main differences are: (a) it does not have the resource and autonomic managers, two features that are unique to the Broker layer; and (b) it replaces the elements that represent signals, calls, and events with elements to represent the command scripts that are received from the Synthesis layer. In addition, the `Handler` now has the responsibility to iterate over the scripts, choosing the appropriate action to interpret each command. A more detailed design of the Controller layer is presented in Section VI, as part of the integrated MD-DSM approach.

B. Separating Domain Knowledge from Model of Execution

Based on the application-level DSMLs and their respective execution engines described in Section IV, as well as their common features, the next logical step was to separate the *domain-specific knowledge* (DSK) from the model of execution (MoE) in the execution engines. This resulted in a *generic MoE* (or simply GMoE) for each layer of the reference architecture. In the following, we describe the separation of the DSK from the MoE for each layer, paying special attention to the Controller layer. Key to the instantiation of an MD-DSM platform, as shown in Figure 2, is the integration of the middleware model (instantiated from the middleware metamodel) and the DSK.

As suggested in the previous section, the separation between DSK and MoE for the UI layer is facilitated by the existing tools used to create the modeling environment. The domain-specific knowledge is contained in the metamodel for the DSML and the tools (Eclipse Modeling Framework [20] and Graphical Modeling Framework [21]) provide the MoE needed to instantiate the UI.

The input to the Synthesis layer is a sequence of user-defined DSML models and the output is a set of control scripts sent to the Controller layer for processing. The semantics used to execute DSML models in the Synthesis layer involves comparing two models at runtime: the model that is currently running (an empty model if the system has just been started) and a new (updated) model submitted by the user. Based on the differences between the two models and on the current state of the system, the semantics is determined [11].

The MoE for the Synthesis layer includes the actions performed by the model comparator, change interpreter, and dispatcher components (see Section V-A). The domain-specific knowledge includes the metamodel for the DSML, labeled transition systems containing the behavior, and the metamodel for the control scripts. The labeled transition systems contain the behavior for the level of abstraction relevant to the synthesis process. A more detailed explanation of the separation of the domain-specific knowledge and the MoE for the Synthesis layer is presented in [11]

Unlike the UI and Synthesis layers of the execution engine, the DSK and MoE are more tightly coupled in the Broker layer. Allen et al. [22] provide a detailed description of the behavior of the network communication broker for the CVM. Although there is no current work describing how the DSK and MoE may be separated for the existing broker of the CVM, Section V-A outlines an early approach for instantiating the DSK and MoE for the components of this layer. This is one of the areas for future research in MD-DSM.

The main layer that addresses operational variability is the middleware control layer (Controller). Its main purpose is to execute the command scripts received from the Synthesis layer. It does so by isolating the commands contained in a script and dynamically generating, for each command, an executable model that conveys the operational semantics of the command in accordance with the current context and user-defined rules. To achieve this goal, the Controller layer focuses on classification as a pillar of its design. In this approach, domain operations are placed in categories that describe their goal. Additionally, we classify various data used by the layer using a similar mechanism, but with the purpose of being able to refer to these data as opposed to categorizing them [23].

By breaking down operational facets to their composite operations and data, and classifying these components based on their goal, we are able to demarcate the specific concerns of a domain. Having determined this classification, the remaining executing components are void of any domain-specific concerns and therefore formed the domain-independent portion of the architecture.

To properly capture domain-specific concerns (classification and operations) we define two artifacts, and related sub-artifacts:

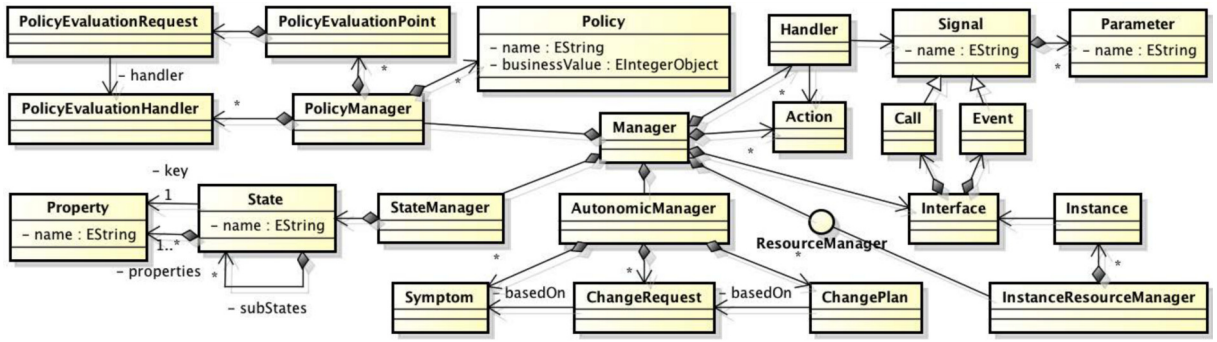

Fig. 6. Metamodel for the Broker layer.

- *Domain Specific Classifiers*, or DSCs, categorize operations and data based on the business rules of a domain. A set of classifiers is generated for a domain following an analysis of the various operations that take place in a given system, and the data with which these operations are concerned. Once generated, the DSCs serve as a mechanism to describe interfaces with implicit domain-specific constraints. This is tangentially related to the definition of interfaces found in object oriented design.
- *Procedures*, and their accompanying *execution units* (EUs), undertake the domain specific operations of the controller. They are classified by DSCs (to reduce complexity, current constraints limit a single procedure to be classified by a single DSC), allowing them to be considered as candidates to realize the abstract operation (i.e., the interface) that matches their classifying DSC.

With these domain-specific artifacts, the execution engine mentioned above is able to perform the operations of the system's middleware and can examine various ways of executing a particular command depending on the number of procedures whose classification matches a particular DSC.

The generation of an execution model operates on procedure metadata to determine the optimal configuration of a set of procedures to carry out a requested operation based on active policies. It determines valid configurations by examining the DSC-described dependencies of a procedure X, and matches them with other procedures that are classified by the DSCs on which X depends. This step is repeated recursively while ensuring that unwanted configurations such as cycles are avoided, until a procedure dependency tree is generated. This tree is referred to as an *Intent Model* (IM), see Figure 7, whose operation is classified by the classifying DSC of the root procedure. It is understood that once generated, an IM can perform the requested operation through its execution.

Once generated, an IM is executed by the Controller layer by examining the EUs of the various procedures and executing the statements found therein. A set of domain-independent operations are available to a running EU that cover areas such as memory management, event handling, message passing and remote calls. These generic operations form the Controller's model of execution, and dictate how a procedure is able to

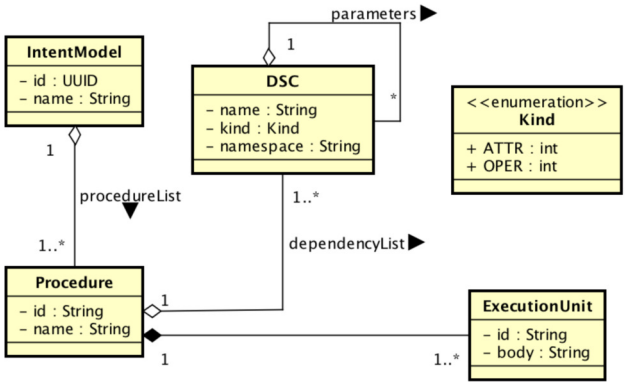

Fig. 7. Intent Model definition.

undertake its function. The concrete implementation of the model of execution is the Controller's execution engine.

The execution engine of the Controller is a stack machine that operates by executing the EUs of the procedure currently on top of the stack. In addition to executing its own code, a procedure X, through its EUs, can call procedures that were matched to its declared dependencies, which results in the called procedure being pushed onto the stack, or it can signal that it has completed its operation, resulting in the procedure being popped from the stack. An invocation of a procedure declared as a dependency is accomplished through a DSC-based call. When invoked, the execution engine retrieves the previously matched procedure from the IM and begins the execution of that procedure's EUs. The execution of an EU involves making calls to the underlying Broker layer through a set of exposed APIs. The APIs allow the Controller layer to execute the various domain-specific operations for which the EUs and containing procedures were developed, as well as any domain-independent operations that the Controller requires, such as remote communication and coordination.

C. Discussion

The metamodel-based approach described in Section V-A is flexible enough to model middleware for different application

domains. It even enables very different middleware configurations, such as those described in Section IV, where an entire layer may be suppressed if not needed. A limitation of the approach, however, is that it does not separate the internal structure and semantics of the middleware (defined by the managers and their features) from the semantics of the DSML that it runs (defined by the set of actions). They are both specified as part of the same middleware model. Although this is a valid approach, as demonstrated in [18] and [19], it results in unnecessary coupling. For instance, if new courses of action (to interpret the DSML constructs) need to be implemented, the entire middleware model may need to be reloaded, which may be a cause of inefficiency. Moreover, the lack of separation may become confusing as it merges the concerns related to middleware engineering with those of application-level DSML engineering (which include the actions that define the DSML semantics).

Ideally, the internal structure and semantics of the middleware and the semantics of the application domain should be specified separately. The approach described in Section V-B allows precisely such separation. Nevertheless, the existing realization of this approach assumes a fixed configuration for the middleware model of execution, thus not exploiting opportunities to make it more tailored (and efficient) for each application domain. MD-DSM, as discussed in the next section, is based on the observation that these two approaches complement each other in a synergistic way, combining their benefits and solving each other's limitations.

VI. Designing an MD-DSM platform

The conceptual elements that comprise the vision of Model-Driven Domain-Specific Middleware were laid out in Section III. In this section, we describe our first efforts towards combining the two foundational principles reviewed in Section V. This combination represents a significant step towards realizing the MD-DSM vision. The result is an approach that enables engineers to exploit the characteristics of each application domain to build middleware that is tailored to the domain. It does so by using a separation of concerns approach that allows the internal structure of the middleware to be described separately from the operational semantics of the application modeling language.

Currently, we have defined the architectural foundation to combine the two principles at the level of the Controller layer. The metamodel for this layer has been extended with constructs to represent the intent model generation feature described in Section V-B. A possible configuration for the Controller layer, which would have been defined as part of a middleware model, is shown in Figure 8. The figure only shows the elements that are related to the integration of the two principles (other elements, such as policy evaluators and resource managers, are omitted for brevity).

As shown in Figure 8, *calls* generated by the Synthesis layer (from the user's application model), together with *events* received from the underlying layer (Broker) or generated at the Controller layer itself, are received by Controller layer's

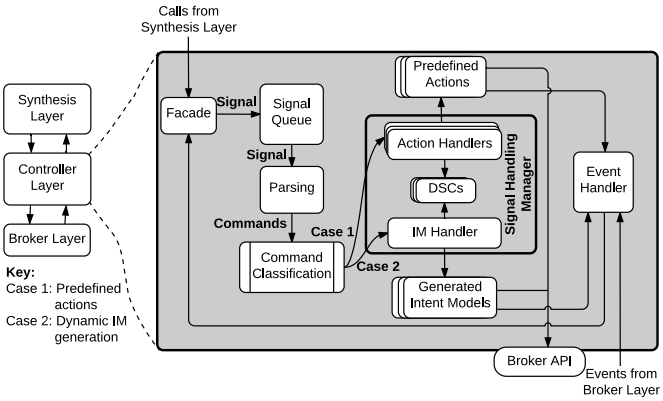

Fig. 8. Sample Model of Execution for the Controller layer.

facade. Both calls and events are treated in the same way and thus are indistinctly called *signals*. Received signals are queued for processing and then parsed to generate the commands that convey the intent of the user's model in a procedural way.

The metamodel enables coexistence of two distinct approaches to define the operational semantics of commands: *Case 1* – selection of predefined actions; and *Case 2* – dynamic generation of intent models (IMs). In both cases, the selection or generation process is guided by domain-specific classifiers (DSCs) defined as part of the layer's model. These DSCs are used either by *Action Handlers* to select an appropriate action to execute each command, or by an *Intent Model Handler* to instrument IM generation using the approach described in Section V-B. Importantly, the choice of which approach to use for each received command is determined by a command classification step that precedes actual command execution. Command classification takes into account domain policies and context information to choose between cases 1 and 2 for each command. Irrespectively of choosing Case 1 or Case 2, the resulting behavior is expressed either in the form of calls to the Broker layer or events that represent exceptional conditions to be processed by the Controller layer itself (via *Event Handler*).

Finally, note that Figure 8 is only an example, as other configurations are possible. For instance, we may define a Controller layer that relies solely on predefined action handlers for domains where efficiency is more important than flexibility. Conversely, for domains with highly dynamic behavior, flexibility may be more important, and we may have a Controller that relies solely (or mostly) on dynamic IM generation. Another rationale for balancing the choice between cases 1 and 2 is related to the amount of memory available to run the middleware. In cases where memory footprint needs to be reduced, dynamic IM generation avoids having to store a large number of predefined actions for each available command.

VII. Preliminary Evidence

In this section we present existing results on the implementation and evaluation of the two foundational elements for the

MD-DSM approach. These results enable us to demonstrate the effectiveness and the performance of these two elements when used separately. They also serve as preliminary evidence for the effectiveness of their combined use.

A. Model-based construction of middleware

The metamodel-based approach described in Section V-A has been successfully employed to construct middleware for the domains of communication and smart microgrids. In both cases, we were able to validate the behavioral equivalence (in terms of the sequence of commands that were generated for the underlying resources as a result of model interpretation) of the model-based implementations of the middleware and their original, handcrafted, counterparts.

An initial performance evaluation was based on a version of CVM's Broker layer built using the metamodel [18]. The intent was to compare the performance of the model-based version with that of the original layer of CVM presented in [22], [24]. A set of eight scenarios for multimedia communication, including session establishment, reconfiguration and recovery from failures, were implemented using both versions of the Broker layer. In terms of raw performance, the model-based version spent, on average, 17% more time to execute the scenarios than the original version. This overhead is a direct consequence of the extra flexibility allowed by the model-based approach. However, we note that in this experiment we decided to model an exact copy of the original Broker layer. The flexibility of the model-based approach would enable us to model leaner configurations for each of the layers, featuring only the strictly required components, thus contributing to compensate for the extra overhead.

Please note that the evaluation did not consider the time required to load the middleware model into the runtime environment and getting it up and running, which is a time-consuming operation. Nevertheless, this operation is only carried out at initialization time, thus not affecting the performance of the middleware when it runs application models.

B. Separating Domain Knowledge from Model of Execution

In order to test the middleware's approach to separate concerns and address variability in operations, as well as the effectiveness of its execution model, we performed a domain analysis, and subsequently created the specific artifacts for the same two domains: communication [10], [23], and smart microgrids [11], [23]. A summary of the results, focusing on the Controller layer of the execution engine for these domains, is presented below.

The evaluation involved the design of DSCs and procedures for both domains, expressing typical scenarios and measuring the ability and efficiency of the layer to execute them. To test the Controller layer's ability to separate concerns, we focused on its execution engine (the domain-independent aspect) to operate with DSCs and procedures from both domains without modification. In order to test variability, we populated the Controller's repository with multiple procedures that matched specific DSCs and then measured its ability to choose one execution path instead of another based on environmental context. In both scenarios, the Controller's repository was populated with metadata of 100 curated procedures aimed at achieving optimum dependency matching. With this test, the Controller layer was able to complete a full generation cycle (IM generation, validation, and selection) in under 120 ms, with the average cycle time quickly approaching 1 ms as we approached 100000 cycles (equivalent to 100000 sequential requests to the Controller) [23]. It was also shown that while the response time of our Controller layer architecture was measurably slower than a previous non-adaptive Controller undertaking the same task, scenarios where adaptability was beneficial to the task at hand would result in as much as an order of magnitude improvement in response time for our adaptive Controller layer (approx. 800 ms for our architecture, compared to approx. 4000 ms for the older non-adaptable architecture). Additionally, due to the separation of domain-specific concerns, we were able to achieve a reduction in lines of code (from 1402 to 1176) resulting in smaller compiled bytecode and execution footprint.

We were able to demonstrate the applicability of our approach with successful tests in both domains. Additionally, we tested the time requirements of the architecture compared to non-adaptable systems and found our results to be promising when considering the additional capabilities achieved [23]. While this does not allow us to declare the general applicability of our approach in any domain for which a DSVM was deemed appropriate, the results are encouraging and motivate further research.

C. Discussion

The two approaches used as the basis for MD-DSM – model-based construction of middleware, and separation of domain knowledge from the model of execution – are fairly orthogonal to each other. The former allows the construction of domain-specific models of execution (MoE), while the latter enables the incorporation of domain knowledge in the generated MoE. In fact, the integration and subsequent operation of the domain knowledge in the MoE is significantly transparent to the actual configuration of MoE in use. Thus, it is reasonable to argue that the separate evaluation of the two approaches provide a good indication for their performance as part of the unified approach. As future work, we aim to confirm this argument by repeating the above experiments using the integrated prototype that we are currently developing.

VIII. Related work

The MD-DSM approach crosscuts research in a number of related areas, from model-driven engineering to configurable and adaptive middleware, including combinations of both.

In particular, the Domain-Specific Development Infrastructures (DSDI) approach [25] closely relates to MD-DSM. It entails the use of a common metamodel to establish the semantics of a set of related domain-specific modeling languages (DSML). A model interpreter is generated from the integration of the metamodels of these modeling languages.

This interpreter, in turn, is able to generate code from application architecture models targeting a particular middleware platform. Although the DSDI approach in principle supports domain-specific middleware (the middleware is represented as one of the integrated metamodels), it crucially differs from MD-DSM in which it assumes that the middleware platform for a particular domain already exists. MD-DSM, in contrast, provides principled support for the creation of domain-specific middleware. Nevertheless, the common metamodel of DSDI, which provides the semantic anchoring for the DSMLs, offers a useful perspective from which to view the relationship between the common middleware metamodel of MD-DSM and the application metamodels. In fact, the DSDI approach to metamodel integration can be useful to automatically enforce conformance between the middleware model and the respective application modeling language in MD-DSM.

Model-Driven Middleware [7] (MDM) focuses on the generation of middleware for distinct applications with specific QoS requirements. MDM has been successfully used to support configuration and deployment of QoS-enabled publish-subscribe middleware for large component-based systems [26] and, more recently, to optimize middleware for the different variants of a software product-line [27]. The approach, however, does not consider the wider problem of building common middleware for specific application domains, as in MD-DSM.

A more explicit approach for the use of models and metamodels to build middleware is proposed in [28], which describes Genie, a development tool that combines component frameworks, reflection, and software families for the development of dynamically configurable middleware platforms. Genie proposes the use of different DSMLs (and thus different metamodels) to model middleware for different domains (e.g., one DSML for the domain of pub-sub middleware, and another for the domain of grid middleware). In contrast, MD-DSM employs a single, domain-independent metamodel to express the structure of middleware platforms, while the domain semantics is captured using a complementary set of constructs, as described in Section V. We argue that this unifies the experience of creating middleware platforms for different domains, without the need to create and manage a variety of middleware modeling languages. More recently, Bencomo et al. [29] have proposed the use of models@runtime to dynamically generate middleware connectors for interoperability among heterogeneous networked systems. Interestingly, the runtime models used to synthesize connectors are themselves created at runtime from models that specify the networked systems that need to interoperate. This provides an interesting perspective for MD-DSM, as their approach could inspire a solution for the interoperability problem across different domain specific middleware platforms.

IX. Conclusion

Configurability and dynamic adaptability are now common middleware features, both in research prototypes and, to a certain extent, in industry products. As a result, it has become possible to tailor middleware to particular domains and usage scenarios. However, the proliferation of application domains increases the demand for custom middleware, with features that closely match the requirements of a particular domain. Under these circumstances, current middleware engineering techniques are just too time-consuming, as they require significant expertise in middleware development. Our approach addresses this problem by raising the level of middleware development from code-level programming to high-level model specification, with the immediate effect of facilitating the creation of custom middleware for emerging domains. Furthermore, the approach extends the benefits of model-driven development to the application level, as the resulting middleware supports the execution of applications that are themselves defined in the form of models using a domain-specific modeling language. We argue that this high-level methodology for application development, coupled with the domain specificity offered by MD-DSM, will contribute to make the task of creating new, purpose-built apps more accessible to novice developers and even to power end-users.

Our research to date has demonstrated the feasibility of MD-DSM in terms of its constituent elements, namely the model-driven construction of middleware and the description and automated generation of the operational semantics needed for the middleware to execute application models. These elements were validated and evaluated in different application domains. The results, both in terms of performance and functionality, not only indicate the feasibility of the combined approach, but also provide good perspectives for our ongoing work to fully realize the MD-DSM vision.

The main research challenges to be addressed include the need for an approach to systematically extract the middleware requirements of an application domain and to provide assurance about the appropriate matching between such requirements and the structure and functionality described in the respective domain-specific middleware model. Related to that, an approach is also needed to systematically ensure that the generated MD-DSM adequately supports the application-level DSML. Supporting more elaborate application-level DSMLs, as in aspect-oriented modeling [30], is another important challenge, and an MD-DSM platform should be capable of simultaneously executing (through a weaving step) multiple related models that describe the different concerns of an application. Finally, performance issues need to be addressed with respect to the requirements and characteristics of each application domain, in order to tune and optimize an MD-DSM instance for its respective domain.

Acknowledgment

The authors would like to thank FAPEG, the research support foundation of the state of Goiás, Brazil, for partly funding this work. This research is part of the INCT of the Future Internet for Smart Cities funded by CNPq, proc. 465446/2014-0, CAPES proc. 88887.136422/2017-00, and FAPESP, proc. 2014/50937-1. The authors would also like to thank Weider Barbosa, from UFG, for his work on an early prototype.

REFERENCES

[1] M. Roman, D. Mickunas, F. Kon, and R. H. Campbell, "LegORB and ubiquitous CORBA," in *Reflective Middleware Workshop*. Palisades, NY: ACM/USENIX, 2000, pp. 1–2.

[2] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan, "A generic component model for building systems software," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 1, p. 1, 2008.

[3] F. Kon, F. Costa, G. Blair, and R. H. Campbell, "The case for reflective middleware," *Communications of the ACM*, vol. 45, no. 6, pp. 33–38, Jun. 2002.

[4] E. Truyen, N. Janssens, F. Sanen, and W. Joosen, "Support for distributed adaptations in aspect-oriented middleware," in *Proceedings of the 7th International Conference on Aspect-oriented Software Development*, ser. AOSD '08. New York, NY: ACM, 2008, pp. 120–131.

[5] C. Zhang and H.-A. Jacobsen, "Resolving feature convolution in middleware systems," in *Proceedings of the 19th ACM Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*. New York, NY: ACM, 2004, pp. 188–205.

[6] H. Pérez and J. J. Gutiérrez, "A survey on standards for real-time distribution middleware," *ACM Comput. Surv.*, vol. 46, no. 4, pp. 49:1–49:39, 2014.

[7] A. Gokhale, K. Balasubramanian, A. S. Krishna, J. Balasubramanian, G. Edwards, G. Deng, E. Turkay, J. Parsons, and D. C. Schmidt, "Model driven middleware: A new paradigm for developing distributed real-time and embedded systems," *Science of Computer Programming*, vol. 73, no. 1, pp. 39 – 58, 2008.

[8] R. Wenge, X. Zhang, C. Dave, L. Chao, and S. Hao, "Smart city architecture: A technology guide for implementation and design challenges," *Communications, China*, vol. 11, no. 3, pp. 56–69, March 2014.

[9] Y. Deng, S. M. Sadjadi, P. J. Clarke, V. Hristidis, R. Rangaswami, and Y. Wang, "CVM – A communication virtual machine," *Journal of Systems and Software*, vol. 81, no. 10, pp. 1640–1662, 2008.

[10] Y. Wu, A. A. Allen, F. Hernandez, R. France, and P. J. Clarke, "A domain-specific modeling approach to realizing user-centric communication," *Software: Practice & Experience*, vol. 42, no. 3, pp. 357–390, 2012.

[11] M. Allison, K. A. Morris, F. M. Costa, and P. J. Clarke, "Synthesizing interpreted domain-specific models to manage smart microgrids," *Journal of Systems and Software*, vol. 96, pp. 172–193, 2014.

[12] L. A. Freitas, F. M. Costa, R. C. A. Rocha, and A. Allen, "An architecture for a smart spaces virtual machine," in *Proceedings of the 9th Workshop on Middleware for Next Generation Internet Computing*, 2014, pp. 7:1–7:6.

[13] P. Costa, G. Coulson, C. Mascolo, L. Mottola, G. P. Picco, and S. Zachariadis, "Reconfigurable component-based middleware for networked embedded systems," *International Journal of Wireless Information Networks*, vol. 14, no. 2, pp. 149–162, 2007.

[14] V. Issarny, C. Bidan, and T. Saridakis, "Achieving middleware customization in a configuration-based development environment: experience with the aster prototype," in *4th Int. Conf. on Configurable Distributed Systems*, 1998, pp. 207–214.

[15] M. Astley, D. C. Sturman, and G. A. Agha, "Customizable middleware for modular distributed software," *Commun. ACM*, vol. 44, no. 5, pp. 99–107, May 2001.

[16] G. Blair, N. Bencomo, and R. France, "Models@run.time," *Computer*, vol. 42, no. 10, pp. 22–27, Oct 2009.

[17] P. C. F. Melo, R. C. A. da Rocha, and F. M. Costa, "Enabling dynamic crowdsensing through models@runtime," *Journal of Applied Computing Research*, vol. 5, no. 1, pp. 17–31, 2016.

[18] G. C. M. Sousa, F. M. Costa, P. J. Clarke, and A. A. Allen, "Model-driven development of DSML execution engines," in *Proceedings of the 7th Workshop on Models@Run.time*, ser. MRT '12. New York, NY, USA: ACM, 2012, pp. 10–15.

[19] A. S. Jr., F. M. Costa, and P. Clarke, "A model-driven approach to develop and manage cyber-physical systems," in *Proceedings of the 8th Workshop on Models@Run.Time*. Miami, Florida: CEUR Workshop Proceedings Series, 2013, pp. 1–11.

[20] The Eclipse Foundation, "Eclipse modeling framework," May 2016, http://www.eclipse.org/modeling/emf/.

[21] ——, "Graphical modeling framework," May 2016, http://www.eclipse.org/modeling/gmp/.

[22] A. A. Allen, F. M. Costa, and P. J. Clarke, "A user-centric approach to dynamic adaptation of reusable communication services," *Personal and Ubiquitous Computing*, vol. 20, no. 2, pp. 209–227, 2016.

[23] K. A. Morris, M. Allison, F. M. Costa, J. Wei, and P. J. Clarke, "An adaptive middleware design to support the dynamic interpretation of domain-specific models," *Information & Software Technology*, vol. 62, pp. 21–41, 2015.

[24] A. A. Allen, "Abstractions to support dynamic adaptation of communication frameworks for user-centric communication," Ph.D. dissertation, Florida International University, Miami, FL, 2011.

[25] G. Edwards and N. Medvidovic, "A methodology and framework for creating domain-specific development infrastructures," in *Proc. 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, Sept 2008, pp. 168–177.

[26] G. Edwards, G. Deng, D. C. Schmidt, A. Gokhale, and B. Natarajan, "Model-driven configuration and deployment of component middleware publish/subscribe services," in *Proc. 3rd Int. Conf. on Generative Programming and Component Engineering*, 2004, pp. 337–360.

[27] A. S. Krishna, A. Gokhale, D. C. Schmidt, V. P. Ranganath, J. Hatcliff, and D. C. Schmidt, "Model-driven middleware specialization techniques for software product-line architectures in distributed real-time and embedded systems," in *Proceedings of the MODELS 2005 Workshop on MDD for Software Product-lines*. ACM/IEEE, 2005, pp. 1–8.

[28] N. Bencomo, G. Blair, and P. Grace, "Models, reflective mechanisms and family-based systems to support dynamic configuration," in *Proc. 1st Workshop on MOdel Driven Development for Middleware*, 2006.

[29] N. Bencomo, A. Bennaceur, P. Grace, G. Blair, and V. Issarny, "The role of models@run.time in supporting on-the-fly interoperability," *Computing*, vol. 95, no. 3, pp. 167–190, 2013.

[30] R. France, I. Ray, G. Georg, and S. Ghosh, "Aspect-oriented approach to early design modelling," *IEE Proceedings-Software*, vol. 151, no. 4, pp. 173–185, 2004.