A Flexible Approach for Adding Middleware Completions to Software Performance Models

Adnan Faisal
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada
faisal@sce.carleton.ca

I. PROBLEM

Models used for software development represent mostly the functional properties of the software, without modeling the platforms on which the software will run. However, from a performance modeling perspective, the information about the underlying platform infrastructure (e.g., middleware, operating system, hardware, networks) on which the software will be deployed is essential for the analysis of software performance (e.g., System response time, Throughput). This issue was identified by Woodside et al. [1] and the authors called these added performance model elements *performance completions*. A key completion for a distributed application is its middleware [2, 3].

Many previous studies have shown that platforms play a key role in software performance. Just to mention a few, in Juric et al. [5] and Gómez-Martínez et al. [6], the response time of message transmission is compared for various middleware (e.g., RMI, RMI-SSL, SOAP), and it is shown that the response times can vary as much as 14%, depending on the middleware used. In Faisal et al. [4] the impact of network latency on the performance of software deployed across multiple clouds is investigated, showing that deploying an application to a more powerful cloud may cause performance degradation rather than gain if the communication overhead is too high. Therefore, it is very important for the performance modelers to have a systematic method for building performance models for the platform infrastructure, and especially for the various features presented within a middleware. This gives them the opportunity to rapidly build performance models for changing circumstances and to reuse existing middleware models.

The objective of the proposed research is "**to develop a systematic, flexible framework that enables model completions for a wide range of middleware for software performance models"**. The goal of such modeling is to predict how an application software would do in terms of performance if deployed to different middleware platforms with various features. The proposed framework is flexible because it offers a mechanism to model a wide range of middleware and their features. Also, the size of the produced models are flexible as the framework allows the modeler to choose desired level of modeling details.

II. RELATED WORK

Four lines of related research on performance completions have been found. Wu et al. [8] propose a subsystem modeling approach in the context of LQN, which has variability only in parameters, but not in structure. Verdickt [9] in line two took the approach of manually modeling CORBA middleware in UML, weaving it to application software model, and then manually transforming the middleware-aware software model from UML to LQN. This approach has automation only in one step (i.e., weaving middleware model to software model), and even this is done only for CORBA (i.e., not middleware in general). The third line of research is a component-based modeling approach in the context of Palladio Component Model (PCM) [10]. Happe et. al. [11] proposed pattern-based performance completions for Message Oriented Middleware (MOM); Becker [12] proposed coupled transformations to compose completions into both generated code and a performance model; Strittmatter et al. [13] used feature models and completions to build a particular communication infrastructure from an abstract connector model; Kapova [14] introduced variability in the model transformation process (rather than just in the model instances). The fourth line of research is based on Aspect Oriented Modeling (AOM). Alhaj et al. [15] proposed an approach to transform platform independent-model to platform-dependent model in the context of a model transformation chain that generates queueing-based performance models from UML design models of service-oriented applications. The Reusable Aspect Model (RAM) proposed by Keinzle et al. [16] can be used to model middleware variability of functional requirements, whereas this research aims to model the variability of non-functional requirements for performance prediction.

The present work is different in two important ways from its predecessors. First, it considers pure performance model completions, not completions as components as in the first and third research lines. A model completion is both more general and simpler than a component or connector completion. The second difference is that (unlike the second, third and fourth research lines) the completion is defined and composed at the performance model level, not in the software specification. This is often simpler because the performance model is at a more abstract level than a design model (e.g., PCM or UML).

These differences give two advantages. First, the resulting performance model is smaller and simpler. Second, design details may be confidential or simply unavailable, and for such cases a performance model may be constructed as a "black box" for the middleware, then composed using the methods proposed here.

III. PROPOSED SOLUTION

The various models that take part in the proposed framework are shown in Figure 1. The Base Middleware Model (BMM) is the simplest possible middleware, which is in reality a single call. The commonality and variability in different middleware features are identified and modeled as feature models. These feature models are realized in Feature Realization Models (FRM). Which features to be composed to the BMM are described in a Feature Composition Descriptor (FCD) and given as input to the "Compose Feature" module of the Middleware Composition Engine (MCE) to produce feature-aware Specialized Middleware Model (SMM). The SMM is calibrated to obtain its service time under various workload.

The next step is to compose the SMM to an application software model that does not contain a middleware model. This software model is called Base Software Model (BSM). The "Compose Middleware" module of the MCE takes BSM, SMM and MCD (Middleware Composition Descriptor) as input to produce middleware-aware Specialized Software Model (SSM).

The performance models can come from any model-creation process, but in this research they are assumed to come from a Domain Specific Language (DSL) targeted for performance modeling called Layered Queuing Network (LQN) [7]. The LQN models can be executed (both analytically and by simulation) to obtain different performance metrics such as response time, throughput, utilization etc.

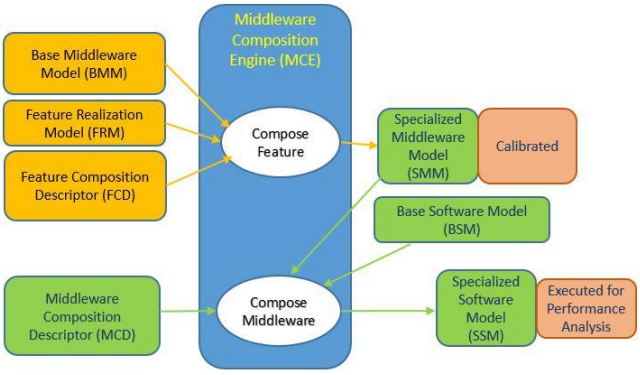

Fig. 1. Overview of the Proposed Framework

The LQN models are capable of representing the software components and their deployment, in order to capture inter-component communications, and to analyze resource interactions between layers of the application. Figure 2 shows an LQN model for a distributed weather application (*WeatherApp*). This is a two layer application where weather stations located in various parts of the city send data to a central weather center for data collection and analysis. For the purpose of brevity, only two weather stations are shown in this model.

Each concurrent entity (called a *task*) is represented by two or more attached rectangles. The rightmost rectangle shows task name and a parameter for its thread-pool multiplicity (e.g., {10}, default value = 1). It has attached rectangles to the left that represent its operations, called *entries* and labeled with the host (CPU) demand for one invocation of the entry (e.g., [$msg*ws1]) and think time (e.g., (Z1)). Each task has a host processor drawn as an oval, with a multiplicity (e.g., {32}, default value = 1) which can represent multiple cores. A *call* from one entry to another is represented by an arrow labeled with the average number of calls (default value=1).

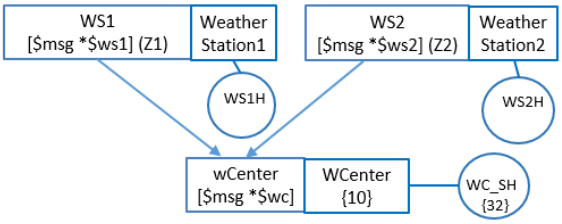

Fig. 2. LQN model of WeatherApp

IV. PRELIMINARY WORK

By carefully observing the popular commercial middleware products (e.g., RMI, Servlet, SPRING, JMS, Web Services), it can be seen that in every middleware the caller and callee must do some marshal/wrapper operation to send and receive messages respectively. This can be seen as a *Mandatory Feature* for every middleware. But, there are many other operations (e.g., NameService, Encryption, ContainerOperations) that may or may not be present in a middleware. We call such operations as *Optional Features*. A minimum middleware is modeled using BMM, and its mandatory and optional features are modeled using feature models and their realizations.

A. Base Middleware Model (BMM)

The BMM contains the minimum number of (i.e., two) tasks and hosts needed to carry out a call in distributed communication. These two tasks represent the caller and the callee. The double bar(||) before the name of a model element tells that this element is a placeholder and it is going to be replaced by the corresponding client or servant element in the composed model. The BMM (and also the realization models that are introduced later) have the following properties: mean number of calls for each call, network latency, multiplicities of tasks and hosts, and call-type (i.e., blocking or non-blocking). Their default values are 1, 0, 1 and blocking respectively. These default values can be modified in the FCD.

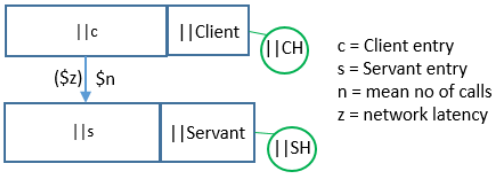

Fig. 3. Base Middleware Model (BMM)

B. Feature Model

The available roles of a middleware are represented using a base feature model (whose root element is MiddlewareFeatures) and a number of sub-feature models. The base feature model has one mandatory feature called *Wrapper*. This feature essentially models the marshalling operation. All other features (e.g., Compression, Encryption, Broker) are optional features.

There are some features (e.g. Broker, ServiceManager) that have their own sub-feature models. A feature with sub-features is shown in thick borders in a feature diagram. A sub-feature can have further sub-features nested within one another. The MiddlewareFeatures are shown in Figure 4. In Figure 5, Broker and ServiceManager features are shown in thick border, indicating they may contain further sub-features.

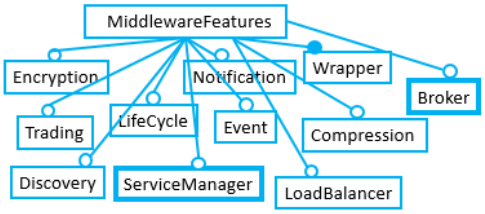

Fig 4. Base feature model

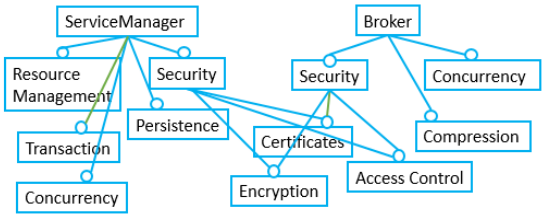

Fig. 5. Sub-feature models of ServiceManager and Broker

C. Feature Realization

Since the BMM is modeled in LQN, the features have to be also realized in LQN. A feature can be realized in two different ways: *Property-Modifying Realization* (PMR) and *Structure-Modifying Realization* (SMR). PMRs do not modify the structure of the BMM, rather they only update model properties (e.g., service demand, call-type etc.). PMRs keep the model compact and are useful when the modeling the concurrency of a feature is not essential. For example, the PMR of encryption feature is as follows:

```
Encryption.demand = $ENCRYPT+$msg*$encrypt
```

The service demand in a realization (as shown in the equation above and also in Figure 6) has two parts: a constant part represented in UPPERCASE letters (e.g., $ENCRYPT) and a variable part represented in camelCase letters (e.g., $encrypt) multiplied by the message size ($msg).

The other type of realization, namely SMR, carry more information than PMR. SMRs increase the size of the model by adding tasks and hosts, but they allow to model concurrency. Figure 6 shows the SMR of encryption feature. SMRs of many other features such as compression and wrapper would also take the same structure, but only the service demand parameters would change (e.g., for compression it would be $compress and $COMPRESS instead of $encrypt and $ENCRYPT). The single bar(|) before the name of a model element tells that this element is going to stay in the composed model.

A feature can be implemented as either PMR, SMR or both and kept in the model library. The appropriate realization is invoked from the library as required by the performance modeller.

Fig. 6. Structure-modifying realization of encryption feature

D. Realization Composition

The BMM is specialized by composing the realizations of the required features to it. For every feature, the following information needs to be provided for its realization composition. (Note that, each of this information has an associated switch to identify it in FCD. Below, the switches are written in parentheses after the name of the information.)

- **Realization type (-t):** can be either PMR or SMR.
- **Realization destination (-d):** tells where a realization to be composed. The destination can be either a call's client, call's servant, both to call's client and servant (which is default), or to a call itself where the feature (e.g. broker) neither adds operations to the client nor to the servant.
- **Realization host (-h):** can be either self, bound or single**.** Self means the realization tasks are deployed at their own separate hosts. Bound (which is the default value for SMR) means the realization task is deployed to its destination tasks host. Whereas, 'single' can be used only when the realization destination is 'both' but both of the realization tasks share a single host. Note that, PMRs are always of 'bound' type.
- **Realization properties (-p):** These are optional list of property parameters to be passed to customize the realization. These properties include call-type (blocking or non-blocking), mean number of calls, multiplicity of realization task and its host etc. This point is not further discussed here due to the lack of space.

  Sometimes the order of realization composition to the BMM may affect the performance overhead. Therefore, a default "order value" is assigned to every realization, causing the realizations to have a "partial ordering" among themselves. Also, the ordering for the recipient of a call is reverse than the ordering for the source of a call. For example, compression realization has a lower order value than encryption realization. Therefore, if a BMM is to be composed with both encryption and compression realizations, then for the source of the call the Middleware Composition Engine (MCE) would compose the compression realization first, then encryption realization. But for the recipient, the encryption realization would be composed first, and then the compression realization. This is logical because the recipient has to decrypt the message first before uncompressing it.

  Note that, one must use SMR if one is interested to keep the ordering of operation in one's model, because PMRs do not preserve concurrency. Also note that the

default partial ordering can be overridden using realization properties.

E. Obtaining Specialized Middleware Model

The list of realizations to be applied to the BMM are described in a text file called Feature Composition Descriptor (FCD). The MCE reads the FCD and composes the FRMs to the BMM, producing Specialized Middleware Model (SMM). The first line of the FCD contains the name of the base model (in this case BMM), followed by a colon (:) and source and destination entries (between a "greater than" sign) that form the call to be updated. Then the type of the call is mentioned after –c switch. The subsequent lines of the FCD file describes the features to be composed. Each of these lines begin with the name of the feature to be applied, followed by composition information separated by switches. If the source model is BMM, then one of the features must be the Wrapper feature since it is a mandatory feature. The last line of the description block contains the –o switch followed by the name of the output model. Below is an example of an FCD

```
BMM:c>s –c block
Wrapper –t struct –d  both –h bound
Compression –t value –d both
Encryption –t struct –d both –h single
SMM –o "SMM1"
```

When this FCD is given as input to the MCE, the SMM shown in Figure 7 is produced. MCE carries out the composition process respecting the partial ordering of the features. The produced SMM is the model that is calibrated and is going to be composed to the BSM.

A powerful property of FCD is that, the performance modeler is not limited to apply features only to the BMM. Rather, any SMR (such as broker) can be referred as the base model in the FCD, and then the subfeatures of that feature can be composed to obtain specialized submodels to be further composed to the BMM.

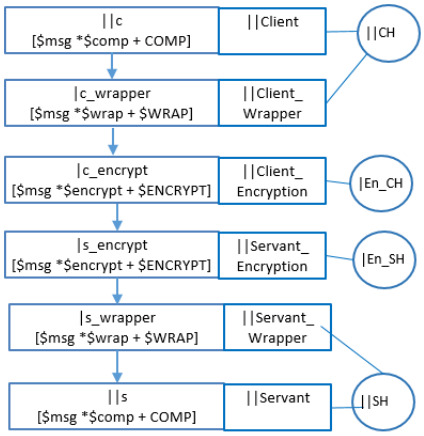

Fig. 7. Specialized Middleware Model (SMM1).

F. Obtaining Specialized Software Model

One or more SMMs are composed to the BMM to obtain a Specialized Software Model (SSM). The description of what middleware to be composed at what calls of the BSM is described in a text file called Middleware Composition Descriptor (MCD). It is possible that many (or even all) of the calls of a BSM use the same middleware. For all those calls, the middleware of choice should be preferably mentioned only once in the MCD. In order to achieve this advantage, the concept of *callgroup* is introduced, that simply identify a set of calls by a single name.

The MCD file starts with the name of the BSM, followed by the –o switch and the name of the SSM. The rest of the file can be divided into two blocks: *callgroup* and *middleware* descriptions. For example, consider the following MCD which describes that the *WeatherApp* BSM presented in Figure 2 should be composed with the Specialized Middleware Model *SMM1* shown in Fig. 7.

```
weatherApp –o weatherAppSpecialized
callgroup ws_wc
WS1 > wCenter
WS2 > wCenter
middleware
ws_wc SMM1 –s Servant_Encryption
```

On the second line of this MCD, a callgroup named *ws_wc* is declared, which has two calls described in the next two lines. Then the middleware section begins. Here only one middleware (*SMM1*) is applied to one callgroup (*ws_wc*). The switch –s tells MCE that the task *Servant_Encryption* is a shared task, meaning it should be created only once and all the calls in the group should share it. The resultant SSM is shown in Figure 8.

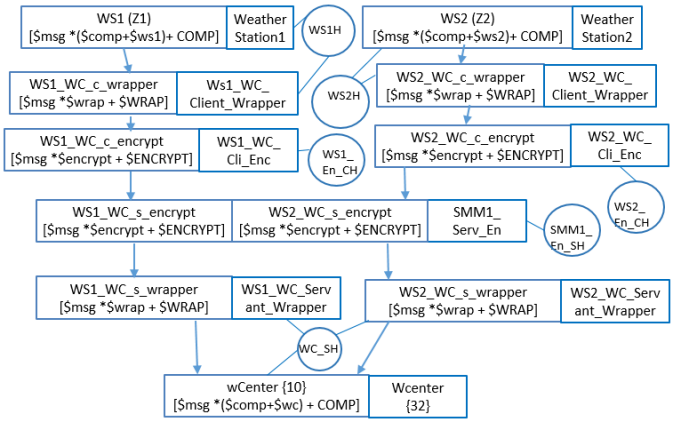

Fig. 8. Specialized Software Model (weatherApp+SMM1)

Note that there is only one task (*SMM1_Serv_En*) that handles encryptions at the server side instead of two tasks. This happened due to the use of the –s switch in the MCD.

V. EXPECTED CONTRIBUTIONS

The main contribution of this research is a flexible, reusable approach to represent the performance impact of a large range of middleware, including many optional features. This is achieved through the following *contributions to knowledge*:

- Modeling the commonality and variability that is present in various middleware. This is done by means of the following sub-contributions: a) proposing a BMM and

identifying its properties so that it can be adapted to a large range of middleware, b) identifying and grouping the middleware features and presenting them in feature models and sub-feature models.

- Solving the frequently occurring modeling problem of "state explosion" by proposing two kinds (SMR and PMR) of realizations for a middleware feature.
- Developing syntax for describing the choice of features and choice of middleware in the forms of FCD and MCD respectively.
- Describing a process to calibrate the SMM to obtain its service time under various workload.
- Developing algorithms for MCE so that it can compose BMM with FRM to obtain SMM, and BSM with SMM to obtain SSM.

This work also has the following *practical contributions*:

- Tool support for storing the BMM and FRMs in a model library.
- Tool support to automatically obtain SMM and SSM from the required input models.
- Calibrating SMMs, and validating the calibrations and the generated models by comparing their performance metrics with those of the software applications running on a test-bed.

VI. PLAN FOR EVALUATION AND VALIDATION

The proposed framework is verified by observing whether the MCE can construct composed models with desired properties and level of details. Validation is done in two phases. First, in order to validate that a large range of middleware can be modeled, we will compose an application software with various types of middleware (e.g., Web Services, Servlet, Spring) and compare the performance metrics of the composed application with SSMs generated by our framework. Second, in order to validate the modeling of various features, we will choose a middleware and calibrate (i.e., get service time) it under various workloads. Then an application software will be composed with that middleware in which different set of calls will use different features. The SSM will be validated by comparing its performance metrics with those of the actual system.

VII. CURRENT STATUS

The author has developed a framework to model the performance completions of large array of middleware and their features. At present, a tool is being developed in Java to automate the entire process of LQN model composition. For the purpose validation, experiments are being run on a test-bed and compared to those generated by the models.. The author expects to complete this research in one year.

ACKNOWLEDGMENT

This research was funded by the Natural Sciences and Engineering Research Council of Canada through its Strategic Network SAVI (Smart Applications on Virtual Infrastructure) and its Discovery Grant programs. The author thanks his co-supervisors, Prof. Murray Woodside and Prof. Dorina Petriu, for their continuous support and invaluable feedback.

REFERENCES

[1] M. Woodside, D.B. Petriu, K. H. Siddiqui, "Performance-related Completions for Software Specifications", *Proc. 24th Int. Conf. on Software Engineering* , Orlando, May 2002.

[2] G. Coulouris, J. Dollimore, T. Kindberg, G. Blair, Distributed Systems Concepts and Design, 5th Edition, Pearson Ed. 2012.

[3] A. Faisal, D. C. Petriu, M. Woodside, "A Systematic Approach for Composing General Middleware Completions to Performance Models", in Computer Performance Engineering (Proc. European Performance Engineering Workshop EPEW14, Florence Sept. 2014), LNCS vol. 8721, Springer, pp 30-44.

[4] A. Faisal, D. C. Petriu, M. Woodside, Network latency impact on performance of software deployed across multiple clouds, Proc CASCON 2013, pp 216-229, Nov 2013.

[5] M. B. Juric, I. Rozman, B. Brumen, M. Colnaric, M. Hericko, "Comparison of performance of Web services, WS-Security, RMI, and RMI-SSL", *Journal of Systems and Software*, n. 79, pp. 689-700, 2006

[6] E. Gómez-Martínez, J. Merseguer "Impact of SOAP Implementations on the Performance of a Web Service-Based Application", *Proc ISPA Workshops*, pp. 884-896, 2006.

[7] G. Franks, T. Al-Omari, M. Woodside, O. Das, S. Derisavi, "Enhanced Modeling and Solution of Layered Queueing Networks", *IEEE Trans. on Software Eng.* v. 35, n. 2, pp. 148-161, 2009

[8] X. Wu., M. Woodside, "Performance Modeling for Software Components," *Proc. 4th Int. Workshop on Software and Performance*, Redwood Shores, Calif., Jan 2004, pp. 290-301.

[9] T. Verdickt, *Performance Analysis of Distributed Systems Based on Architectural System Models*, PhD Thesis, Dept. of Inf. Technology, Universiteit Gent, Belgium, 2007.

[10] S. Becker, H. Koziolek, R. Reussner, "Model-based Performance Prediction with the Palladio Component Model", *Proc. 6th Int. Workshop on Software and Performance*, pp. 56–67, 2007.

[11] J. Happe, H. Friedrich, S. Becker, R. H. Reussner,"A pattern-based performance completion for Message-oriented Middleware", *Proc. 7th Int. Workshop on Software and Performance*, pp. 165-176, 2008.

[12] S. Becker, "Coupled model transformations", *Proc. 7th Int. Workshop on Software and Performance*, pp. 165-176, 2008.

[13] M. Strittmatter, L. Happe, "Compositional performance abstractions of software connectors" *Proc. 3rd International Conf. on Performance Engineering*, pp. 275-278, 2013.

[14] L. Kapova, *Configurable Software Performance Completions through Higher-Order Model Transformations*, PhD Thesis, Karlsruhe Institute of Technology, 2011.

[15] M. Alhaj, D.C. Petriu, "Using Aspects for Platform-Independent to Platform-Dependent Model Transformations", *International Journal of Electrical and Computer Systems*, Vol. 1, Issue 1, pp.35-48, 2012.

[16] J. Kienzle, W. A. Abed, F. Fleurey, J. Jézéquel, J. Klein, "Aspect-Oriented Design with Reusable Aspect Models", Transactions on Aspect-Oriented Software Development, vol. 7, pp. 279 – 327, 2010.