Performance Validation of Mobile Software Architectures

Vincenzo Grassi¹, Vittorio Cortellessa², Raffaella Mirandola¹

¹ Dipartimento di Informatica, Sistemi e Produzione
Università di Roma “Tor Vergata”, Italy
grassiv@acm.org, mirandola@info.uniroma2.it

² Dipartimento di Informatica
Università de L’Aquila, Italy
cortelle@univaq.it

Abstract. Design paradigms based on the idea of code mobility have been recently introduced, where components of an application may (autonomously or upon request) move to different locations, during the application execution. Besides, software technologies are readily available (e.g. Java-based), that provide tools to implement these paradigms. Based on mobile code paradigms and technologies, different but functionally equivalent software architectures can be defined and it is widely recognized that, in general, the adoption of a particular architecture can have a large impact on quality attributes such as modifiability, reusability, reliability, and performance. Hence, validation against specific attributes is necessary and claims for a careful planning of this activity. Within this framework, the goal of this tutorial is twofold: to provide a general methodology for the validation of software architectures, where the focus is on the transition from the modeling of software architectures to the validation of non-functional requirements; to substantiate this general methodology into the specific case of software architectures exploiting mobile code.

1 Introduction

The pervasive deployment of large-scale networking infrastructures is vastly changing the architecture of software systems and applications, leading to more and more applications designed to operate in distributed wide area environments, thus introducing new challenges to architects of scalable distributed applications. Indeed, the large number of available hosts with very different capabilities, connected by networks with varying capacities and loads, implies that the designer is unlikely to know a priori how to structure the application in a way that best leverages the available infrastructure, and that any assumption regarding the underlying physical system, which is made early at the design time, is unlikely to hold later.

This highly heterogeneous and dynamic environment rises problems that could be considered negligible in local area environments. As a consequence, technologies, architectures and methodologies traditionally used to develop distributed applications in local area environments, usually based on the notion of *location transparency*, exhibit several limitations in wide area environments, and often fail in providing the desired quality level. On the contrary, *location awareness* has been suggested as an innovative approach in the design of software applications for wide area environments, to deal since the early design phases with the characteristics and constraints of the different locations. Explicitly considering components location at the application level straightforwardly leads to exploit the location change as a new dimension in the design and implementation of distributed applications. Indeed, *mobile code* design paradigms, based on the ability of moving code across the nodes of a network, have been recently

introduced. Besides, software technologies are readily available (e.g. Java-based), that provide tools to implement these paradigms, so that both have become a central part of the toolset supporting the design of applications for wide area environments

Code mobility, as it is intended in this perspective, should not be confused with the well known concept of process migration, even if the adopted mechanisms to implement them may be similar. Process migration is a (distributed) OS issue, realized transparently to the application (usually to get load balancing), and hence does not represent a tool in the hands of the application designer; on the contrary, code mobility is intended to bring the ability of changing location under the control of the designer, so representing a new tool he/she can exploit to accomplish quality requirements, laying the foundation for a new generation of technologies, architectures, models, and applications.

Using mobile code paradigms and technologies, different but functionally equivalent software architectures can be designed and implemented, and it is widely recognized that, in general, the adoption of a particular architecture can have a large impact on quality attributes of a distributed application such as modifiability, reusability, reliability, and performance [33]. In particular, with respect to performance, code mobility offers to application designers new latitudes in using the systems resources. No longer remote resources must be accessed remotely; instead, (part of) the application can move to use the resources locally. Under the right circumstances, this can reduce both network traffic and network protocol overhead, so reducing the total amount of work done by the system, and improving the performance of the entire system. On the other hand, under the wrong circumstances, the entire system slows down, e.g. because of excessive migration traffic, or increased load at already congested nodes. Hence, validation of mobility-based architectures against specific performance attributes is necessary, and calls for a careful planning of this activity.

The goal of this tutorial is twofold: to provide a general methodology for the validation of software architectures, where the focus is on the transition from the modeling of software architectures to the validation of non-functional requirements; to show how this general methodology can be substantiated in the specific case of software architectures exploiting mobile code. We emphasize the former point in section 2, where we provide a taxonomy of the parameters every approach to software architecture validation should depend on. Then, for the latter point, we review approaches for the validation of mobile software architectures, presenting them in the framework of the above mentioned taxonomy. To provide a basic understanding of the features and performance related costs of different mobile code styles, we first give in section 3 a basic taxonomy of these styles, and then, in section 4, we survey methodologies for performance validation of mobile software architectures. We classify these methodologies as *ad-hoc* and *general-purpose* methodologies. Ad-hoc methodologies consider code mobility “in isolation”, lacking of features to model a whole software application. General-purpose methodologies overcome this limitation by embedding code mobility modeling into some formalism for the specification of software applications. Finally, section 5 concludes the paper and provides hints for future research.

2 Non Functional Requirements Validation

The validation of software architectures can be performed versus functional and/or non-functional requirements (NFR). Approaches basically differ in the two cases, as the former are statements of services the software system should provide, how it should

react to particular inputs and behave in particular situations, whereas NFR are constraints on the services offered by the software system that affect the software quality.

Although validation is being accepted as a crucial activity in software development, yet NFR are often neglected. Economic reasons (such as short time to market and special skills required) and practical reasons (non-functional software aspects are often determined by the latest decisions in the lifecycle, such as the hardware configuration hosting the software system) contribute to the reluctance from the world of software development to adopt an engineered approach to the validation of NFR. A consistent effort has been spent in the last few years in order to fill this gap between software development and validation versus NFR. Beyond every particular approach to the problem, a common ground to work on can be envisaged in the following two issues: (i) determining the amount and the type of information to embed in a software architecture in order to enable its validation versus NFR, (ii) introducing algorithms to translate architecture description languages/notation (augmented with additional information) into a model ready to be validated. Various approaches have been recently introduced for both the issues (see [3] for an overview on this topic). Two attributes appear today crucial to make any approach to the validation of NFR realistically accepted by the software community, that are: transparency, i.e., minimal affection on the software notation and the software process adopted (to cope with issue (i)), and effectiveness, i.e., low complexity algorithms to annotate and transform software models (to cope with issue (ii)).

We propose here a classification of the parameters that all these approaches have to deal with, (mainly aimed at providing guidelines for a structured approach to NFR validation), and we show how a NFR validation approach can be seen as an instantiation from this framework. From section 4 on we focus on a particular class of instances, that are the approaches to the performance validation of mobile software architectures.

The parameters of a methodology to validate a software architecture versus non-functional requirements can be expressed as follows:

architectural style (AS) – the style, if any, adopted to build the software architecture (e.g., client-server, mobile code, etc.¹);

original notation (ON) – the architectural description language/notation used to model the software architecture, as it is from the software development process (e.g., UML, a Process Algebra, etc.);

non-functional attribute (NFA) – the non-functional attribute that is concerned with the set of requirements that the software architecture must fulfill (e.g., reliability, performance, safety, etc.);

missing information (MI) – the information that is lacking in the software architecture description, which is rather crucial for the type of validation that is pursued

¹ Note that an architectural style is defined as a set of construction rules that a developer has to follow while building a software architecture. Depending on the style, those rules can spread over different aspects, such as: types of interactions among components, roles of components, types of connectors, etc. In our case, we focus on architectural styles defined on the capability of components to move.

(e.g., number of invocations of a component within a certain scenario, mapping of components to platform sites, etc.²);

collection technique (CT) – the technique adopted to collect the missing information (e.g., prototype execution, retrieving from a repository of projects);

target model notation (TMN) – the notation adopted for representing the model whose solution provides the non-functional attribute values useful for the validation task (e.g., Petri Nets, Queuing Networks, etc.);

solution technique (ST) – the technique adopted to process the target model and obtain a numerical solution (e.g., simulation, analytical, etc.).

Every validation approach can be reduced to an assignment of values to the above parameters, therefore it may be intended as an instance of the framework that we are outlining. For example, in a Bayesian approach to the reliability validation of a UML-based software architecture, in which the operational profile and the failure probabilities are missing, the following values for the above parameters may be devised:

architectural style (AS) = “don’t care”;

original notation (ON) = Unified Modeling Language;

non-functional attribute (NFA) = reliability;

missing information (MI) = operational profile, failure probabilities;

collection technique (CT) = repository (operational profile), unit testing (failure probabilities);

target model notation (TMN) = Bayesian stochastic model;

solution technique (ST) = numerical simulation.

Obviously the choice of a parameter value is not always independent from the choices of the other ones. In many cases the domain of a choice is restricted to a subset of potential values as a consequence of another parameter assignment. For example, in case a reliability validation has to be performed (i.e., NFA=reliability), it is quite inconvenient to choose a Queuing Model as target (i.e., TMN=Queueing Model), because queues are suitable to represent delays and contentions, and they badly work to combine failure probabilities. Therefore, although a potential domain for every parameter can be defined, in practice limitations may reciprocally induce restrictions of domains while the choices are progressively performed.

In figure 1 we propose a dependency graph, where each node represents one of the above parameters. Two types of edges (i, j) are introduced, both representing a dependency between parameters i and j , with the following semantics:

weak dependency (dashed arrow) – it would be better choosing the value of j after choosing the value of i ; this means that the value assigned to i helps the validation team to better understand which would be the more appropriate choice for j ;

strong dependency (continuous arrow) – a value must be assigned to j after assigning a value to i ; this means that, without knowing the value of i , the validation team cannot perform the choice of j .

² Usually the missing information appears (in the whole approach) either as annotations on the available software architecture description or as an integration of the description itself (in the latter case, for example, as an extension of a software connector).

Figure 1. Graph of dependencies among parameters for NFR validation

Between ON and MI in figure 1 there is a strong dependency as the software model notation determines the set of items and relationships that are available to model the architecture, hence determines also the set of missing items and relationships, depending on the type of validation to perform (NFA). The same type of dependency occurs between NFA and MI, as the information lacking in the architectural design may heavily differ depending on the type of non-functional attribute to validate. As seen in the example above, instead, the dependency between AS and MI can be considered as a weak one, because knowing the architectural style may help to determine the missing information but, in some cases, cannot affect at all this selection (e.g., the architectural style has almost no relationship with the information to be added to UML diagrams in order to perform a reliability analysis of a component based software system). In a similar way it can be considered the weak dependency between AS and ON, because the architectural style may drive software developers to choose the most appropriate notation that better suits that style constraints.

On the other hand, the dependency between TMN and ST has been represented as a weak one because a certain type of model can be solved (almost in all cases) by different techniques with different solution process complexity. Therefore, in the latter case, it would be better to delay the choice of a solution technique after the selection of the model notation, in order to be able to use the technique with lowest complexity. Analogous considerations can be made around the dependency of CT from MI, as if we know what type of information has to be collected then we can devise a much effective technique for the CT task. And, finally, there is a weak dependency between NFA and TMN, derived from the consideration that the same non-functional attribute can be validated using different types of model (e.g. Petri Nets and Queuing Models are suitable models for performance evaluation), but the complexity of the validation process may heavily change if using a notation rather than another, and this depends on the specific non-functional requirements under validation.

For any pair of parameters (i,j) without a connecting path in the graph of figure 1, no evident dependency occurs, namely they can be concurrently chosen because one value does not bring any information on the other one. For example, there is no dependency between CT and TMN, as the way we collect the missing information is not affected by the type of target model notation, which affects, instead, the way we represent that information.

With this classification we have introduced a partial order in the choices that a validation team has to perform in order to accomplish its validation task. Essentially three primary parameters have been identified in figure 1, that are AS, NFA and ON (note that the dependency between AS and ON, being weak, does not always hold). This matches with almost all the practical situations where, starting from an architectural language/notation ON (whose choice should be influenced by the architectural style

AS), and having in mind a non-functional attribute to validate (NFA), all the remaining choices about the target model and the software annotations have to be made.

Once all the parameters have been determined, what remains to do is to introduce a methodology for the translation of the architecture description language/notation, with additional information annotated on it, into the target model ready to be validated. In the framework of the classification scheme of figure 1 such type of translation algorithm can be seen as $ON+MI \rightarrow TMN$. Any translation methodology should match the attributes remarked at the beginning of this section, that is minimal affection on the software process and effectiveness. Several proposals have been formulated in the last few years for translating widely used software notations into target models, but there is still considerable room for further improvements and extensions.

The track of this tutorial is to look, within the framework introduced in this section, at the software validation approaches having AS in the domain of mobile software architectures and NFA in the domain of performance (see section 4).

3 Mobile Code Paradigms

The definition of the *software architecture* of an application, that is its coarse-grained organization in terms of components and interactions among them, represents one of the first and crucial steps in the design stage [5]. Software architectures can be classified according to the adopted *design paradigms* (or *architectural styles*), each of them characterized by specific architectural abstractions/rules and reference structures, that can then be instantiated into actual architectures. Client-server is a traditional example of design style. In this perspective, different mobile code styles can be identified, each characterized by different interaction patterns among components located at different sites, and the available technologies for code mobility provides the *mechanisms* to instantiate them. A review of these technologies is out of the scope of this paper (see [13, 17] for a review of notable examples of them), whereas in this section we present a taxonomy of mobile code styles, aimed at providing a basic understanding of their features and performance related costs, that will be exploited in the subsequent presentation of performance validation methodologies for mobile architectures³.

The taxonomy is largely inspired to the ones presented in [13, 29], and is based on the decomposition of distributed applications into *code components* (the know-how to perform a computation), *resources components* (references to resources needed to perform a computation), *state components* (comprising private data as well as control information that identify a thread of execution, such as the call stack and instruction pointer), *interactions* (events involving two or more components, like exchanging a message), *sites* (locations where processing takes place).

All the basic mobile code styles included in this taxonomy consider a single interaction between components residing at two different sites, aimed at carrying out a given operation. They differ in the distribution of components at the two sites at the beginning of the interaction, in the interaction pattern, and in the distribution of components at the end of the interaction, as shown in table 1, where **A** and **B** denote the components that participate in the interaction, **C** and **R** denote the code and

³ Of course, knowledge of the characteristics of a particular mobile code technology would be necessary to finely tune, in the late phases of the development cycle, the performance model of mobile software architecture.

resources (parameters) needed to perform the operation, while L_1 and L_2 are two different sites.

The styles identified are: *remote execution* (REX), *code on demand* (COD), and *mobile agent* (MA). In all the styles, component **A** initiates the interaction.

Table 1. Mobile code styles

Style	before interaction		after interaction	
	L_1	L_2	L_1	L_2
REX	A , C, R	B	A , C, R	B , C, R
COD	A , R	B , C	A , C, R	B , C
(weak/strong) MA	A , C, R	B	-	A , C, B , R

In the REX style⁴, both the code and the parameters needed to perform the operation are present at the **A** site, that ships both of them to the **B** site, requesting **B** to perform on its behalf the operation (exploiting also other code and resource components already present at site L_2). In general, a reply could be sent to **A** at the completion of the operation. Java *servlets* (“push” case) [37] and the REV scheme [34] are implementations of this style. The COD style is somehow the complementary of REX, since in this case it is **B** that owns the code C and ships it to **A** on its request. Java *applets* [36] are an implementation of this style.

In the two styles examined so far only data and “passive” pieces of code are sent from one location to another one, to re-direct the location of processing, while the location of the active components (in particular, their state component that identifies their thread of execution), remain fixed. On the other hand, in the MA style, an active component moves itself (i.e. its state component) together with needed code and parameters to the **B** site, to exploit locally the resources of that site. A further distinction can be made between a *weak* and a *strong* MA style, where in the former only data state (i.e., private variables) is transferred, while in the latter also the execution state (i.e., instruction pointer and control stack) is transferred. In the case of strong MA, the transferred component can immediately resume its execution at the new site from the exact point where it was stopped, at the expense of freezing, packing and transferring all the computation state, which could be quite heavy. In the case of weak MA, the amount of transferred information (and the work done to capture it) may be much smaller, but some method must be devised to decide, on the basis of the encoded information, where to restart execution after migration. Several technologies that implement weak and strong MA paradigms are reviewed in [13, 17].

4 Performance Validation of Mobile Software Architectures

In this section we propose a classification of those performance validation approaches proposed in the literature that apply to mobile software architectures. In the perspective of the framework in section 2, these are the approaches spreading AS over the mobility-based styles and NFR over the performance requirements. All the remaining parameters

⁴ We prefer to call this style “remote execution (REX)” as in [29] instead of the often used denomination “remote evaluation (REV)” to avoid ambiguity with the REV scheme proposed in [34], which is a particular *mechanism* that implements this style.

may freely vary among different approaches, and we push this classification to show, where feasible, values they assume in each approach instance.

The approaches here presented are partitioned into two main classes: *ad-hoc* and *general-purpose* methodologies.

The contribution of ad-hoc methodologies consists of cost models for a single interaction between components, for different mobile code paradigms. The cost models are provided either as closed-form analytic expressions, or as dynamic models (Petri net) which are numerically evaluated.

General-purpose methodologies can be further classified as methodologies based on formal specifications of the software behavior (process algebras), and methodologies based on semi-formal specifications (Unified Modeling Language). For both cases we highlight the advantages and present some methodologies proposed in the literature to generate a performance model starting from different notations for mobility-based architectural models.

4.1 Ad-hoc Models

Ad-hoc models consider code mobility “in isolation”, providing cost models for a single interaction between components, for different mobile code styles. From our validation framework viewpoint, they consider $NFA \in \{\text{total network load, total processing time}\}$, while the adopted TMN consists of either closed-form analytic expressions, or dynamic models that can be numerically evaluated. Because of the lack of features to model a whole application, ad-hoc models cannot be considered as general validation methodologies. However, they contribute to clarify the dependencies between (AS, NFA) and MI, when AS spreads over mobility-based styles, by giving insights about the quantities that affect the selected NFAs, and hence about the information that should be collected in any validation methodology for these attributes.

We review ad-hoc models proposed in the literature⁵, presenting all of them in a unified scenario, consisting of a single “interaction session” between two partners (**A** and **B**) residing at different locations, with **A** requesting to **B** the execution of an operation that can be articulated in N “low level” requests, and corresponding (intermediate) results.

Closed-Form Models. We present all the models reviewed in this section as special instances of general closed-form expressions for the average total network load, and the average total processing time, denoted as L^X and T^X respectively, with $X \in \{\text{REX, COD, MA}\}$.

Common parameters used in all the closed-forms (hence representing the missing information MI for the considered measures) are:

req : average size (in bytes) of a “low level” operation request

rep : average size (in bytes) of a “low level” result

α^X : communication overhead⁶; B_{req}^X : average size of a single request;

⁵ Most of the papers considered in this section also present models for the client-server style, for the sake of comparison with code mobility styles.

⁶ This coefficient takes into account the overhead caused by additional information needed for connection setup and message encapsulation; in general, the α^X coefficient, $X \in \{\text{COD, MA}\}$,

B_{rep}^X : average size of a single reply;

τ : network throughput (in bytes/sec); Δ^X : average network latency;

M^X : average marshalling/unmarshalling time of a request/reply;

T_{req}^X : average processing time (for **A**) of a request;

T_{rep}^X : average processing time (for **B**) of a reply;

φ^X : semantic compression factor for replies ($0 < \varphi^X \leq 1$).

Other parameters, used only in some closed forms, are listed in the following.

REX style. **A** assembles the original requests into less than N “high-level” operations requests (at most, they are all assembled in a single operation), sends them together with the corresponding code to **B**, and gets the corresponding replies. Closed-form expressions for the network load and the processing time are:

$$L^{REX} = R^{REX} \alpha^{REX} (B_{code}^{REX} + \varphi^{REX} B_{rep}^{REX})$$

$$T^{REX} = \frac{1}{\tau} L^{REX} + R^{REX} (\Delta^{REX} + M^{REX} + T_{req}^{REX} + T_{rep}^{REX})$$

where: R^{REX} : number of “high level” operation requests needed to complete the interaction

B_{code}^{REX} : average size of the code of a high level operation sent to **B** for remote evaluation

Table 2. Proposed parameters instantiations in closed-forms for the REX style

	R^{REX}	α^{REX}	B_{code}^{REX}	B_{rep}^{REX}	φ^{REX}	Δ^{REX}	M^{REX}	T_{req}^{REX}	T_{rep}^{REX}
[2]	1	> 1	> 0	$N rep$	1, $1/N$	-	-	-	-
[21] ⁷	$\geq 1, < N$	> 1	> 0 $(req \cdot N / R^{REX})$	> 0 $(rep \cdot N / R^{REX})$	1	0	> 0	> 0	> 0

COD style. The scenario modeled by the following closed-forms assumes that **A** requests the execution of less than N high-level operations to **B**; to model a COD-based interaction, we assume that **B**, if the needed code is present at its site, executes the operations, otherwise fetches the code from some other site. Closed-form expressions for the network load and the processing time are:

$$L^{COD} = R^{COD} \alpha^{COD} (B_{req}^{COD} + P_{code}^{COD} (B_{fetch}^{COD} + B_{code}^{COD}) + \varphi^{COD} B_{rep}^{COD})$$

$$T^{COD} = \frac{1}{\tau} L^{COD} + R^{COD} (\Delta^{COD} + M^{COD} + T_{req}^{COD} + T_{rep}^{COD})$$

REX, MA} may be dependent on the size of the data exchanged in the communication, that is the overhead coefficient for data of size Y is: $\alpha^X = \alpha^X(Y)$.

⁷ The authors in [21] calls the REX style as “stationary agent access” (SA) style.

where:

R^{COD} : average number of “high level” operations needed to complete the interaction;

P_{code}^{COD} : probability that the code for a high level operation is not already present at the location of **B**;

B_{fetch}^{COD} : average size of the request for the code of a high level operation sent by **B**;

B_{code}^{COD} : average size of the code of a high level operation.

Table 3. Proposed parameters instantiations in closed-forms for the COD style

	R^{COD}	α^{COD}	B_{req}^{COL}	P_{code}^{COL}	B_{code}^{COL}	B_{fetch}^{COL}	B_{rep}^{COL}	φ^{COD}	Δ^{COD}	M^{COD}	T_{req}^{COL}	T_{rep}^{COL}
[2]	1	> 1	> 0	$\geq 0, \leq 1$	> 0	> 0	N_{rep}	$1, 1/N$	-	-	-	-

We point out that the model for the processing time of the COD style has been extrapolated from the models for other styles, since no explicit model for the processing time of this style is present in the literature. For this reason no specific instantiation of parameters in the latter four columns of table 3 is given.

MA style. **A** moves to the **B** site, to interact locally with **B**. Then, it can go back to the starting site, or move to some other site, carrying with it the information accumulated at the **B** site; in the latter case, it can optionally send back the collected information to the starting site. Closed-form expressions for the network load and the processing time are:

$$L^{MA} = \alpha^{MA} \left((P_{code}^{MA} + \beta_{back} \beta_{code}) B_{code}^{MA} + (1 + \beta_{back}) (B_{state}^{MA} + B_{data}^{MA}) + (1 - \beta_{back}) \beta_{rep} \varphi^{MA} B_{rep}^{MA} \right)$$

$$T^{MA} = \frac{1}{\tau} L^{MA} + \Delta^{MA} + M^{MA} + T_{req}^{MA} + T_{rep}^{MA}$$

where: P_{code}^{MA} : probability that the code of the mobile agent is not already present at the location of **B**;

B_{code}^{MA} : average size of the mobile agent code;

B_{state}^{MA} : average size of the mobile agent execution state;

B_{data}^{MA} : average size of the mobile agent data (before the interaction starts);

$\beta_{back} = \begin{cases} 1 & \text{if the agent goes back to the starting location;} \\ 0 & \text{otherwise} \end{cases}$;

$$\beta_{code}^8 = \begin{cases} 1 & \text{if the agent code is not retained at the return location;} \\ 0 & \text{otherwise} \end{cases};$$

$$\beta_{rep} = \begin{cases} 1 & \text{if a "high level" reply is sent to the starting location;} \\ 0 & \text{otherwise} \end{cases}.$$

Table 4. Proposed parameters instantiations in closed-forms for the MA style

	[35]	[9]	[2]	[22]	[21]
α^{MA}	1	1	> 1	> 1	> 1
P_{code}^{MA}	$\geq 0, \leq 1$	$\geq 0, \leq 1$	1	1	1
B_{code}^{MA}	$> 0 (\geq N \cdot req)$	> 0	> 0	> 0	> 0
B_{state}^{MA}	> 0	> 0	> 0	> 0	> 0
B_{data}^{MA}	≥ 0	≥ 0	≥ 0	≥ 0	0
φ^{MA}	$> 0, \leq 1$	1	1, $1/N$	$> 0, \leq 1$	1
B_{rep}^{MA}	$N \cdot rep$	$N \cdot rep$	$N \cdot rep$	$N \cdot rep$	> 0
β_{back}	0	0, 1	0	0	1
β_{code}	-	1	-	-	0
β_{rep}	0, 1	0	0, 1	1	1
Δ^{MA}	$(1 + \beta_{rep})\delta^9$	-	-	0	0
M^{MA}	$2\mu(B_{data}^{MA} + B_{state}^{MA} + \beta_{rep} B_{rep}^{MA})^{10}$	-	-	> 0	> 0
T_{req}^{MA}	0	-	-	0	> 0
T_{rep}^{MA}	0	-	-	> 0	> 0

In all the considered models for the MA style (with the exception of [22], where it is unspecified) it is assumed that, after the completion of the interaction, the agent data grow as follows: $B_{data}^{MA} \leftarrow B_{data}^{MA} + \varphi^{MA} B_{rep}^{MA}$. In this way it is modeled the (possible) accumulation of information collected by the mobile agent as it visits new sites.

With regard to the parameters instantiations shown in table 4, it should be noted that the marshalling/unmarshalling overhead of [35] is calculated under the assumption that the agent code is already available in transport format. [22] analyzes a broadcast

⁸ If the agent goes back to the starting **A** location, then $\beta_{code} = 0$ means that only its data (the original data plus the ones collected at **B** location) and execution state actually go back, since a copy of the (immutable) code has been retained there; the term $(1 - \beta_{back})$ that multiplies β_{rep} means that only if the agent does not go back to the starting location, a reply could be sent there.

⁹ δ denotes the average roundtrip time (in secs.).

¹⁰ $\mu > 0$ represents a marshalling/unmarshalling factor (in secs/byte); this factor is multiplied by 2 to take into account both marshalling and unmarshalling of a message.

data filtering application, where the MA paradigm (with filtering at the server) is compared against broadcast filtering at the client (hence no other mobile code style is modeled, while closed-forms are presented for network load and processing time in the broadcast case). Moreover, [4] builds on the model presented in [2] to derive closed-forms expressing the network load caused by the MA style in conjunction with multicast protocols to deliver the agent to multiple destinations.

Dynamic models. The interaction scenario modeled by the models considered in this section is the same considered before, but, differently from the models of the previous section, here they adopt $TMN = \{\text{Petri net}\}$ and are limited to $NFA = \{\text{total processing time}\}$ [27]. Moreover, these models refer only to $AS \in \{\text{REX, MA}\}$, whereas no model is provided in [27] for the COD style. We do not present details of these models. Anyway, the parameters are basically the same as the ones in the previous section, except for the fact that many of them are instantiated as random variables with a given probability distribution, rather than as constant (average) values, and can be listed as follows (with $X \in \{\text{REX, MA}\}$):

$$\begin{aligned} \alpha^X &= 1; & P_{code}^X &= 1; & B_{rep}^X &: \text{uniformly distributed in } [rep_{min}, rep_{max}] \\ B_{data}^{MA} &= 0; & \beta_{back} &= 0; & \beta_{rep} &= 0 \text{ or } \beta_{rep} = 1; \\ \Delta^X &= 0; & M^X &= 0; & T_{req}^X \text{ and } T_{rep}^X &: \text{exponentially distributed.} \end{aligned}$$

The only remarkable difference with the previous closed-form models concerns the semantic compression that in [27] is modeled only for the MA paradigm as a constraint on the growth of the B_{data}^{MA} parameter, as follows: $B_{data}^{MA} \leftarrow B_{data}^{MA} + D_{rep}^{MA}$, with D_{rep}^{MA} uniformly distributed in $[rep_{min}, n \cdot rep_{max}]$, where $n > 1$ is a parameter independent of the agent history. Hence, differently from the closed-form models of the previous section, the agent data do not grow linearly with the number of visited locations, if the agent visits more than one location.

4.2 General-Purpose Formal Models: Process Algebras

The validation methodologies considered in this section are based on the selection of $ON = \{\text{Process Algebras}\}$, and do not focus on specific NFAs. Hence, the adopted TMN is general enough to allow the evaluation of different NFAs, and consists of $TMN = \{\text{Stochastic Process Algebras} + \text{associated Markov Processes}\}$; correspondingly, possible STs consist of any solution technique suitable for this TMN. From these choices it results that MI includes at least the (exponential) completion rates of all the activities that are modeled in the adopted Process Algebra.

Process algebras are well-known formalisms for the modelling and analysis of parallel and distributed systems. What makes them attractive as ON for the evaluation of large and complex systems, are mainly their compositional and abstraction features, that facilitate building complex system models from smaller ones. Moreover, they are equipped with a formal semantics, that allows a non ambiguous system specification, and a calculus that allows to prove rigorously whether some functional properties hold. Stochastic process algebras are an extension of these formalisms with stochastic features for the specification of system activities duration, that allow the analysis of quantitative non-functional properties. We defer to the vast available literature for details about the general characteristics of these formalisms (e.g., [14]), and focus in this section on process algebras for the modeling of mobile software architectures. We

only provide their (partial) formal syntax and informal descriptions of the corresponding semantics, aimed at illustrating the salient features of different approaches to formal modeling of code mobility. Then, we illustrate a proposed methodology for the translation of this formalism into a TMN for the analysis of NFAs, with TMN consisting of Markov processes (and stochastic process algebras as intermediate TMN).

A process algebra is a formal language, whose syntax basically appears like this:

$$P ::= \mathbf{0} \mid \pi.P \mid P + P \mid P \parallel P \mid \dots^{11}$$

where $\mathbf{0}$ denotes the “null” (terminated) process that cannot perform any action, $+$ and $\parallel$ denote process composition by non-deterministic alternative or parallelism, respectively, and $\pi.P$ denotes the process that performs *action* $\pi \in Act$, and then behaves as P (where Act is a set of possible actions). Process algebras for mobility modeling basically differ in the set Act of actions the defined processes can perform. We group them into two sections, based on the way used to model the location of components.

Example 1. To illustrate some of the modeling approaches reviewed in this and next section, we will use a simple application example based on a traveling agency scenario, where a travel agency periodically contacts K flying companies to get information about the cost of a ticket for some itinerary. The agency exchanges a sequence of N messages with each company, to collect the required information. Using a traditional client-server approach, this means that the agency should explicitly establish N RPCs with each company to complete the task. On the other hand, with a REX approach, the agency could send a code encompassing all the N messages along with some gluing operations, to be executed by each company, getting only the final reply. Within a COD approach, we could think that the agency makes an overall request to each company, and that it is the responsibility of each company to possibly get somewhere the needed code to fulfill the request. Finally, in an MA approach, the agency could deliver an agent that travels along all the K companies getting locally the information, and then reports it back to the agency. *EndOfExample1*

Models with “Indirect” Location Specification. Algebras listed in this section can be considered as a direct derivation from CCS-like algebras [23], and are characterized by the modeling of mobility as a change in the links that connect processes. Before reviewing them, we briefly illustrate a basic CCS-like algebra. In this case, we have $\pi \in \{\tau_i (i = 1, 2, \dots), \mathbf{in}_x, \mathbf{out}_x\}$, where τ_i denotes a “silent” (internal) action of a process¹², while $\mathbf{in}$ and $\mathbf{out}$ are input and output actions, respectively, along the link named x , that can be used to synchronize a process with another parallel process that executes their output or input counterparts along the same link. For example, if two processes are specified as follows:

$$P := \mathbf{out}_a.P_i \qquad Q := \mathbf{in}_a.Q_i$$

from these definitions we get that $P \parallel Q$ evolves into $P_i \parallel Q_i$, that is, processes P and Q synchronize (i.e., wait for each other) thanks to a communication along link a , and

¹¹ Note that, for the sake of simplicity, this syntax is incomplete, since we are omitting constructs to define abstraction mechanisms, or recursive behavior, etc.

¹² Subscript i is used to distinguish different internal actions, which is useful for modeling purposes.

then prosecute in parallel (possibly independently of each other, if no other synchronizing communication takes place in their following behavior).

π -calculus [24]. This algebra, besides synchronization between parallel processes, allows also link names communication, so that we can change the links a process uses to communicate with other processes. The possible system actions are $\pi \in \{\tau_i (i = 1, 2, \dots), \mathbf{in}x, \mathbf{out}x, \mathbf{in}x(y), \mathbf{out}x(Y)\}$, where in addition to the above definitions, Y (y) is a “link name” (link variable), sent (received) over the link named x . For example, with the following specifications:

$$P_1 := \mathbf{out}a(b).P_3 \quad P_2 := \mathbf{out}a(c).P_3 \quad Q := \mathbf{in} a(y).\mathbf{out}y.Q_I$$

we get that $P_1 \parallel Q$ evolves into $P_3 \parallel \mathbf{out}b.Q_I$, while $P_2 \parallel Q$ evolves into $P_3 \parallel \mathbf{out}c.Q_I$. In this example, the parallel composition of Q with P_1 or P_2 gives rise to the evolution of Q into a process that communicates along the link b or c , respectively, and then behaves as process Q_I .

$HO\pi$ -calculus [30]. Besides the operations of π -calculus, this algebra allows also the communication of process names, so that we can change the behavior of the receiving process. The possible system actions are again $\pi \in \{\tau_i (i = 1, 2, \dots), \mathbf{in}x, \mathbf{out}x, \mathbf{in}x(y), \mathbf{out}x(Y)\}$, but, in addition to the above definitions, Y (y) may also be a “process name” (process variable) besides a link name (variable), sent (received) over the link named x . For example, with the following specifications:

$$P_1 := \mathbf{out} a(R).P_3 \quad P_2 := \mathbf{out} a(S).P_3 \quad Q := \mathbf{in} a(z).z.Q_I$$

we get that $P_1 \parallel Q$ evolves into $P_3 \parallel R.Q_I$, while $P_2 \parallel Q$ evolves into $P_3 \parallel S.Q_I$. In other words, the parallel composition of Q with P_1 or P_2 gives rise to the evolution of Q into a process that behaves like process R or S , respectively, and then as process Q_I .

*Example 2*¹³. Let us consider the system of example 1 in the case of $K=2$ flying companies, with F_i and a_i ($i=1, 2$) denoting a company and the channel used to communicate with it, C denoting the overall code corresponding to the N “low level” interactions, and R_i the overall response collected at company F_i . Using $HO\pi$ -calculus, this application could be modeled as follows, in case of REX paradigm (where Sys models the overall application):

$$TravAg = \mathbf{out}a1(C).\mathbf{in}a1(x).\mathbf{out}a2(C).\mathbf{in}a2(x).TravAg$$

$$F1 = \mathbf{in}a1(z).z.\mathbf{out}a1(R1).F1$$

$$Sys = TravAg \parallel F1 \parallel F2$$

EndOfExample2.

Models with “Direct” Location Specification. The above approaches suggest as ON for the modeling of mobile architectures a process algebra where the location of a process is indirectly defined in terms of its connectivity, i.e. the link names it sees and the identity of the processes it can communicate with at a given instant of time using those links; hence, the location of a process can be changed by changing the links it sees (by sending it new link names, as in the π -calculus, or by sending the process itself, i.e., its name, as in the $HO\pi$ -calculus to a receiving process that has a

¹³ Adapted from [25].

different location (again, defined by its connectivity)). Other process algebras approaches have been defined where the location of processes is directly and explicitly defined, giving it a first class status, so allowing for a more direct modeling and reasoning about problems related to locations, such as access rights or code mobility, thus making these algebras somewhat more appealing as ON for mobile architectures. Two of these approaches are the *ambient* calculus and *KLAIM*. In the following we briefly outline some of their features. As before, the presentation is far from complete, with the main goal of only giving some flavor of the way they adopt to model process location and mobility in a process algebras setting.

Ambient calculus [7]. In this formalism the concept of *ambient* is added to the basic constructs for processes definition and composition described above. An ambient has a *name* that specifies its identity, and can be thought of as a sort of boundary that encloses a set of running processes. Ambients, denoted as $n[P]$, where n is the ambient name and P is the enclosed process, can be entered, exited or opened (i.e., dissolved) by appropriate operations executed by a process, so allowing to model movement as the crossing of ambient boundaries. Ambients are hierarchically nested, and a process can only enter an ambient which is sibling of its ambient in the hierarchy, and can exit only into the parent ambient of its ambient; hence, moving to a “far” ambient in the ambients hierarchy requires, in this formalism, the explicit crossing of multiple ambients. The mobility operations for an ambient $n[.]$ are denoted by **inambn**, **outambn**, **openn**, respectively¹⁴. In general, a process cannot forge them by itself, but receives them thanks to the communication operations **in** and **out**. Hence, a process receiving one of such operations through a communication actually receives a capability for it, being allowed to execute the corresponding operation on the named ambient. The (partial) formal syntax of this algebra is then as follows:

$$P ::= 0 \mid \pi.P \mid P + P \mid P \parallel P \mid n[P] \mid \dots$$

with $\pi \in \{\tau_i (i = 1, 2, \dots), \mathbf{in}(x), \mathbf{out}(M), \mathbf{inambn}, \mathbf{outambn}, \mathbf{openn}\}$, where x is a variable and M stands for either an ambient name (n), or a capability for an ambient (either **inambn**, or **outambn**, or **openn**). Communication is restricted to be local, i.e. only between processes enclosed in the same ambient. Communication between non local processes requires the definition of some sort of “messenger” agent that explicitly crosses the required ambient boundaries bringing with itself the information to be communicated. Alternatively, a process can move itself to the ambient of its partner before (locally) communicating with it. In both cases, the messenger or the moving process must possess the needed capabilities.

KLAIM (Kernel Language for Agents Interaction and Mobility) [11]. This formalism allows to define a net of locations that are basically not nested into each other, with direct communication possible, in principle, between processes located at any location, differently from the ambient calculus (anyway, the extension to nested location is possible). Another remarkable difference with the ambient calculus, and with all the previously mentioned algebras, consists of the adoption of a *generative* (rather than message passing) style of communication, based on the use of *tuple spaces* and the

¹⁴ Note that the ambient operations are named **in**, **out** and **open** in the original paper [7]; we have renamed them to avoid confusion with the names used in this paper to denote the message passing communication operations.

communication primitives of the *Linda* coordination language [8]. Tuple spaces are linked to locations, and interaction between processes located at different locations can happen by putting or retrieving the opportune tuple into the tuple space at a given location. Again, the (partial) formal syntax of this algebra is as follows:

$$P ::= \mathbf{0} \mid \pi.P \mid P + P \mid P \parallel P \mid \dots$$

with $\pi \in \{\tau_i (i = 1, 2, \dots), \mathbf{in_t}(t)@l, \mathbf{read_t}(t)@l, \mathbf{out_t}(t)@l, \mathbf{eval_t}(t)@l, \mathbf{newloc}(u)\}$, where the indicated operations are the usual Linda-like operations on a tuple t , restricted to operate on the tuple space associated to the l location¹⁵. Moreover, the $\mathbf{newloc}(u)$ operation allows a process to create a new (initially private) location that can be accessed through the name u . Note that the fields of a tuple may be either values, or processes, or localities, or variables of the previous types. This allows a simple modelling of all the mobile code styles (namely REX, COD and MA), as shown in [11].

Other formal models. Other approaches to mobility modeling (and formal verification of functional requirements for mobile systems) have been proposed, not based on a process algebras framework: Mobile UNITY [26], Mob_{adt} [12], COMMUNITY [38]. The former two have a temporal logic based semantics, and the latter has a category theory based semantics. We do not consider explicitly these approaches since the translation methodology from ON to TMN described in the following section has been presented in the framework of process algebras.

From Process Algebras to Performance Models. In this section we present a methodology for the translation from ON to a suitable TMN, when the application model is built using $\text{ON} = \{\text{Process Algebra}\}$. This approach has been presented in the framework of π -calculus and $\text{HO}\pi$ -calculus, but it is applicable to any formalism with an operational semantics, like all the process algebras presented above. We start with a brief review of operational semantics, and then outline the approach, presented in [25].

The operational semantics of a process specified using the syntax of a given process algebra is given by a *labeled transition system*, i.e. (informally) a graph that represents all the possible system evolutions; each node of the graph represents a particular system state, while a transition represents a state change, and the label associated to a transition provides information about the “activities” that cause the corresponding state change. The transition relation (i.e. which are the states reachable in one step from a given state, and the associated labels) is specified by giving a set of syntax-driven rules, in the sense that they are associated to the syntactic rules of the algebra. Each rule takes the form $\frac{\text{Premises}}{\text{Conclusion}}$ whose meaning is that whenever the premises (that can be

interpreted as a given computational step) occur, then the conclusion will occur as well. (Simplified) examples of such rules are the following:

$$\frac{}{\pi.P \xrightarrow{\pi} P}, \quad \frac{P \xrightarrow{\pi} P'}{P + Q \xrightarrow{\pi} P'}, \quad \frac{P \xrightarrow{\pi} P'}{P \parallel Q \xrightarrow{\pi} P \parallel Q}, \quad \frac{P \xrightarrow{\text{out } \tau} P', Q \xrightarrow{\text{in } \tau} Q}{P \parallel Q \xrightarrow{\tau} P \parallel Q'}$$

¹⁵ Note that the tuple operations are named **in**, **out**, **read** and **eval** in the original paper [11]; we have renamed them to avoid confusion with the names used in this paper to denote the message passing communication operations.

Note that the third rule specifies a transition relation for parallel independent processes, while the fourth rule specifies a transition relation for parallel processes that synchronize themselves through a communication operation¹⁶.

Example 3. The labeled transition system obtained applying rules as the ones specified above to the *Sys* model of example 2 (using $HO\pi$ -calculus) is given by (assuming $C = (\tau_1 + \tau_2).0$):

EndOfExample3

In general, methodologies for the translation of process algebras into a TMN suitable for the analysis of NFAs are based on the association of a stochastic duration, that hence represent a MI for these methodologies, to the activities specified in the process algebra, so obtaining, as a first step, a stochastic process algebra that, in our framework, can be considered as an intermediate notation toward the final TMN. Then, starting from a stochastic process algebra model, a stochastic duration can be associated to each label in the labeled transition system that represents the operational semantic of the original model. If this duration is exponentially distributed (hence expressed by an exponential rate), then we get a continuous time Markov chain. A fundamental problem to make practically usable these approaches is how giving a meaningful value to the exponential rates associated to the transitions, since, in a realistic system, their number is very high. The idea of [25], is to associate to each transition a label that does not merely register the action associated to that transition (e.g., τ_1 , as in example 3), but also the inference rules used during the deduction of the transition, so to keep trace of the “underlying operations” that lead to the execution of that action. For instance, in example 3 the operation underlying the execution of action τ_1 is a selection operation between the two concurrently enabled operation τ_1 and τ_2 . These “enhanced” labels can be used to define a systematic way for deriving the rates to be associated to the system transitions. The enhanced labels are built using symbols from a suitable alphabet (e.g., $\{+, \parallel, \dots\}$), to record the inference rules used during the derivation of the transitions. For example, the transition rules given above would be rewritten as follows to get enhanced labels¹⁷:

$$\frac{}{\pi.P \xrightarrow{\pi} P}, \frac{P \xrightarrow{\theta} P'}{P + Q \xrightarrow{+\theta} P'}, \frac{P \xrightarrow{\theta} P'}{P \parallel Q \xrightarrow{\parallel\theta} P' \parallel Q}, \frac{P \xrightarrow{\theta \text{out } x} P', Q \xrightarrow{\theta' \text{in } x} Q'}{P \parallel Q \xrightarrow{\{\parallel\theta \text{out } x, \theta' \text{in } x\}} P' \parallel Q}$$

¹⁶ In the “standard” semantics of process algebras [24], the label of this latter rule is equal to τ , that is an invisible action, since the two matching input and output operations “consume” each other, making them unobservable for an external “observer” (e.g. a process running in parallel).

¹⁷ Again, we are introducing a simplification: in a complete specification different symbols should be used to distinguish the selection of the left or right alternative in a parallel or alternative choice composition (see [25]).

where an enhanced label is, in general, given by $\theta = \vartheta\pi$, with π denoting, as before, a particular system action, and $\vartheta \in \{+, \parallel, \dots\}^*$ denoting the sequence of inference rules followed to fire that action.

Example 4. The transition system of example 3 would be enhanced as follows:

EndOfExample4

Using the enhanced labels, the rate of a transition can be calculated by defining suitable functions, as follows:

$$\$_b : Act \rightarrow \mathfrak{R}^+, \quad \$_s : \{+, \parallel, \dots\} \rightarrow \mathfrak{R}^+, \quad \$: \{+, \parallel, \dots\}^* \bullet Act \rightarrow \mathfrak{R}^+$$

where $\bullet$ denotes the concatenation operator, $\$_b$ defines the basic exponential rate of an action in a reference architecture dedicated only to the execution of that action without any interference, while $\$_s$ defines a slowing factor in the execution of an action due to the execution of some underlying operation in the target architecture where the action is actually executed. $\$$ is the function that calculates the actual exponential rate of the transition, taking into account all possible interferences, and can be basically recursively defined using $\$_b$ and $\$_s$, as follows¹⁸:

$$\$(\pi) = \$_b(\pi), \quad \$(\sigma\vartheta\pi) = \$_s(\sigma)\$(\vartheta\pi), \quad \sigma \in \{+, \parallel, \dots\}, \vartheta \in \{+, \parallel, \dots\}^*$$

By suitably defining the functions $\$_b$ and $\$_s$, we can limit the problem of calculating meaningful transition rates to the problem of defining only the cost of the “primitive” system actions, and of the slowing factors caused by a particular target architecture. Note that, with respect to our validation framework, this means that the methodology of [25], besides defining a method for the translation from ON to TMN, also gives strong indication about what MI should be collected. Having this information, the calculation of the actual transition rates can be completely automated (but it should be remarked that the definition of the above functions is, in general, quite an ambitious task). It should be noted also that, by changing the definition of $\$_s$, we can also analyze the impact on performance of different target architectures.

Once the rates of all the possible transitions from a given state (representing the system behaving like process P_i) have been determined, the overall rate from state P_i to another state P_j which is one-step successor of state P_i is given by:

$$q(P_i, P_j) = \sum_{P_i \xrightarrow{\vartheta\pi} P_j} \$(\vartheta\pi)$$

(note that, in general, more than one transition from state P_i to state P_j may be present in the graph of the transition system). The process so obtained gives us information about the rate at which the system actions are performed. In general, to carry out a

¹⁸ Note that in this presentation we do not have explicitly considered the problem of how calculating the rate of synchronization (i.e., communication) operations; for a discussion of this topic, see [18].

performance evaluation, we need also a *reward structure*, that associates to each state of the process a performance-related reward (or cost), that constitutes another possible MI that should be collected. In the described approach, these rewards could be calculated using a procedure similar to the one followed to calculate the transition rates. Besides being added to the Markov process derived from a process algebra specification, rewards could also be formally included in a stochastic process algebra, to allow formal reasoning about them. For a discussion about this topic, which is beyond the scope of this paper, see [18].

4.3 General-Purpose Semi-Formal Models: UML

The advantage of using process algebras as ON mainly consists in the possibility of a rigorous and non ambiguous modeling activity. However, the use of these formal notations does not have yet gained widespread acceptance in the practice of software development. On the contrary, a semi-formal notation, the Unified Modeling Language (UML) [6], that lacks some of the formal rigor of the notations considered in the previous section, has quickly become a de-facto standard in the industrial software development process. UML recent success is mainly due to the following reasons [1]:

- It allows to embed into the model static and dynamic aspects of the software by using different diagrams, each representing a different view of the software system. Each view captures a different set of concerns and aspects regarding the subject. Therefore it is broadly applicable to different types of domains or subject areas.
- The same conceptual framework and the same notation can be used from specification through design to implementation.
- In UML, more than in classical object oriented approaches, the boundaries between analysis, design and implementation are not clearly stated. As a consequence, there is more freedom in software development process, even if the Rational Unified Process [19] has been proposed as a guideline for software process development based on UML.
- UML is not a proprietary and closed language but an open and fully extensible language. The extensibility mechanisms and the potential for annotations of UML allow it to be customized and tailored to particular system types, domains, and methods/processes. It can be extended to include constructs for working within a particular context (e.g., performance requirement validation) where even very specialized knowledge can be captured.
- It is widely supported by a broad set of tools. Various tool vendors intend to support UML in order to facilitate its application throughout an organization. By having a set of tools that support UML, knowledge may be more readily captured and manipulated to meet an organization's objectives.

UML consists of two parts: a *notation*, used to describe a set of diagrams (also called the syntax of the language) and a *metamodel* (also called the semantics of the language) that specifies the abstract integrated semantics of UML modeling concepts. The UML notation encompasses several kinds of diagrams, most of them belonging to previous methodologies, that provide specific views of the system. UML diagrams can be distinguished into four main types:

1. Static diagrams: Use Case, Class and Object Diagrams
2. Behavioral diagrams: Activity and State Diagrams
3. Interaction diagrams: Sequence and Collaboration Diagrams

4. Implementation diagrams: Component and Deployment Diagrams

“Standard” UML as ON for mobile architecture. Standard UML can be used as ON for mobile architectures, since UML already provides some mechanisms for this goal. They are mainly based on the use of a tagged value `location` within a component to express its location, and of the `copy` and `become` stereotypes to express the location change of a component. The former stereotype can be used to specify the creation of an independent component copy at a new location (like in the COD and REX styles), and the latter to specify a location change of a component that preserves its identity (like in the MA style). In [6] it is shown how to use these mechanisms within a Collaboration Diagram to model the location change of a mobile component interleaved with interactions among components.

Example 5. The travel agency application can be modeled by the following Collaboration Diagram, based on standard UML, in case of MA paradigm.

EndOfExample5

However, this modelling approach presents some drawbacks, since it mixes together two different views, one concerning the architectural style (e.g. the fact that a component behaves according to some mobility style), and the other one concerning the actual sequence of messages exchanged between components during a particular interaction. Moreover, this approach may lead to a proliferation of objects in the diagrams, that actually represent the same object at different locations. Both these drawbacks can lead to quite obscure models of the application behavior.

“Extended” UML as ON for mobile architecture. To overcome the drawbacks of standard UML as ON for mobile architectures, the dependency between the modeled AS and the chosen ON can be made explicit by adopting a different approach based on the use of both Collaboration and Sequence Diagrams (CD and SD), with a clear separation of concerns between them, as proposed in [15]. The SD describes the actual sequence of interactions between components, which is basically independent of the adopted style and obeys only to the intrinsic logic of the application, while the CD only models the interaction structure (i.e. who interacts with whom) and style, without showing the actual sequence of exchanged messages.

The interaction logic is described using the standard UML notation for SD. The interaction structure is modeled by the links that connect components in CD, with arrows specifying unidirectional or bidirectional interactions. For the interaction style, the main goal is to distinguish a style where components location is statically assigned, from a style where components do change location to adapt to environment

change. To this purpose, the standard `location` tagged value can be used to specify the component location, while it is necessary to extend the UML semantics by introducing a new stereotype `moveTo` that applies to messages in the CD. Where present, `moveTo` indicates that the source component moves to the location of its target before starting a sequence of consecutive interactions with it. If no other information is present, this style applies to each sequence of interactions shown in the associated SD, between the source and target components of the `moveTo` message; otherwise a condition can be added to restrict this style to a subset of interactions between two components. It should be noted that this approach appears suitable to model only mobile architectures where the architecture style is $AS=\{MA\}$.

Example 6 According to the adopted modeling framework, the travel agency example application can be modeled as shown in figure 2.

Figure 2. Travel agency example: (a) interaction logic, (b) architectural style

Figure 2.a shows a SD that describes in detail the “logic” of the interaction, i.e. the sequence of messages exchanged among the components. In this diagram no information is present about the adopted style, that is whether or not some component changes location during the interactions. This information is provided by the CD in figure 2.b, that models a style where component mobility is considered. More precisely, the diagram shows that only `c` can change location, and according to the `moveTo` semantics described above, it moves to the location of `a`, `f1` or `f2` before interacting with them. Note that in figure 2.b the location of `c` is left unspecified ($L?$), since it can dynamically change. In general, it is possible to give it a specified value in the diagram, that would show the “initial” location of the mobile object in an initial deployment configuration. *EndOfExample6*

In general, there could be uncertainty about the convenience of adopting a mobile code style in the design of an application. To model this uncertainty about the architecture (i.e. location and possible mobility of components), a new stereotype `moveTo?` has been proposed in [15], that extends the semantics of the `moveTo` stereotype described above. When a message between two components in a CD is labeled with `moveTo?`, this means that the source component “could” move to the location of its target at the beginning of a sequence of interactions with it. In a sense, this means that, based on the information the designer has at that stage, he/she considers acceptable both a static and a mobile architecture. Hence, a general UML support to model a (possibly) mobile architectural style consists of a CD where some messages are unlabeled, some can be labeled with the (possibly constrained) `moveTo` stereotype, and some with the `moveTo?` stereotype. The former two cases correspond to a situation where the designer feels confident enough to decide about the best architectural style, while the latter to a situation where the designer lacks such a

confidence. In the next section, we illustrate methodologies for the translation from a model defined using the extended UML as ON, to suitable TMNs for the analysis of different NFAs.

From extended UML models to performance models. The goal is to build a stochastic model that describes the system dynamics, whose evaluation provides information about the performance one can expect by adopting an MA or “static” architectural style, as described in the previous section. In terms of the extended UML presented above, the gained insights should allow the designer to substitute the `moveTo?` messages in the preliminary CD with (possibly constrained) `moveTo` messages, or with no such message at all, if the obtained insights provide evidence that a static architectural style is more advantageous. Specifically two different methodologies have been proposed [15, 16] that, starting from a set of annotated UML diagrams derive two different TMNs, namely, a Markov model or a queueing network model. In the following we briefly present these two methodologies.

The first one is suitable for cases when the NFAs of interest are mainly interaction-related measures (e.g., generated network traffic) without considering possible contention with other applications. In this approach, the TMN is a Markov Reward Process (MRP) [28] when the CDs modeling the architectural style only use the `moveTo` stereotype, and a Markov Decision Process (MDP) [28] when the CDs modeling the architectural style also use the `moveTo?` stereotype.

The second one, based on classic SPE technique [32, 33], is suitable for cases where the NFAs of interest are measures like throughput or response time and we are possibly interested in considering contention with other applications on the use of system resources. Two different TMNs are taken into account, namely, Execution Graphs [32] and Extended Queueing Network Models [20] for NFAs with and without consideration to the impact of contention, respectively.

In both cases, it is assumed that the diagrams described in the previous section are augmented with appropriate annotations expressing the “cost” of each interaction with respect to a given performance measure (see, for example, [31] and papers in [39]), to represent the needed MI. For example, if we are interested in the generated network traffic, MI includes at least the size of each exchanged message.

Markov models. In general, a MRP models a state transition system, where the next state is selected according to a transition probability that only depends on the current state. Moreover, each time a state is visited or a transition occurs, a reward is accumulated, that depends on the involved state or transition. Typical measures that can be derived from such a model are the reward accumulated in a given time interval, or the reward accumulation rate in the long period. A MDP extends the MRP model by associating to each state a set of alternative *decisions*, where both the rewards and the transitions associated to that state are decision dependent. A *policy* for a MDP consists in a selection, for each state, of one of the associated decisions, that will be taken each time that state is visited. Hence, different policies lead to different system behaviors and to different accumulated rewards. In other words, a MDP defines a family of MRPs, one for each different policy that can be determined. Algorithms exist to determine the optimal policy with respect to some optimality criterion (e.g. minimization of the accumulated reward) [28].

In the translation methodology adopted in [15], a MRP/MDP state corresponds to a possible configuration of the components location, while a state transition models the occurrence of an interaction between components or a location change, and the

associated reward is the cost of that interaction. In case of MDP, the decisions associated to states model the alternative choices of mobility or no mobility as architectural style, for those components that are the source of a `moveTo?` message.

The translation method from the extended UML to this TMN consists of the definition of some elementary generation rules, and then in the use of these rules to define a MDP generation algorithm [15].

Once the MDP has been generated, it can be solved to determine the optimal policy, that is the selection of a decision in each state that optimizes the reward accumulated in the corresponding MRP. Of course, the optimal policy depends on the values given to the system parameters (e.g., the size of the messages and of the possibly mobile component). Different values for these parameters model different scenarios.

Queueing Network models. A different methodology for the derivation of performance models from extended UML diagrams has been proposed in [16], based on SPE techniques, having queueing network models as basic TMN.

The SPE basic concept is the separation of the software model (SM) from its execution environment model (i.e., hardware platform model or machinery model, MM). The SM captures the essential aspects of software behavior; and is usually represented by means of Execution Graphs (EG). An EG is a graph whose nodes represent software workload components and whose edges represent transfers of control. Each node is weighted by a demand vector that represents the resource usage of the node (i.e., the demand for each resource).

The MM models the hardware platform and is based on the Extended Queueing Network Model (EQNM). To specify an EQNM, we need to define: the components (i.e., service centers), the topology (i.e., the connections among centers) and some relevant parameters (such as job classes, job routing among centers, scheduling discipline at service centers, service demand at service centers). Component and topology specification is performed according to the system description, while parameters specification is obtained from information derived by EGs and from knowledge of resource capabilities. Once the EQNM is completely specified, it can be analyzed by use of classical solution techniques (simulation, analytical technique, hybrid simulation) to obtain performance indices such as the mean network response time or the utilization index.

To cope with mobility, in the methodology proposed in [16], well-known formalisms such as EG and EQNM have been extended by defining the *mob?*-EG and *mob?*-EQNM formalisms with the goal of modelling code mobility and the uncertainty about its possible adoption, within a model of the system dynamics.

To include the information about possible component mobility expressed in the CDs by `moveTo?` messages, a new kind of EG called *mob?*-EG is derived [16]. The *mob?*-EG modifies the original EG by introducing *mv* nodes that model the cost of code mobility. Moreover, the *mob?*-EG extends the EG formalism by introducing a new kind of node, called *mob?*, characterized by two different outcomes, “yes” and “no”, that can be non-deterministically selected, followed by two possible EGs. The EG corresponding to branch “yes” models the selection of component mobility style, while the EG of the branch “no” models the static case.

Example 7. The structure (without labels showing performance related information) of the *mob?*-EG derived from the SD and CD of example 6 is illustrated in the following figure.

EndOfExample7

Mob?-EG can be considered by itself as the TMN for a first kind of performance evaluation corresponding to the special case of a stand-alone application where the application under study is the unique in the execution environment (therefore there is no resource contention). In this case performance evaluation can be carried out by standard graph analysis techniques [32] to associate an overall “cost” to each path in the *mob?*-EG as a function of the cost of each node that belongs to that path. Note that each path in the *mob?*-EG corresponds to a different mobility strategy, concerning when and where components move. Hence these results provide an optimal bound on the expected performance for each strategy, and can help the designer in selecting a subset of the possible mobility strategies that deserve further investigation in a more realistic setting of competition with other applications.

The complete application of SPE techniques implies the definition of a target performance model obtained from the merging of the *mob?*-EG with a QN modeling the executing platform. The merging leads to the complete specification of a EQNM by defining job classes and routing, using information from the blocks and parameters of the *mob?*-EG. However, well known translation methodologies [10, 32] are not sufficient to perform this merging because of the presence of the *mob?* nodes with non-deterministic semantics in the *mob?*-EG; hence it is necessary to give a new translation rule to cope with this kind of nodes. To this end an extension of classical EQNMs has been proposed [16], to be used as TMN when the ON is the extended UML defined in the previous section. The extension is based on the definition of new service centers, called *r?(outing)*, that model the possibility, after the visit of a service center (and therefore the completion of a software block) to choose, in a non-deterministic way, which is the routing to follow: the one modelling the static strategy or the one modelling the mobile strategy.

In such a way, a job visiting center $r?$ generates two different mutually exclusive paths: one path models the job routing when the component changes its location, the other one models the routing of a static component. Note that, as node $mob?$ in the EG, nodes $r?$ are characterized by a null service time, since they only represent a routing selection point. The obtained model is called $mob?$ -EQNM and is characterized by different routing chains starting from nodes $r?$. Note that these different routing chains

are mutually exclusive; in other words a *mob?*-EQNM actually models a family of EQNMs, one for each different path through the $r?$ nodes, corresponding to a different mobility policy.

Example 8. The following figure illustrates an example of *mob?*-EQNM derived from the *mob?*-EG of example 7, exploiting also information about the execution platform (e.g., obtained from a UML Deployment Diagram). The figure evidences the mutually exclusive routing chains.

EndOfExample8

When the *mob?*-EQNM is the TMN, the ST suggested in [16] for contention based analysis is based on solving the *mob?*-EQNM through well assessed techniques [20, 32], separately considering each different EQNM belonging to the family modeled by the *mob?*-EQNM. When the number of different EQNMs is high, this solution approach could result in a high computation complexity. This problem can be alleviated by exploiting results from the stand-alone analysis. However, more efficient solution methods deserve further investigation. Starting from the obtained results it is possible to choose the mobility strategy which is *optimal* according to the selected criterion, for example the one that minimizes the response time.

5 Conclusions

The primary goal of this tutorial has been to provide a structured view within the domain of performance validation of mobile software architectures. The classification of the approaches presented in section 4 has been supported by a general framework (section 2) classifying the parameters each approach has to deal with. In table 5 we summarize the values (in some cases the classes of values) assumed by the parameters introduced in section 2 in all the types of approaches reviewed in section 4.

Besides ad-hoc models, whose merits and limitations have been outlined in section 4.1, two kinds of approaches for the systematic modeling and analysis of NFA in mobile software architectures emerge from our review, based on the use of formal or semi-formal languages as ON. We would like to remark here that our review is probably not complete, but we believe it is representative of existing approaches.

The merit of formal languages comes primarily from their lack of ambiguity, and their precise compositional features. However, as it can also be inferred from table 5, their use in NFA validation requires the assignment of reward and exponential duration to all the modelled actions, that could be quite a difficult task. The method presented in

section 4.2 provides guidelines about how building meaningful rates, but leaves open the problem of how collecting the required detailed information. One way to overcome this drawback could be based on bridging the gap between process algebras and other formalisms (like UML) used by software designers, to facilitate the extraction of the detailed information required by process algebras from artifacts produced by the designers.

On the other hand, the use of UML as ON from which to derive performance models is not immune from problems as well, so the general problem of deriving meaningful performance models from UML artifacts deserves further investigation by itself. Also UML modeling of mobile software architectures still appears not completely satisfactory, because of the lack of widely accepted models for all the mobile code styles reviewed in section 3.

Table 5. Summary of parameters instantiation in the reviewed approaches to non functional requirements validation of mobile software architectures

O N		AS	NFA	TMN	MI	ST
none		REX, COD, MA	processing time, network load	closed-form	model parameters	analytic
		REX, MA	processing time	Petri net		numerical
Formal (P.A.)	“indirect location”	REX, COD, MA	“any”	stochastic P. A. and MRP	transition and reward rates	mainly numerical
	“direct location”					
Semi- Formal (UML)	standard					
	with mobility- oriented extensions	MA	interaction related (stand-alone applications)	MRP, MDP	performance annotations in the SDs	mainly numerical
			throughput, response time (stand-alone and contention-based measures)	Execution Graph, EQNM	performance annotations in the UML diagrams	analytic, numerical, simulation

Besides the primary goal of reviewing non functional requirements validation approaches for mobile software architectures, another goal of this tutorial has been to promote the identification of instances of the proposed general validation framework in other areas, such as reliability validation of real-time systems, just to name one. We do not claim this classification as being “the right one”, but with this tutorial we rather would like to solicit feedback, extensions and new instances to come out.

Everybody involved in modern (distributed, embedded, mobile) software systems design, is concerned about “software quality”, but still very few of them take into account the actual possibility of introducing methodologies/techniques/tools to systematically improve this attribute. Therefore we believe that in the area of non-functional requirements validation much work should be done in the direction of schematization to make it an acceptable activity from the software designer side.

Acknowledgements

Work partially supported by MURST project “SAHARA: Software Architectures for heterogeneous access network infrastructures”.

Bibliography

1. S. Alhir, "The true value of the Unified Modeling Language", *Distributed Computing*, 29-31, July 1998.
2. M. Baldi, G.P. Picco "Evaluating the tradeoffs of mobile code design paradigms in network management applications" in Proc. *20th Int. Conf. on Software Engineering (ICSE 98)*, (R. Kemmerer and K. Futatsugi eds.), Kyoto, Japan, Apr. 1998.
3. S. Balsamo, M. Simeoni "Deriving Performance Models from Software Architecture Specifications" *Res. Rep. CS-2001-04*, Dip. di Informatica, Università di Venezia, Feb. 2001; *ESM 2001, SCS, European Simulation Muticonference 2001*, Prague, 6-9 June 2001.
4. M. Barbeau "Transfer of mobile agents using multicast: why and how to do it on wireless mobile networks" *Tech. Rep. TR-00-05*, School of Computer Science, Carleton University, July 2000.
5. L. Bass, P. Clements, R. Kazman, *Software Architectures in Practice*, Addison-Wesley, New York, NY, 1998.
6. G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, Addison Wesley, New York, 1999.
7. L. Cardelli, A.D. Gordon "Mobile ambients" *Foundations of Software Science and Computational Structures* (M. Nivat ed.), LNCS 1378, Springer-Verlag, 1998, pp. 140-155.
8. N. Carriero, D. Gelernter "Linda in context" *Communications of the ACM*, vol. 32, no.4, 1989, pp. 444-458.
9. T.-H. Chia, S. Kannapan "Strategically mobile agents" in Proc. *1st Int. Conf. on Mobile Agents (MA '97)*, Springer-Verlag, 1997.
10. V. Cortellessa, R. Mirandola "PRIMA-UML: a performance validation incremental methodology on early UML diagrams" *Science of Computer Programming*, Elsevier Science, vol 44, n.1, pp 101-129, July 2002.
11. R. De Nicola, G. Ferrari, R. Pugliese, B. Venneri "KLAIM: a kernel language for agents interaction and mobility" *IEEE Trans. on Software Engineering*, vol. 24, no. 5, May 1998, pp. 315-330.
12. G. Ferrari, C. Montangero, L. Semini, S. Semprini "Mobile agents coordination in Mob_{adl}" Proc. of *4th Int. Conf. on Coordination Models and Languages (COORDINATION'00)*, (A. Porto and G.-C. Roman eds.), Springer-Verlag, Limassol, Cyprus, Sept. 2000.
13. A. Fuggetta, G.P. Picco, G. Vigna "Understanding code mobility" *IEEE Trans. on Software Engineering*, vol. 24, no. 5, May 1998, pp. 342-361.
14. N. Gotz, U. Herzog, M. Rettelbach "Multiprocessor system design: the integration of functional specification and performance analysis using stochastic process algebras" in *Performance Evaluation of Computer and Communication Systems* (L. Donatiello and R. Nelson eds.), LNCS 729, Springer-Verlag, 1993.
15. V. Grassi, R. Mirandola, "Modeling and performance analysis of mobile software architectures in a UML framework" in *<<UML2001>> Conference Proceedings*, LNCS 2185, Springer Verlag, October 2001.
16. V. Grassi, R. Mirandola, "PRIMAmob-UML: a Methodology for Performance analysis of Mobile Software Architecture", in *WOSP 2002, Third International Conference on Software and Performance*, ACM, July 2002.
17. R. Gray, D. Kotz, G. Cybenko, D. Rus "Mobile agents: motivations and state-of-the-art systems" in *Handbook of Agent Technology*, AAAI/MIT Press, 2001.
18. H. Hermanns, U. Herzog, J.-P. Katoen "Process algebras for performance evaluation", *Theoretical Computer Science*, vol. 274, no. 1-2, 2002, pp. 43-87.
19. I. Jacobson, G. Booch, J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley Object Technology Series, 1999.
20. R. Jain, *Art of Computer Systems Performance Analysis*, Wiley, New York, 1990.

21. T. Kawamura, S. Joseph, A. Ohsuga, S. Honiden "Quantitative evaluation of pairwise interactions between agents" in *Joint Symp. on Agent Systems and Applications and Symp. on Mobile Agents (ASA/MA 2000)*, Springer LNCS 1882, pp. 192-205, 2000.
22. D. Kotz, G. Jiang, R. Gray, G. Cybenko, R.A. Peterson "Performance analysis of mobile agents for filtering data streams on wireless networks" in *3rd ACM Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM 2000)*, Aug. 2000.
23. R. Milner, *Communication and Concurrency*, Prentice Hall, 1989.
24. R. Milner, *Communicating and Mobile Systems: the π -calculus*, Cambridge Univ. Press, 1999.
25. C. Nottegar, C. Priami, P. Degano "Performance evaluation of mobile processes via abstract machines" *IEEE Trans. on Software Engineering*, vol. 27, no. 10, Oct. 2001, pp. 867-889
26. G.P. Picco, G.-C. Roman, P.J. McCann "Reasoning about code mobility in Mobile UNITY" *ACM Transactions on Software Engineering and Methodology*, vol. 10, no. 3, July 2001, pp. 338-395.
27. A. Puliafito, S. Riccobene, M. Scarpa "An analytical comparison of the client-server, remote evaluation and mobile agent paradigms" in *1st Int. Symp. on Agent Systems and Applications and 3rd Int. Symp. on Mobile Agents (ASA/MA 99)*, Oct. 1999.
28. M.L. Puterman, *Markov Decision Processes*, J. Wiley and Sons, 1994.
29. K. Rothermel, F. Hohl, N. Radouniklis "Mobile agent systems: what is missing?" in *Distributed Applications and Interoperable Systems (DAIS 1997)*, 1997.
30. D. Sangiorgi "Expressing mobility in process algebras: first-order and higher-order paradigms" PhD thesis, Univ. of Edinburgh, 1992.
31. B. Selic, "Response to the omg rfp for schedulability, performance and time", OMG document number ad/2001-06-14, June 2001.
32. C. Smith, *Performance Engineering of Software Systems*, Addison-Wesley, Reading, MA, 1990.
33. C. Smith, L. Williams, *Performance solutions: A Practical Guide to Creating Responsive, Scalable Software*, Addison Wesley, 2002.
34. J.W. Stamos, D.K. Gifford "Implementing remote evaluation" *IEEE Trans. on Software Engineering*, vol. 16, no. 7, July 1990, pp. 710-722.
35. M. Strasser, M. Schwehm "A performance model for mobile agent systems" in *Int. Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 97)*, vol. II, (H.R. Arabnia ed.), Las Vegas 1997, pp. 1132-1140.
36. Sun Microsystems "The Java language", White Paper, 1994.
37. Sun Microsystems "The Java servlet API", White Paper, 1997.
38. M. Wermelinger, J.L. Fiadeiro "Connectors for mobile programs" *IEEE Trans. on Software Engineering*, vol. 24, no. 5, May 1998, pp. 331-341.
39. WOSP 2000, Proc. of the *2nd Int. Workshop on Software and Performance*, ACM, 2000.