Decision Support for Mobile Cloud Computing Applications via Model Checking

Luca Aceto
Reykjavik University, Iceland

Andrea Morichetta, Francesco Tiezzi
IMT Institute for Advanced Studies Lucca, Italy

***Abstract*—Deciding whether to offload some computation is a crucial issue in Mobile Cloud Computing systems. Various proposals in the literature try to face this problem taking decisions at design time, relying on the developer's expectations or basing the choice on resource usage prediction. This paper proposes a new methodology based on a formal framework, whose goal is to provide runtime support for offloading decisions. By means of a domain specific language (MobiCa), a developer can define both system and application structure, in terms of involved devices, computational power, code partitioning, required memory etc. Using a formal verification technique, based on the well-known model checker UPPAAL, the framework provides exhaustive and automatic decision support for offloading decisions. Using the diagnostic trace generated at runtime by UPPAAL, driven by some query verified on the timed automata model associated with the MobiCa specification, the framework decides which application fragments should be remotely executed. This process allows one to reduce memory and battery consumption, ensuring at the same time the best system performance. The proposed approach is exemplified on a navigator case study, probably one of the most used applications on mobile devices.**

I. Introduction

In the last decade, portable devices have increasingly pervaded our daily lives. Innovation in hardware has continuously provided powerful computational resources in lighter and smaller electronic handsets. This hardware evolution has triggered a tremendous amount of new mobile applications; in a few years the mobile application market has exploded [1]. The success of this market is strictly related to the ubiquitous access to data and computation that should satisfy the needs of users everywhere and at any time.

The majority of mobile applications that run on a portable device are strongly connected to the network for accessing data, but maintain the computational load locally. This characteristic requires a large amount of energy in battery-limited devices. For this reason there exist wired infrastructures, such as workstations, servers and, in particular, cloud systems, that provide more computational resources without being limited by stringent size, weight and energy consumption. A combination of mobile devices and cloud infrastructures can provide computational resources everywhere and at any time in mobile devices. This is at the basis of an emerging paradigm, called *Mobile Cloud Computing* (MCC) [8]–[10], for developing mobile applications. It relies on the *offloading* concept, i.e. the capability of moving the computational power and data storage away from mobile devices into the cloud.

There are many reasons [4] for executing part of the computation on remote infrastructures in addition to the mobile device on which it is currently carried out. The first obvious potential benefit is improving storage capacity and performance. Indeed, even if the computational resources available to a mobile device have increased rapidly over the past few years, this computing power is still smaller than that of the stationary counterparts. This is because smartphones and tablets must be much smaller and lighter than servers and desktop computers. Due to this gap between mobile and cloud processing power, CPU intensive applications can be executed much faster on remote than on mobile devices. On the other hand, interactive applications that require few computational resources may be executed on mobile devices as fast as on servers. Performance does not depend only on processor speed, but also on memory, storage, the ability of parallelizing across multiple cores and servers, and better connections. All of these characteristics can be simply provided by a common cloud infrastructure with a very small investment.

The second benefit is reducing energy consumption. According to [13], mobile device users found longer battery life to be more important than all other features. A mobile device should budget its finite source of energy wisely in order to arrive to the end of the day without exhausting all its available energy before. Designers are involved in an arduous challenge to make devices that are as energy efficient as possible, for example using hardware power-saving modes or reducing the speed and quality of activities performed by mobile applications. While these measures are essential for extending battery lifetime, they also noticeably degrade the mobile users' experience due to the slowing down of the execution of applications. Computation offloading is clearly an attractive alternative, because using remote computation and storage improves performance and, at the same time, saves battery power giving a better user experience.

Unfortunately the use of a remote infrastructure does not always come without a cost. Sometimes computation offloading may degrade the application's performance. A possible way for measuring performance is to asses if the time saved by performing the computation remotely is greater than the time spent communicating inputs and outputs over the network against the total computation executed locally. This measure, of course, could be highly influenced by network latency, bandwidth or computational power requirements of the offloaded code. For example, with high latency, low bandwidth and a consistent amount of offloaded code with a very low computational power requirement, executing the computation locally can improve the performance rate. The issues discussed above for performance apply also to energy saving. Indeed, code offloading through a power hungry wireless network may

require more energy than performing the same activity locally. According to the study proposed in [14], a critical aspect in energy saving is the necessity of finding at runtime, from time to time, the best trade-off between energy consumed by computation and energy consumed by communication.

In this paper, we face the challenges mentioned above by defining a new methodology for developing MCC applications. It is based on a formal framework devised to design the system structure and to provide runtime support to offloading decisions, with the aim of finding the desired trade-off between energy consumption and performance. More specifically, we propose a domain-specific language, called MobiCa, for designing MCC systems in terms of installed applications, code partitioning, and device characteristics (computational power, memory, network bandwidth, etc.). The main advantage of a domain-specific language, with respect to general-purpose modelling languages, is to allow system designers to focus on those aspects of the mobile applications that are relevant for the MCC paradigm and, in particular, for taking offloading decisions. These aspects are indeed first-class citizens of MobiCa, thus providing a high-level modelling perspective that abstracts from low-level details (e.g., the specific computations performed by the application components) that are not necessary for the offloading decision support.

The proposed methodology is as follows: (1) the developer of the system designs it using MobiCa, by focusing on the MCC characteristics of the application(s) of interest; (2) the developer implements the functionalities of the application components defined in the MobiCa specification; (3) the MobiCa specification is used to automatically generate the runtime decision support system to be used by devices to appropriately execute the MCC applications.

In particular, in this paper we define the domain specific language at point 1 (Section II), and we provide a framework, based on the formal tool UPPAAL[1], for the decision support at point 3 (Section III). Point 2 is instead out of the scope of this paper; we expect that the MobiCa specification could be used to automatically generate the skeleton implementation of the applications, which would be then filled by the developer.

The decision support system consists of a timed automata model [2] enabling performance analysis via UPPAAL. We have defined and implemented[2] a semantics for MobiCa via translation into timed automata, which automatically associates an UPPAAL model to each MobiCa specification. This takes a weight off the developer, who does not have to take care of the details specified in the timed automata model. The main novelty of this approach, with respect to the related work in the literature, is the use of a formal verification technique based on model checking for offloading decisions. It provides an exhaustive and automatic evaluation of the system configuration by means of the well-established, reliable and efficient model checker UPPAAL [5]. In particular, the verification of given queries is used to generate diagnostic traces corresponding to the best offloading strategies for the current configuration of the system. The implementation of the MobiCa-to-UPPAAL transformation relies on the software framework Xtext[3].

We illustrate the MobiCa approach throughout the paper by means of a case study concerning one of the most used applications in mobile devices: the navigator. The greatest challenge for navigation system developers is to provide an application that is able to find the right route, and recalculate it as quickly as possible in case of changes. These tasks should possibly consider the current traffic condition.

Due to these strong computational requirements, the navigation application is an interesting case study to analyze under the MCC paradigm. A common behavioural schema for a navigation application is depicted in Figure 1. The user sets the destination in the *configuration panel*, which activates the *controller*, whose first job is to check the current *GPS* position and pass it to the *path calculator*. This latter component interacts with the *maps* and the *traffic evaluator* systems to provide the best itinerary. The itinerary is processed by the *navigator*, which provides the right information to the *navigation panel* and in parallel reactivates the *controller* for checking possible updates to the route. As last step, the *navigation panel* activates *voice* and *speed trap indicator*.

Structure of the paper. The rest of the paper is organized as follows. In Section II we introduce the MobiCa language, with his syntax and characteristics. In Section III we describe the semantics of the MobiCa language given by translation into Timed Automata. In Section IV we describe the tool automating the translation and present the UPPAAL analysis applied to the navigator case study. In Section V we survey the background on MCC and review the related research. We conclude our paper in Section VI with some remarks and a discussion of avenues for future work.

II. The MobiCa Language

In this section we introduce the domain-specific language MobiCa (*Mobile Cloud Computing Language*), specifically devised for modelling MCC systems at a high-level of abstraction. Such abstraction allows one to focus on those aspects of MCC systems that are relevant for taking decisions on whether to offload a given component of a mobile application. Low-level details, such as the precise computation performed by the component, are abstracted. In this way, we have a modeling language that permits easily writing compact specifications, which enable an efficient analysis for selecting the most appropriate offloading strategy. In fact, the semantics of the language associates to each specification a model, specifically a network of timed automata (see Section III), supporting the verification of properties about offloading through a model checking technique.

The building blocks of MCC applications expressed in MobiCa are black-box elements called *fragments*. Each fragment corresponds to a small part of an application devoted to a unique specific purpose. The language permits to model applications partitioned in fragments with different granularity, which depends on the level of detail required by the considered application domain. In coarse-grained partitions, fragments represent components offering functionalities and services to

[1] http://www.uppaal.org

[2] The software tool implementing the MobiCa semantics, together with an Eclipse plugin for editing MobiCa specifications, can be downloaded from http://sysma.imtlucca.it/mobica/.

[3] http://www.eclipse.org/Xtext/

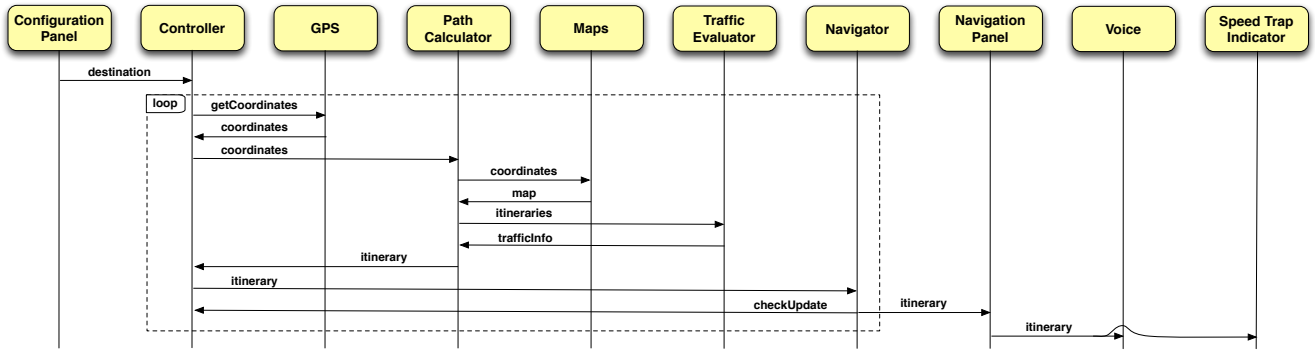

Fig. 1. Navigation case study: sequence diagram

SYSTEMS:	N	::=	$M \mid C \mid N_1 \mid N_2$
MOBILE DEVICES:	M	::=	$(c, b, n, m) \triangleright \tilde{A}$
CLOUD MACHINES:	C	::=	$c \triangleright \tilde{A}$
APPLICATIONS:	A	::=	$\langle \tilde{F}; S \rangle$
FRAGMENTS:	F	::=	$f[i, m, s, o]$
STRUCTURE:	S	::=	$f_1 \; Op \; \tilde{f}_2 \mid S_1 \; ; \; S_2$
OPERATORS:	Op	::=	$\longrightarrow \mid \dashrightarrow \mid \twoheadrightarrow$

TABLE I. MOBICA SYNTAX

each other; instead, in the fine-grained ones, fragments correspond to single functionalities, tasks or actions. Moreover, although MobiCa also permits modelling MCC applications having a static code separation between local and remote execution, it is designed to model more flexible and dynamic systems, where the computation of any fragment can in principle be offloaded. The offloading decisions are taken according to the current contextual conditions, e.g. network connection, bandwidth, servers' load, battery energy, etc. In fact, due to the variability of these factors, static partitioning leads in general to poor performance optimization.

The syntax of the MobiCa language is defined by the BNF grammar given in Table I. As a matter of notation, we use $\tilde{\cdot}$ to denote tuples of objects, e.g. $\tilde{F}$ stands for the tuple of fragments $F_1, \ldots, F_n$ (with $n > 0$).

A MobiCa specification is a *system* N consisting of a network of mobile devices M and cloud machines C (composed by means of the parallel operator $|$).

A *mobile device* corresponds to any portable device that is limited in computational capabilities, and without a stable connection and a constant power supply. Typical examples are smartphones, tablets, etc. Specifically, a mobile device $(c, b, n, m) \triangleright \tilde{A}$ can be seen as a container of applications $\tilde{A}$ characterized by some operational information: the computational power of the device c (expressed in terms of number of instructions that can be executed per second), the battery level b, the network bandwidth n and the used memory m. All such operational information is defined as integer values and indicates the status of the device at a given instant of time. It is worth noticing that $\tilde{A}$ represents the MCC applications installed in the device that are subject to dynamic offload handling. Non-MCC applications are not explicitly represented, although their effects on resources of the device are taken into account when the operational information is determined.

A *cloud machine*, differently from a mobile device, has a larger computation and memory capability, and a stable connection. Thus, for what concerns the aspects of interest for our analysis, a cloud machine $c \triangleright \tilde{A}$ only specifies the number c of instructions executed per second, and the installed applications $\tilde{A}$.

An *application* is represented as a pair $\langle \tilde{F}; S \rangle$. The first field is a tuple of fragments, while the second describes how they are connected to each other in order to form the structure of the application. The first fragment of the tuple $\tilde{F}$ is the initial fragment of the structure (i.e., it is the root of the corresponding graph). A *fragment* $f[i, m, s, o]$ is uniquely identified by a name f and specifies the following parameters: the number i of instructions to execute, the amount m of memory required at runtime, the amount s of data to be transferred for synchronization in case of offloading, and finally a boolean o indicating whether the fragment is offloadable or not. Notably, the first three parameters should be thought of as average values, which could be estimated from the code or, more practically, determined at runtime by monitoring the application execution. Notice also that, in case of offloading, it is necessary to synchronize local and remote application instances to keep the state of the system consistent.

A *structure* S is a graph whose nodes are fragments and whose edges represent their interactions. A structure is specified in MobiCa by a collection of terms of the form $f_1 \; Op \; \tilde{f}_2$, each one defining a set of edges (from f_1 to each fragment in $\tilde{f}_2$). There are three kinds of edges, corresponding to three different ways to proceed with the application execution from one fragment to further fragments according to three different synchronization operators Op:

- *Non-deterministic choice* ($\longrightarrow$) indicates that the execution will progress from the source fragment to *one* of the target fragments, which is non-deterministically selected. This operator allows one to abstract from choices that are internal to fragments.
- *Sequential progress* ($\dashrightarrow$) permits the computation to sequentially progress from the source fragment to the target ones (following the order in the tuple $\tilde{f}_2$). This operator is used to make explicit the relationship between a fragment and the others necessary to carry out its task. This allows one to partially 'shed light' on the black-box nature of fragments.
- *Parallel execution* ($\twoheadrightarrow$) permits the execution to progress from the source fragment to *all* target ones, by activating their parallel execution. This operator

allows one to express applications with concurrent components.

If more groups of edges have the same source, the execution proceeds from this source fragment by selecting a group in a non-deterministic way, and then by activating the target fragments of this group according to the corresponding operator. These two steps are performed atomically. For example, given the following structure term $f_1 \multimap\!\!\!\!\rightarrow f_2, f_3;\ f_1 \dashrightarrow f_4, f_5$, it is possible to proceed from f_1 either by activating f_2 and f_3 in parallel, or by firstly activating f_4 and then, once f_1 is activated again, by activating f_5. It is also worth noticing that a fragment may have more than one incoming edge; each time an activation signal arrives from one of them, the fragment is activated, if it is not already activated, otherwise this activation is postponed (in other words, concurrent activations of the same fragment are not allowed).

Not all applications allowed by the syntax in Table I are meaningful. We only consider terms satisfying well-formedness conditions, i.e. syntactic constraints that can be easily verified with a static check on the syntax of terms.

Definition 1 (Well-formed applications): An application $\langle \tilde{F}; S \rangle$ is well formed if the following conditions hold: *(i)* all fragment names occurring in S are defined in the tuple $\tilde{F}$; and *(ii)* there is no term $f_1\ Op\ \tilde{f}_2$ in S such that f_1 occurs in the tuple $\tilde{f}_2$ (i.e., self-loops are disallowed).

We conclude this section by showing how the navigation application informally described in the Introduction can be specified in MobiCa. In particular, due to the nature of this application, we can consider a simple scenario involving one mobile device and one cloud machine:

$$(16, 100, 2, 0) \triangleright \mathit{NavigationApp} \ \mid\ 32 \triangleright \mathit{NavigationApp}$$

In the above configuration, the cloud machine has twice the computational power than the mobile device (32 vs. 16 instructions per second). The battery of the mobile device is completely charged (the level is expressed in percentage), the network bandwidth is low (2 Mb/s), and all memory dedicated to apps execution is available (the used one is 0).

The application $\mathit{NavigationApp}$ is defined as follows:

$$\begin{array}{l}\langle\ \mathit{configPanel}[11, 1, 0, \mathit{false}], \mathit{controller}[15, 20, 55, \mathit{true}],\\ \mathit{pathCalc}[90, 50, 30, \mathit{true}], \mathit{traffEval}[99, 22, 22, \mathit{true}],\\ \mathit{map}[25, 200, 100, \mathit{true}], \mathit{navigator}[110, 10, 15, \mathit{true}],\\ \mathit{navPanel}[11, 30, 45, \mathit{true}], \mathit{voice}[5, 40, 90, \mathit{true}],\\ \mathit{speedTrap}[5, 60, 100, \mathit{true}], \mathit{gps}[5, 20, 0, \mathit{false}]\ ;\ S_{navigation}\ \rangle.\end{array}$$

For instance, the fragment $\mathit{configPanel}$ consists of 11 instructions, and requires 1 Mb of memory at runtime and 0 Mb to be synchronized. Moreover, it is not offloadable.

The structure $S_{navigation}$ of the applications is defined by the following rules:

$$\begin{array}{l}\mathit{configPanel} \longrightarrow \mathit{controller};\\ \mathit{controller} \dashrightarrow \mathit{gps}, \mathit{pathCalc}, \mathit{navPanel};\\ \mathit{gps} \longrightarrow \mathit{controller};\\ \mathit{pathCalc} \dashrightarrow \mathit{map}, \mathit{traffEval}, \mathit{controller};\\ \mathit{traffEval} \longrightarrow \mathit{pathCalc};\\ \mathit{map} \longrightarrow \mathit{pathCalc};\\ \mathit{navigator} \multimap\!\!\!\!\rightarrow \mathit{navPanel}, \mathit{controller};\\ \mathit{navPanel} \multimap\!\!\!\!\rightarrow \mathit{voice}, \mathit{speedTrap}\end{array}$$

Figure 2 reports an intuitive graphical representation of the structure of the navigation application.

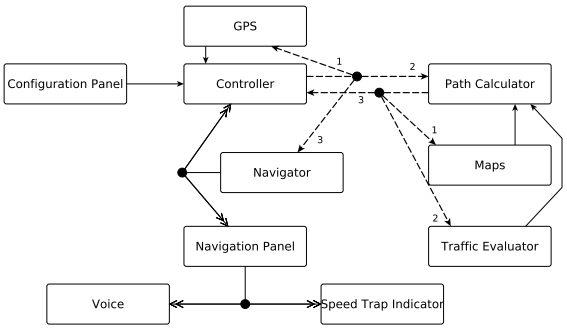

Fig. 2. Navigation application: graphical representation of the structure

III. From MobiCa to Timed Automata

To provide the decision support for MCC applications, in this section we define the translation from the MobiCa language to UPPAAL Timed Automata (TA).

The translation process consists of two main phases: a *static* one that generates the same set of TA (except for some parameters) regardless of the MobiCa specification given as input, and a *dynamic* one that generates TA specifically related to the behaviours described in the given specification.

We present below firstly a brief introduction to timed automata and then we describe the dynamic translation, followed by the static one.

A. Background notions on timed automata

A timed automaton is a finite-state machine extended with clock variables typically used for real-time-system analysis. In UPPAAL, a system is modeled as a network of timed automata running in parallel. The progress of such a system is determined by the enactment of a single automaton transition, or by the synchronization with other automata.

Let us consider, for example, the TA in Figure 3. It is composed of three locations and three transitions. The double circle indicates the initial location, and the labels on the transitions indicate the input/output (with tags ?/!) synchronization actions on channels. There are two types of channels: the binary one, that models binary and blocking synchronization, and the broadcast one, that instead models a kind of asymmetric one-to-many synchronization. In the presence of more than one outgoing transition in a location, one of the enabled transitions will be non-deterministically choosen.

B. Dynamic translation

In this phase we focus on the translation of behavioural information in the application. We follow a compositional approach: first, we separately translate each term $f_1\ Op\ \tilde{f}_2$ of the structure, and then we compose their translations. We start by presenting the translations of single structure terms, and then we show how to compose them.

Below we discuss three different translations of $f_1\ Op\ \tilde{f}_2$, which depend on the interaction operators Op.

Non-deterministic choice. Figures 3 and 4 show the translation of $f_0 \longrightarrow f_1, \ldots, f_n$. Each target fragment f_i, with $i \in \{1, \ldots, n\}$, is translated in a TA (Figure 3) that could be possibly extended during the TA composition discussed later.

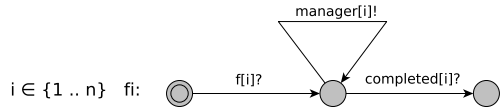

Fig. 3. Translation: target fragments

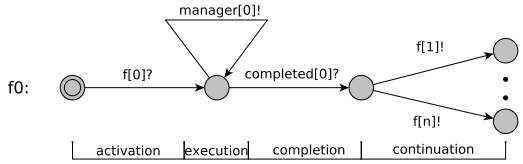

Fig. 4. Translation: non-deterministic choice

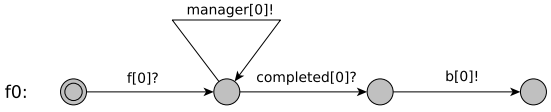

Fig. 5. Translation: parallel execution

This TA models the life cycle of the fragment, composed of three phases: *activation*, *execution* and *completion*. The first transition of the TA is labelled by an input on channel $f[i]$ used to activate the corresponding fragment. From the reached location there are two outgoing transitions, but only one of them is enabled at a given time (depending on the synchronisation with other TA). In the execution phase, the enabled transition is the one labelled by an output on channel $manager[i]$, used for interacting with the manager regulating the execution of the fragment. Once the processor completes the execution of the fragment, the enabled transition is the one labelled by an input on channel $completed[i]$. The TA corresponding to the fragment f_0 (Figure 4) extends the behavior described so far with the *continuation* phase. This additional behaviour corresponds to the translation of the non-deterministic operator. It consists of a set of outgoing transitions from the location reached by the completion phase, which are labelled by outputs on channels $f[1]$, ..., $f[n]$. Each of them is used to activate a target fragment via synchronisation on the corresponding channel. According to the semantics of UPPAAL TA, only one of these transitions will be non-deterministically selected.

Parallel execution. When translating a structure term $f_0 \longrightarrow\!\!\!\!\!\rightarrow f_1, \ldots, f_n$, each target fragment f_i, with $i \in \{1, \ldots, n\}$, is translated as in Figure 3, except for the first transition, where the input on channel $f[i]$ is replaced by an input on the *broadcast* channel $b[0]$. The translation of f_0 is shown in Figure 5. In this case, in the continuation phase, an output on channel $b[0]$ is performed. In this way, all target fragments are simultaneously activated. Indeed, a broadcast channel in UPPAAL allows an output on it to synchronize with an arbitrary number of input transitions on the same channel. Thus, we just need a single channel, which is indexed by the identifier of the sender fragment (f_0). Notably, the broadcast sending is never blocking, so it can be executed also if there is no receiver.

Sequential progress. In case of a structure term $f_0 \dashrightarrow f_1, \ldots, f_n$, the translation of the target fragments is like in the non-deterministic case (Figure 3), enlarged in the continuation phase, with a transition labeled by an output on channel $f[0]$ that brings back the activation to f_0. Instead, the translation of f_0 takes care, in the continuation phase, of

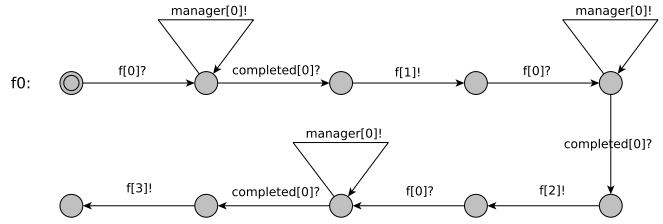

Fig. 6. Translation: sequential progress (example for $f_0 \dashrightarrow f_1, f_2, f_3$)

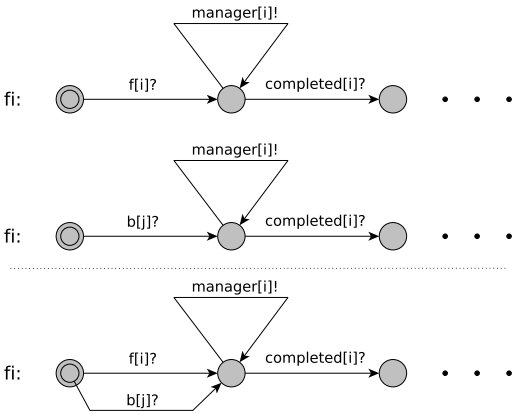

Fig. 7. Translation: merging in the activation phase

activating in sequential order the fragments f_i by means of outputs on channels $f[i]$ (Figure 6). Notably, each time a new target fragment is activated, f_0 waits for a new activation and execution, possibly involving a different number of instructions. The TA terminates with the activation of the last target fragment f_n.

Now, we show how the TA obtained by means of the translations discussed above are merged together. Indeed, a fragment can appear more than once as source or target of an edge of the structure. Thus, to have a single TA for each fragment at the end of the translation, all TA corresponding to the same fragment are collapsed. This merging process only affects the activation and the continuation phases, thus leaving unchanged the execution and completion ones.

Merging in the activation phase. This part of the merging process removes the redundancies on the activation of the TA corresponding to a given fragment. In particular, it unifies all states and transitions in common and leaves the rest of the structure unchanged. At the end of the process, the TA of the fragment f_i will have at most one transition labelled by an input on $f[i]$, and possibly a number of transitions labelled by inputs on channels $b[j]$. We show a simple example in Figure 7: the two TA in the upper part are composed to generate the one in the lower part. In this way, the fragment f_i can be activated individually (by means of channel $f[i]$) or in parallel with other fragments waiting for a communication on the same broadcast channel $b[j]$.

Merging in the continuation phase. This part of the merging process takes care of structure terms with the same source. The resulting TA contains in the continuation part the union of the transitions of each operator translation. This corresponds to composing the continuation parts in a non-deterministic way, as prescribed by the informal semantics of MobiCa presented in Section II. In fact, UPPAAL non-deterministically chooses a synchronization if several possibilities are enabled. We show

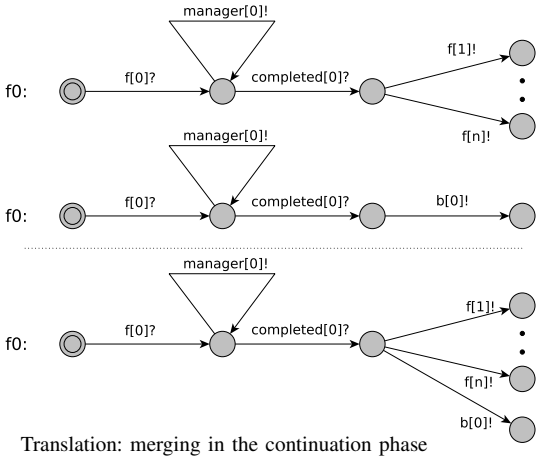

Fig. 8. Translation: merging in the continuation phase

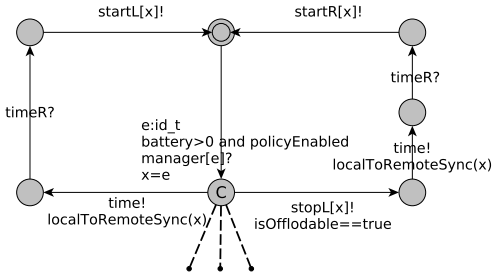

Fig. 9. Translation: manager (an excerpt)

an example in Figure 8, where the continuations of a non-deterministic operator and of a parallel one are combined.

C. Static translation

In this phase of the translation we generate all TA that always have the same form (e.g., the one corresponding to the CPU) and produce all necessary declarations.

Manager. The manager is the instrument used by the system for managing the execution of each fragment. It decides which action to perform, so directing the computation locally or remotely. The corresponding TA is shown in Figure 9. From the initial state, it performs a transition to take as input the index of the calling fragment (stored in the variable x). Then, according to the values of variables describing the status of the system (e.g., battery level) and to the fact that the previous execution was performed locally or remotely, the manager decides the action to perform. The permitted actions are the following ones (only two of them are depicted in the figure):

- *Local and remote start*: These actions are used for starting the fragments locally or remotely according to the current condition of the system. In the TA, the local start is realised by means of an output on channel $startL[x]$. Before this transition, there is an interaction with the TA representing the clock of the system (via channels $time$ and $timeR$) in order to model the elapsing of time due to the data synchronisation.
- *Offloading and migrate*: These actions are used for moving the computation of a fragment from local to remote. In the former action, the fragment is kept loaded in the local memory, while in the latter the memory is released. In particular, the offloading action is modelled by a sequence of outputs on channels $stopL[x]$ (the local fragment is stopped) and $startR[x]$ (the fragment is started remotely).
- *Back and restore*: These actions are complementary to offloading and migrate, respectively. The former one brings back the computation locally, leaving the fragment loaded in the cloud. The latter one brings back the computation locally as before, but freeing the memory in the cloud.

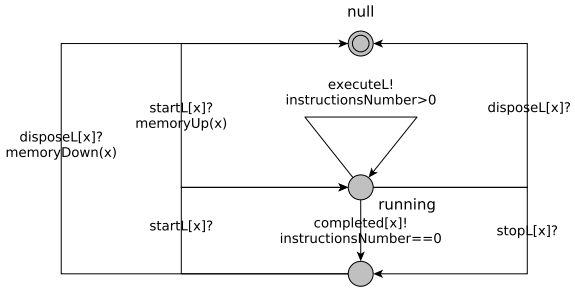

Fig. 10. Translation: local state

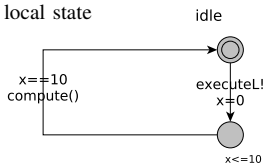

Fig. 11. Translation: local CPU

Local and remote states. The execution state of each fragment is maintained by two TA: one for the local state (Figure 10) and another for the remote one (which is similar, and thus omitted). The state can assume three values, corresponding to three locations in the TA: *null*, *idle*, and *running*. All fragments initially are in the null state; then, according to the action selected by the manager, they change their states. A fragment can synchronize with the CPU, and execute a computation cycle, only when it is in the running state. This process can be repeated so long as there are available instructions. Only when all instructions have been executed the fragment can complete (which is signalled by an output on channel $completed[x]$) and pass to the idle state. Between consecutive cycles of the processor, the system can re-call the manager and proceed with the computation accordingly.

The remote state TA has the same structure of the local one. Notably, at a given time, only one of the two automata can be in the running state for the same fragment. For example, when the local state is running, the remote one is either idle or null. In this situation, an offloading action results in the passage from running to idle locally and from null/idle to running remotely.

Local and remote CPU. The TA corresponding to the CPU play a crucial role in the appropriate modelling of the device's battery life and the elapsing of the computational time. We have two TA, one modelling the CPU of the mobile devices and another for the CPU of the cloud machines. Each TA is composed of two locations with a circular behavior (see Figure 11 for the local CPU). The first transition receives in input the activation by a fragment for executing some instructions. Then, the second transition lets time pass by a given amount, and decreases the battery level and the number

of instructions, all by means of the *compute()* function. If the fragment has a number of instructions that is less than the processor capacity, the *compute()* function decreases the battery level in percentage of the executed instructions. For the remote processor we have a similar TA, where however the battery consumption is not taken into consideration.

Non-structural details. The last step of the translation concerns the definition of non-structural application details, which will be declared as global variables in the UPPAAL model. Among these details, we have the information related to the environment that influences the execution of the application but is not part of it, like the initial battery levels, the computational power of CPUs, the bandwidth of the network connection, etc. UPPAAL provides different ways to represent these data. For our purpose we use integer constants for representing the number of instructions executed per second by the processors and the value of the current network bandwidth. For all other parameters, like memory and battery, we use integer variables. The translation of the information of each fragment is more complex. To have a more compact representation, we have defined a data structure composed by three integer variables, representing the number of instructions, the amount of memory required to install the fragment and the amount of memory for data synchronisation, and by a boolean variable, describing the offloadability of a fragment. So, the information on all fragments is encapsulated in an array of elements having the above structure. Other declarations concern the channels used for the inter-fragment interaction. As shown in Section III-B, we use two types of channels: synchronous and broadcast. Finally, we have also declared a clock variable for managing the elapsing of time.

IV. MobiCa at work

The use of a domain specific language (DSL) to describe MCC systems simplifies the task of a system developer. Indeed, it is much easier and faster to design an MCC system in MobiCa than to model it entirely in UPPAAL. On the other hand, the design and implementation of a DSL is not a trivial task; a good DSL should have a formal semantics, a parser, a compiler and a code generator. To simplify these steps we rely on the Xtext tool, an Eclipse open-source framework that allows one to generate editors equipped with auto-complete mechanisms, syntax highlighting, code completion and static error highlighting.

The Xtext tool offers a language for specifying the grammar of the accepted DSL. Using this grammar we can describe a set of rules that determine if some expressions are valid or not for the desired language. Of course, in order to have a good editor, in addition to the grammar specification we should provide some other functionality such as validators, scopes and translation rules. The implemented grammar for the MobiCa language, which we omit here for lack of space, is a bit more complicated than the abstract one presented in Table I. It includes some supplemental keywords, identifiers and symbol names, which are required by Xtext.

Figure 12 shows the MobiCa Eclipse environment generated using the Xtext framework. In the Eclipse Project Explorer, it is possible to see the navigator case study; in particular, its implementation in MobiCa is shown in the editor. The `Navigator.mcc` is composed by a network of a `Cloud_machine` and a `Mobile_device` that collaborate for providing the navigator system. The navigator application is defined using the keyword `Application`, and it is composed of a set of `Fragments` that are connected according to the transitions defined inside the `Structure` block.

Using the generator, Xtext is able to transform the compact Navigator application into the corresponding UPPAAL TA format, written in a new XML file that will then be loaded in the UPPAAL tool, as an ordinary system to be analyzed.

The analysis of an application is a fundamental phase during its life cycle. In this paper, we propose to take advantage of the analysis techniques used at design time as decision support for the application at runtime. The idea is to use the diagnostic trace, which is automatically generated by the tool during the verification of system properties, as a schedule for taking decisions at runtime. UPPAAL can provide three kinds of traces according to the diagnostic trace set-up: some trace, the shortest one, or the fastest one.

In Table II, we present some relevant queries for the navigator case study and the obtained results according to the selected type of diagnostic trace.

Query	Trace	Time	Battery	Memory
E<>VOICE.id36 and globalTime<300	Shortest	299	87	21
and battery>50 and memory<100	Fastest	255	72	41
E<>VOICE.id36 and globalTime<299 and battery>72 and memory<41	Fastest	267	80	21
E<>VOICE.id36 and globalTime<400 and battery>50 and memory<100	Some	325	84	43
and fragmentStateRemote[3]==0	Fastest	281	69	63

TABLE II. UPPAAL verification results

Note that the queries in Table II are not related only to a specific application, but they have parameters, such as battery, memory and elapsing of time, that are relevant to MCC at large. The first query can be read as: "It is possible to find a trace where VOICE is activated in at most 300 units of time, with at least 50% of residual battery and with memory usage that is smaller than 100 Mb". Verifying this property in the UPPAAL tool using the shortest-diagnostic-trace option, we obtain as result the shortest system trace that satisfies the mentioned query. The trace is shown in Figure 13, where for simplicity we consider only the execution of one of the three iterations of the fragment *CONTROLLER*. The fragment is activated with a synchronization on channel $F[1]$ and proceeds by calling the manager. Note that, before its execution, the fragment checks if a data synchronization is needed, and then starts the execution remotely. After the execution of all instructions, the *CONTROLLER* will pass the execution to the *GPS* and so on. The given trace proceeds until the activation of the fragment *VOICE*, listing step by step if the fragment is executed locally or remotely. The final resource usage for the mentioned query is shown in Table II. The residual battery is equal to 87, the used memory is equal to 21 and the time consumed is equal to 299. These results are obtained executing the fragments *CONTROLLER, PATH_CALCULATOR, TRAFFIC_EVALUATOR, MAP, NAVIGATOR,* and *NAVIGA-*

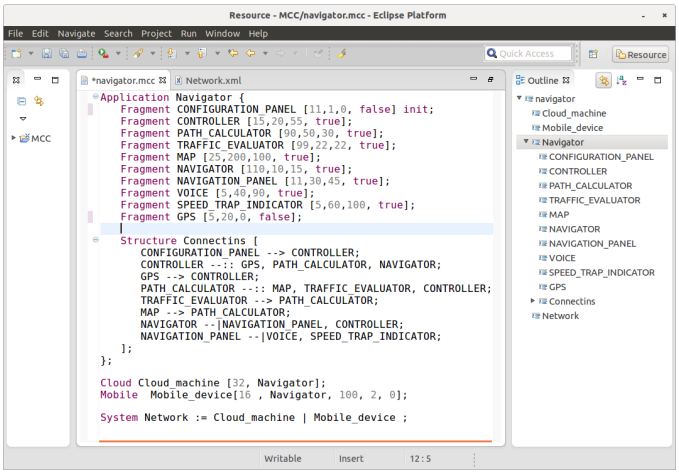

Fig. 12. MobiCa Editor: Navigator application

TION_PANEL remotely, whereas *CONFIGURATION_PANEL* and *GPS* are executed locally. Performing the same query, but setting the diagnostic-trace option to fastest trace, we have an improvement in time performance, but with a higher usage of battery energy and memory. Unlike in the previous scenario, the resulting trace suggests that one executes the first two iterations of the *CONTROLLER* fragment locally and all the other fragments as before. Of course, we can also check if there exists a good compromise between performance and resource usage. For example, as depicted in the second row in Table II, we can check a query using the results of the previous queries as constraints. In this case, using always the fastest-trace option, we obtain a result with a minimal consumption of memory and a good compromise in time and battery usage. In order to achieve this result, the trace suggests that we execute the *CONTROLLER* fragment locally at the first iteration and remotely in the others. This result is possible thanks to a dispose action that is executed by the manager at the beginning of the second iteration.

Using UPPAAL, it is also possible to verify more complex queries, like the one in the third row of Table II. In addition to the previous constraints, there we want to force the system to terminate the *TRAFFIC_EVALUATOR* fragment locally. The result, using the fastest-trace option, shows a decrease in performance and in both battery and memory consumption, but guarantees the best trace in performance while ensuring that the given fragment terminates locally. Notably, the best solution is to execute this fragment locally only at the last iteration, in order to satisfy the constraints and to take advantage of the remote execution power for the first two iterations.

V. Related Work

The state of the art in MCC offers techniques, such as optimization metrics, partitioning methods and offloading strategies, in order to deal with the fact that the state of an application depends on the specific input provided by the users and that the environment is constantly changing. We now survey some of the approaches used in current MCC frameworks to achieve optimization, partitioning and offloading.

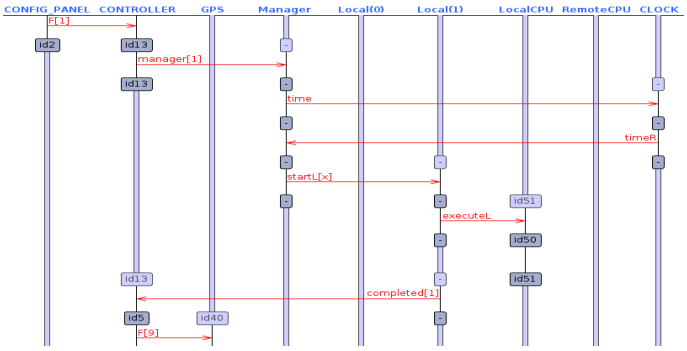

Fig. 13. UPPAAL Verification

A. Optimization metrics

The main objective of MCC is to attempt to maximize or minimize a chosen metric while achieving the expected system behaviour. Different metrics can be adopted; energy saving and time saving are probably the most important. However, there is no agreement on which metrics to employ and most frameworks only consider a subset of them at design time.

The Remote Processing Framework (RPF) from [17] attempts to improve battery lifetime by migrating large tasks on server machines. On the other hand, more innovative frameworks like Odessa [16] try to improve system performance using a greedy algorithm to manage data parallelism, stage offloading and pipeline parallelism. CloneCloud [6] can minimize both execution time and energy used for a target computation. MAUI [7] and Cuckoo [12] consider both energy and execution time. Chroma [3] weighs both execution time and fidelity when making decisions. Instead Spectra [11] balances execution time, energy and application quality.

Sometimes having more than one optimization metric generates additional challenges, mainly due to the different measurement units (second, joule, etc.) and moreover because optimization metrics may be conflicting. For example, consider energy saving and time saving together: the faster the execution, the more energy is required. The same situation occurs when considering time saved and network delay: a longer computation requires offloading to speed up the execution but, at the same time, usually more network transmission is needed for data synchronization. The literature proposes different approaches to deal with these issues. For example, MAUI treats the combination of multiple optimization metrics using a linear programming model and maximizing a given function. Instead, Odessa executes a computation remotely only if all the considered metrics are improved by offloading or bases offloading decisions on a priority measure over metrics.

The choice of an 'optimal' optimization metric is closely related to the type of application, the input provided by the users and the preferences that they have in system optimization. Hence the selection of optimization metrics in the above-mentioned frameworks is not always profitable: it may or may not match the preferences of the users of an application. So a bad choice in terms of metric may degrade the users' experience. For this reason, in MobiCa we consider multiple optimization metrics using verification queries. These queries are expressed in a temporal logic, and can be directly verified using model checking techniques. The expressiveness

of the logic allows one to freely combine different optimization metrics. This technique also gives the developer the possibility to analyze different metrics at the same time and to use the one that fits the given constraints best. Another interesting advantage is the possibility for the end user to interact with the system and to define his own threshold-constraint preference, in order to customize the optimization query.

B. Partitioning methods

Partitioning is the step during which a developer decides on a method for splitting the application into offloadable and not offloadable components. In general, there is a large number of ways to split an application, but very fine-grained partitions lead to a degradation in performance.

Finding the 'best' partition is not a simple task. One obvious strategy is to explore all possible partitions and choose the best one. Of course, this method is intractable in practice because of the very large number of candidate partitions. More static methods, like the one used in RPF, generate a copy of the application remotely and at runtime decide where to execute a fragment. The main drawback of this approach is the replication of information: in order to keep the data consistent, the system should continuously exchange updated data. On the other hand, Spectra relies on a partition provided by the developer, who can use his intuition and knowledge of the morphology of the application to try and achieve good results. This method has low complexity, but lacks adaptability, which is a major drawback in the highly dynamic world of MCC. A similar approach is used in MAUI, which asks the developers to annotate the offloadable methods in an application. Then at runtime MAUI uses its solver, at each method invocation, to decide if the candidate method should be offloaded. In Chroma the developer specifies a possible partition by defining some tactic files, each of which describes a possible way of combining remote procedure calls. At runtime, Chroma explores all possible tactic plans and picks the best one for the given resource availability. As another example, Odessa models an application as a data-flow graph in which the nodes are computational components, called stages, and the edges are connectors representing the data dependency between the stages. At runtime, Odessa dynamically decides where stages should be placed and when to employ a parallel computation. This representation of an application is more suitable for coarse-grained partitioning because the graph structure abstracts from many details. Cuckoo is a Java based framework relying on the programming activity/service model used in Android, which distinguishes between computation intensive parts (services) and interactive parts of the application (activities). Cuckoo requires the programmer to give a local implementation and then generates code for the same implementation on a remote server. Typically, remote and local methods are the same but they may also be different, since the remote implementation can run on a different architecture.

Other, more innovative methods choose the partition granularity without requiring any programmer assistance. They rely on advanced virtual machines that are able to identify components that can be remotely executed using reflection-oriented programming and other innovative techniques. CloneCloud offers an excellent example of this approach; it identifies constraints on possible migration points using static analysis techniques on code compiled to run on Android's Dalvik virtual machine. The static analysis in this framework distinguishes three kinds of situations: 1) tasks that access hardware devices must execute on the platform that has those hardware features; 2) tasks that access the same memory location should be executed on the same device; and 3) if a task is offloaded, also the methods that it invokes should be offloaded.

Summing up, the above-mentioned frameworks assume that the number of partitions in an application is usually small. One reason could be that only one or two components in an application contain substantial computation, and so only these computationally heavy tasks are candidates for offloading.

Also in MobiCa a developer can describe the partitioning of an application. We recall that an application is given by a set of fragments, which interact with each other following a given structure. The model provided for describing the application is similar to the one proposed in Odessa, but it gives a developer more freedom in selecting the granularity of a partition.

C. Offloading strategies

MCC frameworks can also be categorized according to their offloading strategy, that is, the process used by a MCC system to select a partition that best achieves its predefined goals. This should be the partition that maximizes the system's chosen metric or utility function. In order to achieve the offloading strategy, the mentioned frameworks should rely on some mobile environment conditions, such as network bandwidth, latency and server load, but also on application-related constraints like the amount of computation required.

RPF bases its strategy on the idea that the decision procedure should be adaptive and based on direct observation. For this reason, it requires that the same process run alternately locally and remotely in order to determine which is currently the best choice. This approach has limited scalability and can lead to poor performance in highly dynamic systems, where the available resources and the cost of a previously-learned partition change rapidly.

These limitations with direct observation have led to other approaches that separately predict *resource supply* (resource availability) and *resource demand* (resource required by the partition). The system MAUI falls under this typology: the demand in this case is the amount of bytes required to send the input for the computation to the remote cloud plus the amount of bytes required to deliver the result of the computation back to the mobile device. MAUI continuously measures the network connection to estimate its bandwidth and latency. Based on these data, MAUI formulates an optimization problem whose solution can be used to predict which method should be offloaded or executed locally on the smartphone. MAUI can adapt quickly to changes in the network conditions, but, like in RPF, can realize that the environmental conditions have changed only via direct observation.

Other approaches that use the supply-demand method, such as the one proposed in [15], overcome this limitation with a history-based technique, which has been used in other MCC systems such as Spectra and Chroma.

History-based prediction frameworks build a model based on past inputs and use it to make predictions or decisions,

rather than following only the current system configuration. In order to have a good result in terms of battery and computation required, the existing frameworks rely also on some heuristics given by the developer. The hints provided by the developer are usually related to the typology of the application input and the complexity of the fragment.

Odessa proposes a history-based prediction based on a decision engine, running in two different threads, that is able to manage simultaneously data parallelism, stage offloading and pipeline parallelism. Both threads make their decisions using a greedy algorithm that looks for bottlenecks based on historical data. If bottlenecks are found, the algorithm increases the parallelism in the computation or offloads the code remotely. A similar approach is used by Cuckoo, where the framework intercepts a method invocation using a proxy and then decides whether to invoke the method locally or remotely, using heuristics, context information and history.

Compared to above-mentioned works, the MobiCa approach takes decisions at runtime basing the offloading strategies on real resource usage data and device application congestion. This methodology allows for more accurate results with each configuration metric and system configuration.

VI. Conclusions

In this paper we have presented a formal methodology for providing runtime decision-support for MCC applications. The methodology is based on MobiCa, a new language that focuses on expressing the aspects of an MCC system that are relevant for offloading decisions. The language semantics permits to automatically generate a global model that represents the entire MCC scenario under consideration. This model can be used as input for the UPPAAL model-checker in order to improve system performance, reduce energy usage and so give a rich user experience.

For the time being, the proposed approach for the offloading decision support has been implemented as a proof-of-concept. Thus, the optimization of its performance is left for future work. Another aspect that we do not consider in this paper concerns how the proposed approach could be transferred to technology. A possible idea is to include the decision support as part of a larger middleware infrastructure, built above the operating systems of the mobile device and the cloud machines. The scope of the middleware should be to provide the right information and parameters of the running environment to the UPPAAL analyzer. Considering the existing frameworks in the literature, our runtime analysis could be simply integrated inside the solver of the well-known middleware MAUI (described in Section V). Another interesting direction is to study the best interval of time for consecutive invocations of the decision support system. Indeed, the analysis results obtained with a snapshot of the system configuration are useful only for the period of time in which the values of the constants (e.g., network bandwidth) change only slightly. Currently, we use a static time window, but we can easily improve our analysis considering continuous time, introducing a new TA to model the network bandwidth fluctuations basing its values on a historical resource trend. As a last improvement, we can incorporate in MobiCa a policy language for allowing developers to program custom (offloading) policies.

ACKNOWLEDGEMENTS: Luca Aceto has been supported by the projects 'Nominal Structural Operational Semantics' (nr. 141558-051) of the Icelandic Research Fund and 'Formal Methods for the Development and Evaluation of Sustainable Systems', grant under the Programme NILS Science and Sustainability, Priority Sectors Programme of the EEA Grants Framework. Andrea Morichetta and Francesco Tiezzi have been supported by the EU projects ASCENS (257414) and QUANTICOL (600708) and by the MIUR PRIN project CINA (2010LHT4KM).

References

[1] "The mobile app economy is exploding." [Online]. Available: http://goo.gl/doZ6vr

[2] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.

[3] R. K. Balan, M. Satyanarayanan, S. Y. Park, and T. Okoshi, "Tactics-based remote execution for mobile computing," in *MobiSys*. ACM, 2003, pp. 273–286.

[4] M. V. Barbera, S. Kosta, A. Mei, and J. Stefa, "To offload or not to offload? The bandwidth and energy costs of mobile cloud computing," in *INFOCOM, 2013*. IEEE, 2013, pp. 1285–1293.

[5] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on Uppaal," in *Formal Methods for the Design of Real-time Systems*. Springer, 2004, pp. 200–236.

[6] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "CloneCloud: Elastic execution between mobile device and cloud," in *EuroSys*. ACM, 2011, pp. 301–314.

[7] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: Making smartphones last longer with code offload," in *MobiSys*. ACM, 2010, pp. 49–62.

[8] H. T. Dinh, C. Lee, D. Niyato, and P. Wang, "A survey of mobile cloud computing: architecture, applications, and approaches," *Wireless Comm. and Mobile Computing*, vol. 13, no. 18, pp. 1587–1611, 2013.

[9] N. Fernando, S. W. Loke, and W. Rahayu, "Mobile cloud computing: A survey," *Future Generation Computer Systems*, vol. 29, no. 1, pp. 84 – 106, 2013.

[10] J. Flinn, "Cyber foraging: Bridging mobile and cloud computing," *Synthesis Lectures on Mobile and Pervasive Computing*, vol. 7, no. 2, pp. 1–103, 2012.

[11] J. Flinn, S. Park, and M. Satyanarayanan, "Balancing performance, energy, and quality in pervasive computing," in *Distributed Computing Systems, 2002.*, 2002, pp. 217–226.

[12] R. Kemp, N. Palmer, T. Kielmann, and H. Bal, "Cuckoo: A computation offloading framework for smartphones," in *Mobile Computing, Applications, and Services*. Springer, 2012, pp. 59–79.

[13] K. Kumar and Y.-H. Lu, "Cloud computing for mobile users: Can offloading computation save energy?" *Computer*, vol. 43, no. 4, pp. 51–56, 2010.

[14] A. P. Miettinen and J. K. Nurminen, "Energy efficiency of mobile clients in cloud computing," in *USENIX*, 2010, pp. 4–4.

[15] D. Narayanan, J. Flinn, and M. Satyanarayanan, "Using history to improve mobile application adaptation," in *Workshop on Mobile Computing Systems and Applications*, 2000, pp. 31–40.

[16] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan, "Odessa: Enabling interactive perception applications on mobile devices," in *MobiSys*. ACM, 2011, pp. 43–56.

[17] A. Rudenko, P. Reiher, G. J. Popek, and G. H. Kuenning, "The remote processing framework for portable computer power saving," in *SAC*. ACM, 1999, pp. 365–372.