IMT School for Advanced Studies, Lucca

Lucca, Italy

A Formal Approach to Decision Support on Mobile Cloud Computing Applications

PhD Program in Computer, Decision, and Systems Science

XXVIII Cycle

By

Andrea Morichetta

2016

The dissertation of Andrea Morichetta is approved.

Program Coordinator: Prof. Rocco De Nicola, IMT School for advanced studies, Lucca

Supervisor: Prof. Rocco De Nicola, IMT School for advanced studies, Lucca

Supervisor: Prof. Francesco Tiezzi, University of Camerino

Tutor: Prof. Francesco Tiezzi, University of Camerino

The dissertation of Andrea Morichetta has been reviewed by:

Dan Grigoras, University College Cork

Ivona Brandic, University of Wien

IMT School for Advanced Studies, Lucca

2016

Acknowledgements

In writing this thesis I closed a fantastic chapter of my life, full of challenges and hard work. It has been a great privilege to spend three years in a prestigious school like the IMT in the amazing city of Lucca.

I would like to express my sincere gratitude to my co-supervisor Francesco Tiezzi. He continually and convincingly supported me. Without his guidance and persistent help this dissertation would not have been possible. I would also like to thank my supervisor Rocco De Nicola, for his support through all these years.

I would like to thank many other people like Kim G. Larsen that hosted me for six months in the CISS Group at the Aalborg University and all my co-authors with whom I had the pleasure to work, special thanks are due to Luca Aceto for his tireless work and unfailing readiness.

A special thank is due to my family and Elisa for helping me to survive all the stress of these years and not letting me to give up.

Contents

List of Figures

List of Tables

Declaration

In this thesis we focus on the main field of my research, so we include only part of the publications achieved during my Phd program. Part of the material presented has been previously published in two co-authored papers. Chapter 3 is based on (AMT15) a joint work with Luca Aceto, Reykjavik University and Francesco Tiezzi, University of Camerino. Chapter 4 is based on (AMT15) and (ALMT15) a joint work with Luca Aceto, Reykjavik University, Kim G. Larsen, Aalborg University and Francesco Tiezzi, University of Camerino.

Vita

February 3, 1987	Born, Macerata, Italy.
2006-2009	Bachelor degree in Computer science, Final mark: 107/110, University of Camerino.
March 2011 - June2011	Erasmus period, University of Iceland.
2009-2011	Master Degree in Computer Science, Final mark: 110/110 cum laude, University of Camerino.
March 2012- February 2013	Research Fellow in the Software E. group, I.S.T.I. A. Faedo CNR, Pisa, Italy.
February 2013 Present	Phd Candidate, IMT Lucca, Italy.
November 2014 - April 2015	Visiting Professor Kim G. Larsen, University of Aalborg, Denmark.
June/July 2015	External Examiner in Computer Science, ITC ”A Gentili”, Macerata, Italy.

Publications

1. Programming and Verifying Component Ensembles. Rocco De Nicola, Alberto Lluch-Lafuente, Michele Loreti, Andrea Morichetta, Rosario Pugliese, Valerio Senni, Francesco Tiezzi: FPS@ETAPS 2014: 69-83
2. The SCEL Language: Design, Implementation, Verification. Rocco De Nicola, Diego Latella, Alberto Lluch-Lafuente, Michele Loreti, Andrea Margheri, Mieke Massink, Andrea Morichetta, Rosario Pugliese, Francesco Tiezzi, Andrea Vandin: The ASCENS Approach 2015: 3-71
3. Decision Support for Mobile Cloud Computing Applications via Model Checking. Luca Aceto, Andrea Morichetta, Francesco Tiezzi: MobileCloud 2015: 199-204
4. A Cost/Reward Method for Optimal Infinite Scheduling in Mobile Cloud Computing. Luca Aceto, Kim G. Larsen, Andrea Morichetta, Francesco Tiezzi: FACS 2015: 66-85

Authors are listed in alphabetical order.

Presentations

Conference talks:

1. Decision Support for Mobile Cloud Computing Applications via Model Checking in the IEEE Mobile Cloud Conerence (San Francisco, US)
2. A cost/reward method for optimal infinite scheduling in Mobile Cloud Computing in the FACS Conference (Rio de Janeiro, BR)

Other talks:

1. Specifying and Verifying SCEL programs with SPIN in the ASCENS Project (Modena, IT).
2. Decision Support for Mobile Cloud Computing Applications via Model Checking, Aalborg University
3. Optimal Scheduling for Mobile Cloud Computing Applications via Model Check- ing in the CINA Project (Civitnova Marche, IT)

Abstract

Mobile Cloud Computing (MCC) is an emergent topic growths with the explosion of the mobile applications. In MCC systems, application functionalities are dynamically partitioned between the mobile devices and cloud infrastructures. The main research direction in this field aims at optimizing different metrics, like performance, energy efficiency, reliability and security, in a dynamic environment in which the MCC application is located. Optimization in MCC refers to taking advantages from the offloading process, that consists in moving the computation from the local device to a remote one. The biggest challenge in this aspect is to define a strategy that is able to decide when offloading and which part of the application to move. This technique, in general, improves the efficiency of a system, although sometimes it can lead to a performance degradation.

To decide when and what to offload, in this thesis we propose a new general framework supporting the design and the runtime execution of applications on their own MCC scenarios. In particular the framework provides a new specification language, called MobiCa, equipped with a formal semantics that permits to capture all characteristics of a MCC system. Besides the strategy optimization achieved by exploiting the potentiality of the model checker UPPAAL, we propose a set of methods for determining optimal finite/infinite schedules. They are able to manage the resource assignment of components with the aim of improving the system efficiency in terms of battery consumption and time. Furthermore, we propose two optimized scheduling algorithms, developed in Java, based on the exploitation of parallel computation in order to improve the system performance.

Chapter 1

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 (EXP). 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) (FLR13; DLNW13; Fli12), 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 (BKMS13) 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 (KL10), 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 at the end of the day without exhausting all its available energy before. Designers are involved in the 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 improving the 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 (MN10), 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.

It is therefore natural to ask how one should best build systems that combine all the introduced concepts. Current solutions tend to move from one extreme to the other; from one side it is given most functionality to the cloud focusing the computation remotely and permitting its access to a thin-client; from the other side it is given most functionality to the mobile focusing the computation in a client architecture that runs usually in a portable device. In both of the cases we are in front of a static partition of the application. This client-server approach is known to be poorly suited for mobile environments, in which resource availability changes rapidly. MCC with respect to this old approach, introduces an important innovation: the dynamism of the application. In this way, the mobile applications are able to adapt to the continuous changing of the resources availability, by reallocating data and computation in order to improve performance, reduce energy usage and satisfy the final user goal.

Applications that benefit from MCC are both interactive and resource-intensive. Non-interactive applications can be executed easily on the cloud, since such applications can tolerate the communication latency between mobile and cloud. Application non resource-intensive instead can be executed entirely on mobile. The application that are currently most popular with mobile users like web-browsing and e-mail require often few resources to the mobile device. Consequently MCC provides some benefit that cannot be perceived by the final user. On the other hand, the list of application supported by MCC includes more demanding applications, like speech recognition, face recognition, speaker identification and image identification. The challenge is to find a method that attempts to perceive the maximum benefit from the MCC paradigm for all applications where there is not a pronounced convenience to execute them in a hybrid computation with respect to the cloud or the mobile.

Here below we describe a concrete offloading problem and all the factors that can influence negatively the MCC achievements. One concrete problem consists in executing a number of interdependent fragments of code into a number of heterogeneous resources like mobile devices and cloud infrastructures. Usually a fragment cannot be executed before all its predecessors have terminated, unless it is in parallel. Furthermore, each fragment can be execute in only one resource at the same time. An example of offloading of fragments is depicted in Figure 1. In detail, there are three fragments (A, B, C) that should be executed in a MCC system. In the first case (1) the computation is local on the mobile device for the fragments A and C and remote on cloud for the fragment B. How it is possible to notice in the figure, with respect to the second case (2) where the computation is maintained local for all fragments, the hybrid computation leads to an improvement of performance outlined by a shorter processing phase. Of course we cannot take this example as a general rule, the offloading is not always convenient. Indeed if we consider the last case (3), the hybrid computation degrade the final performance of the system, behaving worst than the second case. These negative performances are generated by a congested network, that slows down the processing of data synchronization, thus degrading the total performance of the system. In other words, the system is not able to compensate the synchronization time with the speed up obtained by executing the fragment B on a more powerful processor.

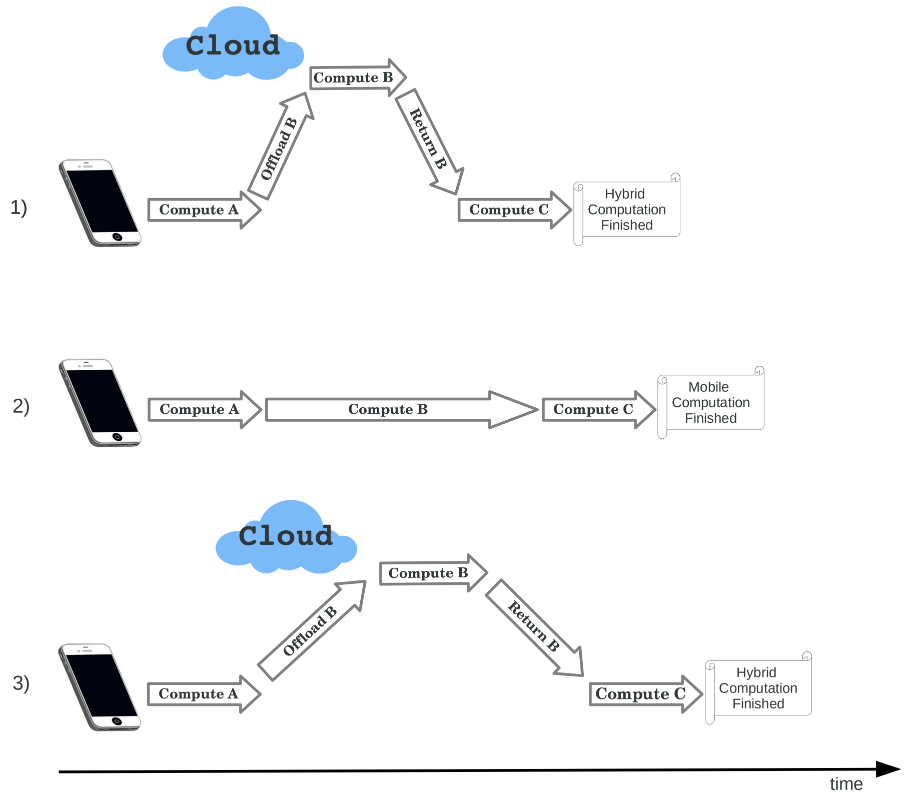


Figure 1: Offloading problem

This thesis will focus on the MCC paradigm, and aims at supporting the development of mobile applications whereby the data processing and storage are moved from the mobile devices to powerful and centralized computing platforms located in the cloud. Creating applications for MCC environments requires computing and communication capabilities, that should be gracefully integrated between local and remote devices. This integration can give a rich user experience under the battery life concern, data/storage capacity and computational power. The user experience as mentioned in (OG13a) is the key concept of a MCC contest. Only integrating new functionalities with a new model of interaction, computation intensive works can have a positive impact on the final user experience. For achieving this goal and make the interactions between the mobile and the cloud as seamless as possible, the developer should consider and take under control a set of parameters like the quality and coverage of the network, the time and the energy cost required for the communication of data over the network and possibly disconnections that inhibit the regular application usage. All these features make the MCC a very attractive and cutting-edge topic in research.

1.1 Contributions

What we want to do in this thesis is to exactly provide a new formal-based methodology, useful in MCC, for the offloading decision support and strategy assessment. The main novelty with respect to the related work in the literature, which mainly uses linear programming methods, is the introduction of a formal verification. This provides an exhaustive system configuration evaluation, by using well-established model checking techniques.
In other words, we imagine to have a MCC scenario where there is an application running on a mobile device that performs code offloading according to some strategies defined by the developer in terms of policies. The duty of these policies is to manage the computation offloading according to the resources usage of the mobile device and the cloud. So, at this point, to achieve a good result in terms of performance and en-

ergy saved the main role is played by the policies setting, and so, by the developer decisions made at design time. Of course, at design time, for the policies configuration, the developer can follow his experience, results of other applications or adjust values according some preliminary tests, but for sure he is not able to foresee all possible situations. For this reason we provide a framework that relies on a formal analysis, based on a model checking technique, to explore automatically and exhaustively all possible situations of the system, and give the right suggestions to the developer for properly tuning the policies configuration.
To achieve the mentioned goals, we need an application model to be used as input in an existing model checker. Indeed, the effort of creating a compatible model with an existing tool is given by the fact that proposing a novel ad-hoc model would require to develop a related software tool for the verification, with all problems connected. Furthermore a framework based on well established tool is more reliable and efficient. To make a fast prevision, our model should contain all information regarding the application behavior and the application surrounding environment. Some possible examples could be: information about the structure of the application, resources usage, offloading strategies, execution traces and so on. At this point it is simple enough to imagine the final complexity of this model and we are also conscious that mobile applications developer will not inclined to invest their time in generate a low-level model during the design of an application. However, we are confident that the developers can provide a more high-level specification of their systems. Therefore, in our framework we provide a high-level language that developers can use for the application specification. The framework, starting from this specification and by exploiting the formal semantics of the language, will generate automatically the input model for the incorporated verification tool. This tool, by means of some specific techniques, is able to generate an optimal scheduler that will manage the offloading strategy in order to improve the performance, while minimizing the energy consumption by devices. These concepts are imperative for the applicability and satisfiability of prefixed service level agreement for computation-hungry applications in small devices.

Below we summarize the contributions presented in this thesis.

MobiCa: a Domain Specific Language for MCC MobiCa is a domain-specific language 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 developer designs the system using MobiCa, by focusing on the MCC characteristics of the application(s) of interest, choosing the right partitioning, the typology and the time of the analysis.

Automatic translation into UPPAAL[1]. At the developer will be required only to provide the application specification defined with the MobiCa language. The framework, starting from this specification and by exploiting the formal semantics of the language, will generate automatically the input model for the verification tool. We have defined and implemented a semantics for MobiCa via translation into timed automata (AD94), which automatically associates an UPPAAL model to each MobiCa specification. Timed automata are a formal language, that can be used to model and analyse the timing behavior of computer systems, and can be verified using the well know model checker UPPAAL (see Chapter 2). This takes a weight off the developer, who does not have to take care of the details specified in the timed automata model. 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.

[1]UPPAAL, is an acronym based on a combination of UPPsala and AALborg universities.

Design-time decision support. A design-time technique is provided to synthesize schedulers that produce infinite schedules ensuring the satisfaction of a property for infinite runs of the application. This is particularly useful for MCC, where applications are supposed to provide permanent services, or at least to be available for a long period. In particular, considering that our model is equipped with constraints on duration, costs and rewards, we are interested in identifying the *optimal* schedulers that permit the achievement of the best result in terms of energy consumption and execution time. In fact, over infinite behaviors, it is possible to recognize a cyclic execution of components that is optimal and is determined by means of the limit ratio between accumulated costs and rewards. Consequently, an optimal scheduler is given by maximizing or minimizing the cost/reward ratio.

Run-time decision support. The main objective of our run-time decision support is to attempt to maximize or minimize a chosen metric while achieving the expected system behaviour at run-time. Different metrics can be adopted; energy saving and time saving are probably the most important. In our approach we try to identify the best execution strategy for a finite amount of time/energy/fragments, basing this method on cost minimization for reachability properties. To mark this methodology usable in practice, we present also a dedicated framework able to incorporate in the application the decision support and to collect important data for the verification phase. Furthermore, the framework is in charge to manage the analysis phase and take care of the reconfiguration of the system according to the environment variations.

Stratego Analysis. Besides to standard UPPAAL models, MobiCa specification can be translated to UPPAAL Stratego models. In this case, rather than verifying a proposed controller synthesis as usual with a Statistical model checking, we use the Stratego facilities. Stratego, by means of an efficient on-the-fly algorithm, is able to construct at design-time a system controller that guarantees the correctness of reachability and

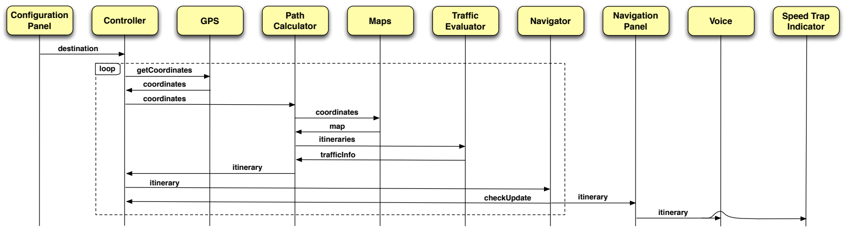


Figure 2: Navigation case study: sequence diagram

safety properties for timed and hybrid games. However, once a strategy has been synthesized, Stratego permits further and deeper analysis in terms of other additional properties that may or may not hold under the strategy. Furthermore it permits to optimize a synthesized nondeterministic safety strategy with respect to a desired performance measure.

Parallel algorithms. We also propose two optimized algorithms, developed in Java, that try to cover the performance issues observed using the model checking techniques and in particular UPPAAL. The main advantages of the proposed algorithms with respect to the model checking methods is the exploitation of parallel computation, that in high distributed systems like the cloud ones, permits to have significant benefits in terms of performance and throughput. The two algorithms are developed on purpose for finding the optimal scheduling in MCC systems, revisiting the well known algorithms for the shortest path and for the depth first search in a tree.

Assesment on a case study. We show the effectiveness and feasibility of our approach by means of a larger case study, drawn from (AMT15), concerning a navigator application. This kind of application is one of the most complex and used in mobile devices. This is an interesting case study for this work from the point of view of its complexity and its strong real-time requirements. In particular, 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, considering the current traffic condition. The dynamism of the current traffic variable requires a constant evaluation of the traffic due to the unpredictability evolution of its condition over time. This characteristic makes the traffic a real-time parameter to consider in order to obtain always the best navigation route.

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 2. The system starts when the user inserts the destination in the *configuration panel* that consequently activates the *controller*. The *controller* in turn asks the *GPS* for the current coordinates and forwards them to the *path calculator*. The *path calculator*, interacting with the *map* and the *traffic evaluator*, will provide a possible itinerary. The itinerary is processed by the *navigator*, which forwards information to the *navigation panel*. This latter component, with the help of the *voice* and *speed trap indicator*, provides the navigation service to the end user. The *navigator* is also responsible for reactivating the *controller* in order to check possible updates of the route.

1.2 Structure of the thesis

The rest of the thesis is organized as follows.
In Chapter 2, to accommodate readers with a cross-disciplinary background, we provide a comprehensive introduction to terminology, notation, and tools that are used extensively throughout the remainder of the thesis. In Chapter 3 we introduce the domain specific language MobiCa, with its syntax and semantics given by translation into Timed Automata. In Chapter 4 we present the main techniques we used to find the optimal scheduling at design-time for infinite behaviour, at run-time for finite behaviour, and using Stratego. In Chapter 5 we introduce two algorithms developed in Java with the purpose of filling the gap of the model checking technique in MCC applications. Chapter 6 provides the context for

our research contribution with a discussion of related work in Mobile Cloud Computing and formal method applied to strategy techniques. Finally, Chapter 7 summarizes the main contributions and propose possible directions for further research in Formal Methods applied to the MCC field.

Chapter 2

Background Notions

The use of formal specification in software and hardware development is a very diffused technique. In the last few years the use of formal verification, like prediction and strategy analysis, on different fields has increased the interest of the researchers.

In the following we present some basic notions of timed automata, underlying the peculiarity of UPPAAL model checker, one of the most relevant verification environments for real time systems. The choice of UPPAAL with respect to its competitors was driven by the high flexibility of timed automata in describing dynamic environments based on time and for its well supported range of tools that has permitted an exhaustive analysis of the system scenario. In our work we rely also on the Xtext tool, that is the base IDE for developing new domain specific languages on Eclipse. This tool allows us to specify the structure of a MCC system in MobiCa exploiting all the supporting feature available on a generic language.

2.1 Timed Atomata

In this section we report key concepts concerning timed automata; notions and definitions are mostly borrowed from the book by Aceto et al. (AILS07). A timed automaton is a finite state machine composed by states, transitions, and real valued-clocks typically used for real-time system analysis. The time is the main characteristic that distinguishes timed automata from other formalisms; in particular time can elapse when the automaton is in a state or location. The system can evolve from one state to another one only if exists a transition that connects them; the execution of a transition is supposed to be instantaneous. A clock constraint on a transition is called guard, and invariant when it is on a location. Time can elapse in a location only if the invariant is satisfied, while a transition can occur only if the guard associated is satisfied. A guard can be a boolean expression or a condition on clock values. The time in the system elapses at the same rate for all clocks and at any instant a clock is equal to the time that has elapsed since the last time it was reset.

Definition 1 (Timed Automaton) *A timed automaton over a finite set of clocks C and a finite set of actions* ***Act*** *is a quadruple*

$$(L, \ell_0, E, I),$$

where:

- *L is a finite set of locations, ranged over by ℓ,*
- *$\ell_0 \in L$ is the initial location,*
- *$E \subseteq L \times B(C) \times Act \times 2^C \times L$ is a finite set of edges, and*
- *$I : L \to B(C)$ assigns invariants to locations.*

The set of clock constraints B(C) is defined by the abstract syntax:

$$g, g_1, g_2 ::= x \bowtie n \mid g_1 \wedge g_2;$$

where $x \in C$ is a clock, $n \in N$ and $\bowtie \in \{\leq, <, =, >, \geq\}$
We write $\ell \xrightarrow{g,a,r} \ell'$ instead of $(\ell, g, a, r, \ell') \in E$. For such an edge, ℓ is called the source location, g is the guard, a is the action, r is the set of clocks to be reset and ℓ' is the target location.

Semantically, a timed automaton is modelled by a transition system defined as follow:

Definition 2 (Timed Labeled Transition System (TLTS)) *Let A = (L, ℓ_0, E, I) be a timed automaton over a set of clocks C and a set of actions Act . We define the timed labelled transition system T(A) generated by A as T(A) = (Proc, Lab, \{ $\xrightarrow{a}$ $|a \in$ Lab \}), where:*

- *Proc = \{(ℓ, v) | (ℓ, v) $\in$ L $\times$ (C $\rightarrow \mathbb{R}_{\geq 0}$) and v $\models$ I(ℓ)\} , i.e. states are of the form (ℓ,v), where ℓ is a location of the timed automaton and v is a valuation that satisfies the invariant of ℓ ;*
- *Lab = Act $\cup$ $\mathbb{R}_{\geq 0}$ is the set of labels;*
- *$\longrightarrow$ is defined by:*
  - *$(\ell, v) \xrightarrow{a} (\ell', v')$ if there is an edge ($\ell \xrightarrow{g,a,r} \ell'$) $\in$ E such that v $\models$ g, v′ = v[r] and v′ $\models$ I(ℓ'),*
  - *$(\ell, v) \xrightarrow{d} (\ell, v + d)$ for all d $\in \mathbb{R}_{\geq 0}$ such that v $\models$ I(ℓ) and v+d $\models$ I(ℓ).*

Let v_0 denote the valuation such that $v_0(x)$ = 0 for all x$\in$C . If v_0 satisfies the invariant of the initial location ℓ_0, we shall call (ℓ_0, v_0) the initial state (or initial configuration) of T(A).

We use v[r] to denote the set of clocks valuations where the values of clocks r are reset to zero. Formally:

- for each d $\in \mathbb{R}_{\geq 0}$ the valuation v+d is defined by:

$$(v + d)(x) = v(x) + d \; for \; each \; x \in C$$

- for each r $\subseteq$ C, the valuation v[r] is defined by:

$$v[r](x) = \begin{cases} 0 & if \; x \in r, \\ v(x) & otherwise. \end{cases}$$

Generally a real time model is a collection of timed automata running in parallel, called network of time automata. Timed automata in a network are able to collaborate by synchronizing input/output (with tags

?/!) actions on transition channels. The communication is implemented when one parallel component raises a synchronization request on a particular channel and another component accepts the request on the same channel. Both components can then simultaneously perform the communication transitions, and the common assumption is that the duration of the synchronization action is zero time units, so, the communication is instantaneous. This form of communication is also called *hand-shake* synchronization.

Definition 3 (Network of Timed Automata) *Let n be a positive integer and, for each i ∈ {1,..., n}, let*

$$A_i = (L_i, \ell_0^i, E_i, I_i)$$

*be timed automata over a set of clocks C and a set of actions **Act**. We call the composition* $A = A_1|A_2| \ldots |A_n$ *a network of timed automata with* n *parallel components.*

The following definition formalizes the behaviour of a network of timed automata.

Definition 4 (Network of Timed Automata Semantics) *Let* $A = A_1|A_2| \ldots |A_n$*, where* $A_i = (L_i, \ell_0^i, E_i, I_i)$ *for each* $i \in \{1, ..., n\}$*, be a network of timed automata over a set of clocks C and actions* $Act = \{c! \mid c \in Chan\} \cup \{c? \mid c \in Chan\} \cup N$*. We define the TLTS T(A) generated by the network A as*

$$T(A) = (Proc, Lab, \{\xrightarrow{\alpha} |\alpha \in Lab\}).$$

Here

- $Proc = \{(\ell_1, \ell_2, ..., \ell_n, v)|(\ell_1, \ell_2, ..., \ell_n, v) \in L_1 \times L_2 \times \ \times L_n \times (C \to \mathbb{R}_{\geq 0})$ *and* $v \models \bigwedge_{i \in \{1,\ldots,n\}} I_i(\ell_i)\}$ *(i.e. states are of the form* $(\ell_1, ..., \ell_n, v)$*, where each* ℓ_i *is a location in the component timed automaton* A_i *and* v *is a valuation over the set of clocks C that satisfies the invariants of all locations i present in the state),*
- $Lab = N \cup \{\tau\} \cup \mathbb{R}_{\geq 0}$ *is the set of labels, and*
- *the transition relation is defined as follows:*

- $(\ell_1, ..., \ell_i, ..., \ell_n, v) \xrightarrow{a} (\ell_1, ..., \ell'_i, ..., \ell_n, v')$ *if* $a \in N$ *and there is an edge*
  $(\ell_i \xrightarrow{g,a,r} \ell'_i) \in E_i$ *in the* i*th component automaton such that*
  $v \models g, v' = v[r]$ *and*
  $v' \models I_i(\ell'_i) \wedge \bigwedge_{k \neq i} I_k(\ell_k)$;
- $(\ell_1, ..., \ell_i, ..., \ell_j, ..., \ell_n, v) \xrightarrow{\tau} (\ell_1, ..., \ell'_i, ..., \ell'_j, ..., \ell_n, v)$ *if* $i \neq j$ *and there are edges* $(\ell_i \xrightarrow{g_i,\alpha,r_i} \ell'_i) \in E_i$ *and* $(\ell_j \xrightarrow{g_j,\beta,r_j} \ell'_j) \in E_j$ *such that* α *and* β *are complementary,*
  $v \models g_i \wedge g_j, v' = v[r_i \cup r_j]$, *and*
  $v' \models I_i(\ell'_i) \wedge I_j(\ell'_j) \wedge \bigwedge_{k \neq i,j} I_k(\ell_k)$;
- $(\ell_1, ..., \ell_n, v) \xrightarrow{d} (\ell_1, ..., \ell_n, v + d)$ *for all* $d \in \mathbb{R}_{\geq 0}$ *such that*
  $$v + d' \models \bigwedge_{i \in \{1,...,n\}} I_i(\ell_i)$$
  for each real number d′ in the interval [0, d].

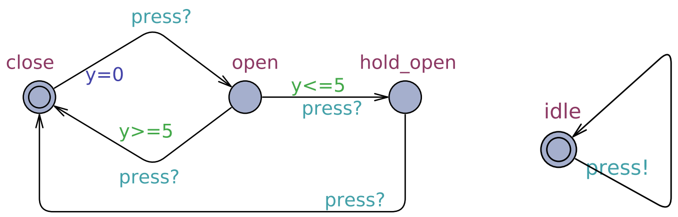
Let v_0 *denote the valuation* $v_0(x) = 0$ *for all* $x \in C$. *If* v_0 *satisfies the invariants of all the initial locations* ℓ_0^i *, we shall call* $(\ell_0^1, \ell_0^2, ..., \ell_0^n, v_0)$ *the initial state (or initial configuration) of T(A).*

Where *N* is a finite set of ordinary action names including τ formally:

$$Act = \{c!|c \in Chan\} \cup \{c?|c \in Chan\} \cup N$$

2.1.1 Timed Automata in UPPAAL

UPPAAL is a tool for the verification of real-time system. The tool, as described in the tutorial (BDL04), is designed to verify systems modelled as networks of timed automata extended with additional features, such as bounded integer variables and urgency. A current state of the system is defined by the locations, the clock values, and the values of the variables of the automata. Every automaton may fire an edge (sometimes misleadingly called a transition) separately or synchronise with another automaton, which leads to a new state. Figure 3 shows a network of timed automata describing the interaction between two automata modelling an user with an automatic door. The door in Figure 3 (a) is modelled with three locations: close, open, and hold_open. If the user presses



(a) Door (b) User

Figure 3: The door example

a button, i.e., synchronises with `press?`, then the door is opened. If the user presses the button again, the door is closed. However, if the user is fast and rapidly presses the button twice, the door is opened and then blocked in open position. The user model is shown in Figure 3 (b). The user can press the button randomly at any time or even not press the button at all. The clock y of the door is used to detect if the user was fast ($y < 5$) or slow ($y >= 5$).

We present below the UPPAAL modelling language based on timed automata and later the query language, that is a subset of TCTL, used for specifying property to be checked. For having a more complete view of all the UPPAAL constructs, we refer the interested reader to the Appendix A.

Templates are the skeleton of automata and are defined with a set of parameters that can be of any type (e.g., int, chan). These parameters are substituted for a given argument in the process declaration.

Constants are declared as `const` followed by a `name` and a `value`. Constants by definition cannot be modified at runtime.

Bounded integer variables are declared as `int[min,max] name`, where min and max are the lower and upper bound, respectively. The bounds are checked upon verification and violating a bound in guards, assignments or invariants leads to an invalid state that is discarded (at runtime). If the bounds are omitted, the default range of -32768 to 32768 is used.

Binary synchronisation channels are declared as `chan name`. An edge labelled with an output action `c!` synchronises with another labelled input action `c?`. A synchronisation pair is chosen non-deterministically if several combinations are enabled.

Broadcast channels are declared as `broadcast chan`. In a broadcast synchronisation one sender `c!` can synchronise with an arbitrary number of receivers `c?`. Any receiver than can synchronise in the current state must do so. The send on a broadcast channel is not a blocking action. If there are no receivers, then the sender can still execute the `c!` action.

Urgent synchronisation channels are declared by prefixing the channel declaration with the keyword `urgent`. Delays must not occur if a synchronisation transition on an urgent channel is enabled. Edges using urgent channels for synchronisation cannot have time constraints, i.e., no clock guards. Time is not allowed to pass when the system is in an urgent location.

Committed locations are even more restrictive on the execution than urgent locations. A committed location does not permit the time elapsing and has the priority on other executions. Usually it is used for creating atomic sequence in the execution between more than two components.

Arrays are data structures, like in classic programming languages, where type can be clocks, channels, constants and integer variables. They are defined by appending a size to the variable name, e.g. `chan c[4]; clock a[2]; int[3,5] u[7]`.

Initialisers are used to initialise integer variables and arrays of integer variables. For instance, `int i = 2;` or `int i[3] = {1, 2, 3}`.

Record types are declared with the `struct` keyword and permit to create a structure composed of different types. e.g. `struct  int a; bool b;  s1 = { 2, true };`.

Custom types are defined with the C-like `typedef` construct. You can define any custom-type from other basic types such as records.

User functions are defined either globally or locally to templates. Function are callable from the update in edges or in invariant if the return type is `bool`.

Expressions in UPPAAL

Expressions in UPPAAL are defined on clocks and integer variables. Expressions can be setted using the dedicated text-box contained in the property form of edges.

- **Select** A select label contains a comma separated list of `names: type` expressions where `name` is a variable name and `type` is a defined type (built-in or custom). These variables are accessible on the associated edge only and they will take a non-deterministic value in the range of their respective types.

- **Guard** A guard is a particular expression side-effect free (i.e. expressions that does not change the value of variables or the state of the system); it evaluates to a boolean and is defined only on clocks, integer variables, and constants. Guards over clocks are essentially conjunctions (disjunctions are allowed over integer conditions).

- **Synchronisation** A synchronisation label is either of the form *Expression!* or *Expression?*, or is an empty label.

- **Update** An update label is a comma separated list of expressions with a side-effect; expressions must only refer to clocks, integer variables, and constants. They may also call functions.

- **Invariant** An invariant is an expression bound to a state that satisfies the following conditions: it is side-effect free; only clock, integer variables, and constants are referenced. It is a conjunction of conditions of the form `x<e` or `x`$\leqslant$`e` where `x` is a clock reference and evaluates to an integer. An invariant may call a side-effect free function that returns a boolean.

2.1.2 UPPAAL query language

UPPAAL uses a simplified version of the TCTL logic for verifying if the model under analysis satisfies a set of requirements. Like in TCTL, the UPPAAL query language consists in path formulae and state formulae.

State formulae describe properties on an individual state, whereas path formulae quantify over paths or traces. The path formulae are classified according to the quantifier and generally can express *reachability*, *safety* and *liveness* properties.

State formula A state formula is an expression that describes a particular condition on a state. It can be a boolean evaluation or the internal state of a process, using the notation `Process.location`. Another important expression on the state formula is the deadlock that is expressed with the keyword `deadlock`; a system is in deadlock if there are no outgoing action transitions for the current state and neither a delay on that state is possible.

Path formula The path formulae are divided in three subcategories according to the property to verify:

- The **reachability** is the simplest form and checks if there exists a path that eventually satisfies a certain property. This formula is also used for validating the basic behaviour of the model. In UPPAAL, we write this property using the syntax $E <> \varphi$.
- The **safety** property is used for avoiding that something bad will happen. For example in a critical domain a bad action must be always avoided. In UPPAAL these properties are formulated positively: a formula should be true in all reachable states with the path formula $A[\,]\varphi$ or $E[\,]\varphi$.
- The **liveness** property expresses the willingness that something good will eventually happen. The liveness in UPPAAL is expressed with the path formula $A <> \varphi$.

2.1.3 The UPPAAL tool

UPPAAL is composed of two main parts: the interface and the model checking engine. The graphical interface that is shown in Figure 4 is

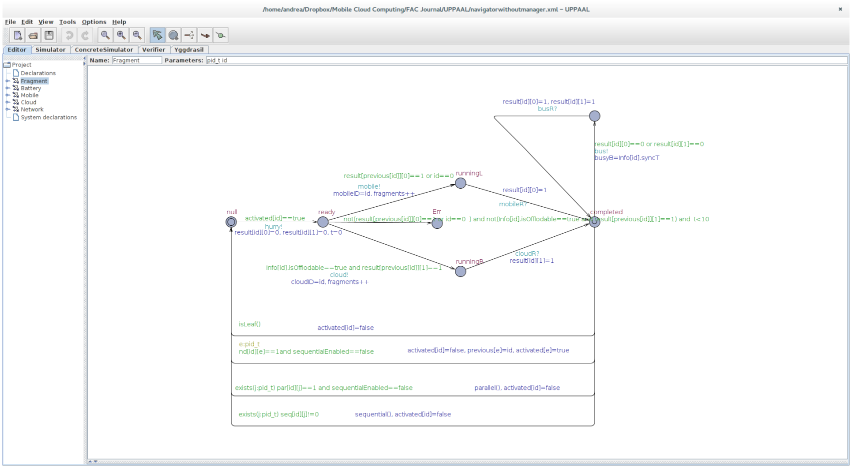

/home/andrea/Dropbox/Mobile Cloud Computing/FAC Journal/UPPAAL/navigatorwithoutmanager.xml - UPPAAL
File Edit View Tools Options Help
Editor Simulator ConcreteSimulator Verifier Yggdrasil
Project
Declarations
Fragment
Battery
Mobile
Cloud
Network
System declarations
Name: Fragment
Parameters: pid_t id
result[id][0]=1, result[id][1]=1
busR?
result[id][0]==0 or result[id][1]==0
bus!
busyB=Info[id].syncT
result[previous[id]][0]==1 or id==0
runningL
mobile!
mobileID=id, fragments++
result[id][0]=1
mobileR?
null
activated[id]==true
hurry!
result[id][0]=0, result[id][1]=0, t=0
ready
Err
not(result[previous[id]][0]==1 or id==0) and not(Info[id].isOfflodable==true and result[previous[id]][1]==1) and t<10
completed
cloudR?
result[id][1]=1
runningR
Info[id].isOfflodable==true and result[previous[id]][1]==1
cloud!
cloudID=id, fragments++
isLeaf()
activated[id]=false
e:pid_t
nd[id][e]==1and sequentialEnabled==false
activated[id]=false, previous[e]=id, activated[e]=true
exists(j:pid_t) par[id][j]==1 and sequentialEnabled==false
parallel(), activated[id]=false
exists(j:pid_t) seq[id][j]!=0
sequential(), activated[id]=false

Figure 4: UPPAAL editor

divided in three main parts: the editor, the simulator, and the verifier, accessible via three tabs under the main toolbar.

The editor contains the system model, which usually is composed by a network of timed automata that are called processes in the tool. The editor is divided in two parts: the one on the left is the tree panel where is shown the system structure and permits to access the different templates and declarations; on the right hand side instead there is the canvas graphical editor. Figure 4 shows a fragment template composed by seven locations and some edges. The edges are labelled with guard conditions (e.g. `activated[id]==true`), synchronisations (e.g. `hurry!`) and assignments (e.g. `result[id][0]=0`). In the tree panel is possible to find the global declaration, that contains variables, clocks, channels and constants. Fragment, Battery, Mobile, Cloud and Network are example of templates and usually are equipped with a local declaration of variables that is restricted to the template scope. The system declaration contains the processes instantiation, that can be used to parametrize template or directly invoked.

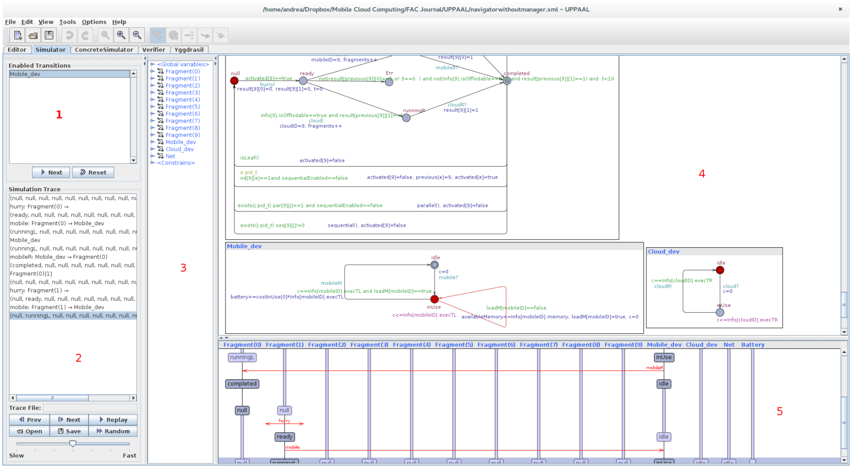
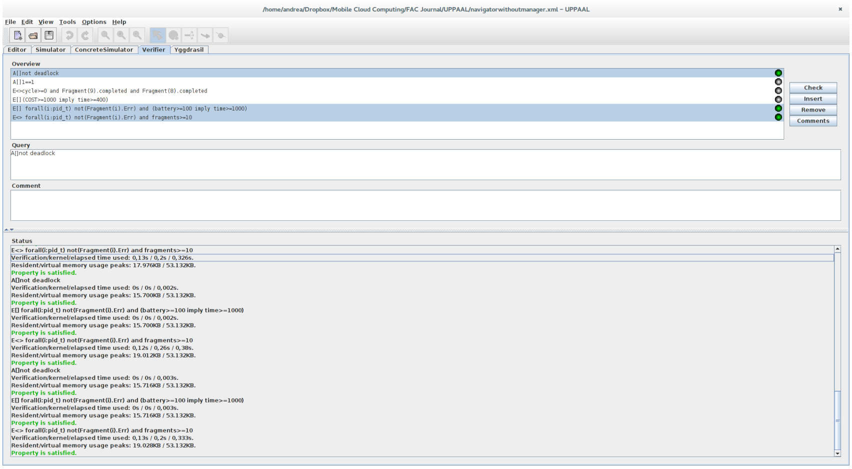

/home/andrea/Dropbox/Mobile Cloud Computing/FAC Journal/UPPAAL/navigatorwithoutmanager.xml - UPPAAL
File Edit View Tools Options Help
Editor
Simulator
ConcreteSimulator
Verifier
Yggdrasil
Enabled Transitions
Mobile_dev
1
Next
Reset
Simulation Trace
2
Trace File:
Prev
Next
Replay
Open
Save
Random
Slow
Fast
3
4
5
Mobile_dev
Cloud_dev
Fragment(0) Fragment(1) Fragment(2) Fragment(3) Fragment(4) Fragment(5) Fragment(6) Fragment(7) Fragment(8) Fragment(9) Mobile_dev Cloud_dev Net Battery

Figure 5: UPPAAL simulator

The Simulator, represented in Figure 5, is used for simulating a singular instance of the system. There are two ways of interaction, the first one is the manual selection of the next action to execute, or the other one is the automatic selection performed by the tool, which randomly chooses between the enabled actions.

The simulator view shows five sub-windows. In the window 1 there is the *enabled trasition* where it is possible to choose the next transition. In the window 2, just below, there is the *simulation trace*, that shows the current state for each process after the execution of an action. The window 3 is the *variable view*, it shows the current values of variables and clocks distinguished between global one and local imputable to each process. The window 4 shows the automata state in a graphical way. The window 5, called *sequence chart* shows the synchronization between the processes.

The verifier is the last part of the graphical interface (Figure 6) where the user can verify properties expressed in the query box and selectable in the overview list. The *status panel* is used to visualize the results of the



```
A[]not deadlock
A[]1==1
E<>cycle>=0 and Fragment(9).completed and Fragment(8).completed
E[](COST>=1000 imply time>=400)
E[] forall(i:pid_t) not(Fragment(i).Err) and (battery>=100 imply time>=1000)
E<> forall(i:pid_t) not(Fragment(i).Err) and fragments>=10
```

```
A[]not deadlock
```

```
E<> forall(i:pid_t) not(Fragment(i).Err) and fragments>=10
Verification/kernel/elapsed time used: 0,13s / 0,2s / 0,326s.
Resident/virtual memory usage peaks: 17.976KB / 53.132KB.
Property is satisfied.
A[]not deadlock
Verification/kernel/elapsed time used: 0s / 0s / 0,002s.
Resident/virtual memory usage peaks: 15.700KB / 53.132KB.
Property is satisfied.
E[] forall(i:pid_t) not(Fragment(i).Err) and (battery>=100 imply time>=1000)
Verification/kernel/elapsed time used: 0s / 0s / 0,002s.
Resident/virtual memory usage peaks: 15.700KB / 53.132KB.
Property is satisfied.
E<> forall(i:pid_t) not(Fragment(i).Err) and fragments>=10
Verification/kernel/elapsed time used: 0,12s / 0,26s / 0,38s.
Resident/virtual memory usage peaks: 19.012KB / 53.132KB.
Property is satisfied.
A[]not deadlock
Verification/kernel/elapsed time used: 0s / 0s / 0,003s.
Resident/virtual memory usage peaks: 15.716KB / 53.132KB.
Property is satisfied.
E[] forall(i:pid_t) not(Fragment(i).Err) and (battery>=100 imply time>=1000)
Verification/kernel/elapsed time used: 0s / 0s / 0,003s.
Resident/virtual memory usage peaks: 15.716KB / 53.132KB.
Property is satisfied.
E<> forall(i:pid_t) not(Fragment(i).Err) and fragments>=10
Verification/kernel/elapsed time used: 0,13s / 0,2s / 0,333s.
Resident/virtual memory usage peaks: 19.028KB / 53.132KB.
Property is satisfied.
```

Figure 6: UPPAAL verifier

query and the communication with the server. The green/red semaphore near each property in the *overview* window indicates the result coming out from the verification phase activated by selecting the property and pushing the check button. The verification can be executed also with the trace generation settable from the option menu and by selecting the diagnostic trace of interest. At the end of the verification with trace, the system will ask the willingness to import the resulting trace on the simulator to allow further check to the user.

2.2 Domain Specific Languages

A Domain Specific Language (DSL), according to (Bet13), is a small language, developed on purpose for a particular application domain. People find DSLs valuable because allow the programmer to increase his productivity, improve the communication with domain experts and do not require strong programming notions. Of course the DSL is not a substitute of a general purpose language like Java or C. But it permits solving problems easier and faster on a particular domain with respect to traditional programming language. Sometimes it is not clear if it is more convenient to use a general language or a new DSL, for example may you want to introduce a new language for describing data of a model or application and someone else maybe prefer XML. Of course this is up to the developer, but the objective fact is that the XML is a good machine readable language but not human-readable; indeed XML tends to be verbose, and it fills documents with too much syntax noise.

Implementing a new DSL involves a series of procedures that allow the machine to read the text written in that DSL, succeed in the parsing phase, process and interpret it or generate the code in another language. This procedure is not always the same, but most of the listed phases are typical of all implementations. Usually a language is decomposable in single atomic elements. These elements can be a *keyword* such as the word *class* in Java, an *identifier* such as a class name, a *symbol name* such as a variable name; and all other elements are *literals*. The process of decomposing a sequence of characters into atomic elements is called **lexical analysis** and the phase that checks for each statement that it is respecting the syntax structure expected by the language is called **parsing**. The duty of the parser is to check if the given program respect a grammar. A **grammar** is a set of rules that describe the form of the constructs that are valid according to the language syntax. The parsing mechanism does not guarantee the correctness of a program, this because for example the type checking cannot be done during the parsing but is part of the semantics analysis.

A good success for a DSL is given also by a IDE support. A large range of available features for the language generation permit an easier access to the final user, increasing his opinion on the DSL use. Among the most important features it is possible to find the **syntax highlight**, that is the ability of colouring and formatting keywords in the language for giving the right visibility and an immediate feedback on the structure of the program. The **background parsing**, that checks continuously the program syntax to revel as soon as possible error; this implies less effort for fixing the problem in terms of time and costs. **Error markers** have the duty of showing the right point in the language where the error happened in order to stand out the right point that need to be fixed. **Content assistant** is the auto complete mechanism that helps the programmer to remember commands or just the right sequence of parameters. **Hyperlinking** permits to navigate between references in a program, for example from a variable to its declarations or from a method call to its declaration. **Quickfixes** are fixes automatically suggested by the IDE to the programmer, for example, given a method that does not exist, two possible quick fixes could be to create a new method or simply correct the name generated by a typing error.

2.2.1 The Xtext tool

Xtext covers all aspects of a complete language infrastructure, from parsers, to compiler or interpreter and other useful tools. Xtext comes with great default solutions for all these aspects which at the same time can be easily tailored to developer individual needs. The host environment for Xtext is Eclipse, a free software distributed under the Eclipse public license. It is an Integrated Development Environment(IDE) based on multi-language and multi-platform paradigm. It offer a complete IDE for the Java language and rely on a plug-in system. In Eclipse using the traditional Xtext grammar language is possible to describe the syntax of a new DSL. The specialty for JVM languages is the possibility to inherit from an abstract grammar Xbase, which predefines the syntax for the reusable parts. It is not needed to use all of them directly and of course is possible to change the syntax or add new concepts, as it seems fit. Xtext can be simply installed from the Eclipse marketplace, once installed, it is possible to create a new project just following the Eclipse wizard for the Xtext Project creation.

Following the wizard the system will create a new project

```
Project name: org.xtext.example.mydsl
Language name: org.xtext.example.mydsl.MyDsl
Language Extensions: mydsl
```

In the workspace will compare four folders as follows

1. org.xtext.example.mydsl contains the grammar definition and all runtime components (parser, lexer, linker, validator, scope, etc.);
2. org.xtext.example.mydsl.tests contains the unit tests;
3. org.xtext.example.mydsl.ui contains the eclipse editor and the related functionalities;
4. org.xtext.example.mydsl.sdk contains the features of the project.

The wizard will automatically open the grammar file MyDsl.xtext in the editor, that is usually contained in the org.xtext.example.mydsl part under the src folder.

```
grammar org.xtext.example.mydsl.MyDsl with org.eclipse.
    xtext.common.Terminals

generate myDsl "http://www.xtext.org/example/mydsl/MyDsl"

Model:
greetings+=Greeting*;

Greeting:
'Hello' name=ID '!';
```

The first rule in the grammar is usually the start rule and coincides with the root of the syntax three. In our example the model is composed by an arbitrary number (any operator *) of greetings which will be added (+=) to a feature called elements. The Greeting rule start with the keyword *Hello* followed by an identifier, which is parsed by a rule called ID, and by symbol !. The rule ID is a kind of terminal rule literal and it is in the org.eclipse.xtext.common.Terminals package. The value returned by the call to ID is assigned to the feature name.

Generate Language Artifacts

Once defined the grammar of the language it is possible to execute the code generator that will derive the various language components. To do so, once located the file *GenerateMyDsl.mwe2* next to the grammar in the package explorer view. From this context menu, it is sufficient to choose *Run As → MWE2 Workflow.* This invocation triggers the Xtext language generator, producing the parser, the serializer and some additional infrastructure code. The described process terminates with some logging messages in the Console View and the keyword ”Done”.

Run the Generated IDE Plug-in

In order to test the IDE integration is necessary to select *Run → Run Configurations* from the Eclipse menu and choose *Eclipse Application → Launch Runtime Eclipse*. This preconfigured launch shortcut has the appropriate memory values and parameters already setted. The system, after the click on the run button starts a new Eclipse workbench with the newly developed plug-ins installed. In the new workbench, is possible to create a new project following the *File → New → Project → Java Project* procedure and therein a new file with the file extension specified in project creation (**.mydsl*). This last step will open the adequate entity editor, where is possible to exploit the default functionalities for code completion, syntax highlighting, syntactic validation, linking errors, the outline view, find references, etc.

The final DSL implemented is able to accept the language composed by the Keyword ”Hello” followed by an identifier and closed with the symbol ”!”.

```
Hello Marius!
Hello Andrew!
...
```

An improvement in the presented example could be the introduction of a validator. To achieve this task in Xtext there are two ways, the first one is going directly in the *GenerateMyDsl.mwe2* file and uncomment the `NamesAreUniqueValidator` line. With this new line, the system checks if the language contains multiple rules with the same identifier, if yes, an error alert for definition ambiguity is raised.

```
// Xtend-based API for validation
fragment = validation.ValidatorFragment auto-inject {
//composedCheck = "...validation.ImportUriValidator"
composedCheck = "org.eclipse.xtext.validation.
    NamesAreUniqueValidator"
         }
```

Of course this is the simplest validation that can be automatically done by the tool, more advanced check can be directly implemented in the *MyDslValidator.xtend* in the `org.xtext.example.mydsl.validation` package. Possible checks are related to Cross-link or custom syntax validation, both of them can be alerted to the user with error or warning messages.

Java Code Generation

As soon as the generation of the Xtext artifacts for a grammar, a code generator stub will be put into the runtime project of the language. This allows to automatically generate code rewritten in Java or any other language derived from the DSL elements.

2.2.2 Grammar language

The grammar language in Xtext is specifically designed for developing domain specific languages. The idea is to describe the syntax of the DSL using the given grammar. The grammar is composed by *terminal rules*, that are sequences of characters that are atomic symbols, informally called token rules. The ID rule is the most important terminal rule, it is usually composed by a sequence of letters or numbers in any order. In the definition of the grammar, *data type rules* are preferred to terminal one, this because using types is possible to avoid the ordering that is instead mandatory in the terminal one. The *return type* is present in each terminal rule and it is assumed that is of the form **`ecore::EString`**. A different type is necessary to specify other types; for example INT for the willingness to return an integer value.

The terminal rules has a cardinality that it is described using the Extended Backus-Naur Form. There are four different cardinalities:

1. exactly one (no operator)

2. zero or one (operator ?)

3. any (zero or more, operator *)

4. one or more (operator +)

A *parser rule* does not produce a single atomic terminal token but a tree of non terminal tokens. These rules are also called EObject rules because they are handled as a plan for the creation of a semantic model (Abstract Syntax Tree AST).

Not all the expressions that are available in terminal rules can be used in parser rules. Character ranges, wildcards, the until token and the negation as well as the EOF token are only available for terminal rules. For the parser rules the admitted elements are groups, alternatives, keywords and rule calls.

Chapter 3

Domain Specific Language MobiCa

In this chapter 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 components, 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 expressed as a network of timed automata (see Section 2.1), thus supporting the verification of properties 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 each other;

SYSTEMS:		
N	$::=$	$(c, e_m, e_c, e_b)(b, n, m) \triangleright \tilde{A} \quad \vert \quad c \triangleright \tilde{A} \quad \vert \quad Sys_1 \vert Sys_2$
APPLICATIONS:		
A	$::=$	$\langle \tilde{F}; S \rangle$
FRAGMENTS:		
F	$::=$	$f[i, m, s, o]$
STRUCTURE:		
S	$::=$	$f_1 \; Op \; \tilde{f_2} \quad \vert \quad S_1 \, ; \, S_2$
OPERATORS:		
Op	$::=$	$\longrightarrow \quad \vert \quad \dashrightarrow \quad \vert \quad \twoheadrightarrow$

Table 1: MobiCa syntax

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 environmental 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.

3.1 Language syntax

The syntax of the MobiCa language is defined by the BNF grammar given in Table 1. 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 and cloud machines (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 $(c, e_m, e_c, e_b)(b, n, m) \triangleright \tilde{A}$ can be seen as a container of applications $\tilde{A}$ characterized by some operational information. The static information is: the computational power of the device c (expressed in terms of number of instructions that can be executed per second), the energy consumed per time unit by the mobile device when it is in use e_m, when it is in idle e_c and during the synchronization along the bus e_b. The information that instead may change at runtime is: 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 $f_1 \twoheadrightarrow f_2, f_3;\ f_1 \dashrightarrow f_4, f_5$, 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 executed 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 1 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 5 (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).*

Below we use MobiCa to model a simple scenario where an optimal infinite scheduling is necessary for minimizing energy consumption and improving system performance. A more realistic scenario will be discussed later on in section 3.4.1

Example 1 (A simple application) *This scenario is inspired by one from (GK99). It concerns a simple MCC application, whose graphical representation is shown in Figure 7 (right-hand side). The application is composed of three fragments, f_0, f_1 and f_2, connected by means of the non-deterministic operator ($\longrightarrow$) and by the sequential operator ($\dashrightarrow$). Since the application behavior is deterministic in this case, the unique run is composed by an initialization phase $f_0 \rightarrow f_2$, followed by an infinite loop $f_2 \rightarrow f_0 \rightarrow f_2 \rightarrow f_1 \rightarrow f_2$. The fragment f_0 can be executed only locally, instead the fragments f_1, f_2 can be executed either on the mobile or in the cloud, with the only requirement of maintaining the data consistent. For consistency we intend that either a fragment is executed on the same location of its predecessor or at a different location only after the result of the predecessor has been synchronized.*

Application structure:

$$\boxed{\begin{array}{lll} f_0 & \longrightarrow & f_2; \\ f_2 & \dashrightarrow & (f_0, f_1); \\ f_1 & \longrightarrow & f_2 \end{array}}$$

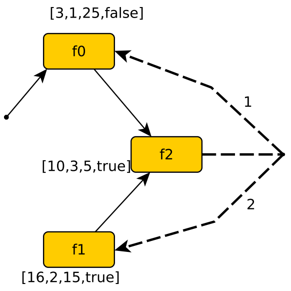


Figure 7: A simple example of a MobiCa application

In the figure, the fragments are annotated with 4 parameters; in order, we have: the execution time on the mobile device (given by the number of instructions divided by the mobile computation power, i.e. i/c), the execution time on the cloud, the synchronization time of the results on the bus (given by s/n) and a boolean value representing the offloadability of the fragment (a false value indicates that only the local execution is admitted, as in the case of f_0). The graphical notation in Figure 7 is formalized in terms of the so-called System Graph in Definition 6. Notably, the memory parameters of MobiCa systems are not considered in this specific formalization.

A schedule for the simple application shown in Figure 7 should provide a sequence of execution choices (i.e. local vs remote) for each of the three fragments between the available resources. A schedule is optimal if the total execution time or energy cost is minimum, considering that the energy consumption per time unit for the mobile device is 5 when it is in use, 1 in the idle state, and 2 for the synchronization.

The Gantt chart in Figure 8 depicts three possible schedules for the proposed example application. For each of them, we indicate the location of execution between mobile(M) and cloud(C), and the use of the bus(B). The values of T and E at the end of the sequence are the time and the energy required by the schedule for computing a complete loop cycle. In the first schedule, the computation is maintained locally for all fragments; this behavior is reasonable when the network is not available. Another approach might be to maintain the computation locally only for the non-offloadable fragments (in our case only f_0) and try to move the computation remotely as soon as possible; this allows one to manage the task congestions in the mobile device. The third schedule instead takes into consideration the sequence of offloadable fragments and executes the computation remotely only when the synchronization of data is minimal.

3.2 TA-based semantics

We describe here the semantics of MobiCa, given in terms of a translation to networks of Timed Automata (TA). Such a semantics can indeed be used to solve the previously described scheduling problem, by resorting to the analysis facilities provided by the UPPAAL model checker.

The translation is divided in two parts: the *passive* part, which focusses on resources, and the *active* one, which focusses on the applications. Thus, the TA corresponding to a given MobiCa system is the composition of the results of the passive and active translations merged to-

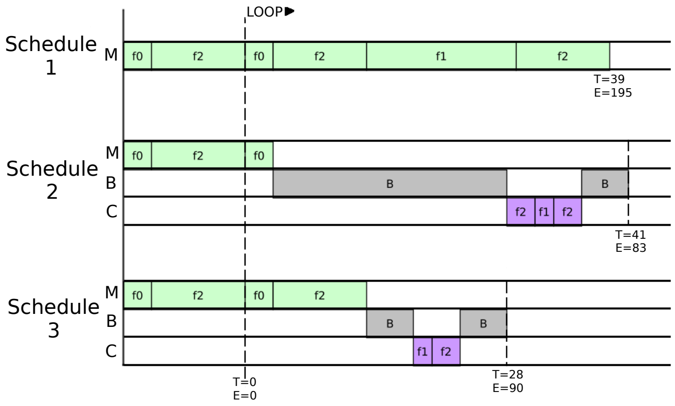


Figure 8: Schedules for the simple application

gether by means of a global declaration. Below we describe the details of the translation in terms of UPPAAL models.

3.2.1 Global declaration

The global declaration consists of all the shared variables used for the synchronization of fragments, clocks for keeping track of time, and variables stating the internal state of the resources. In the global declaration we find also the structure S of the application declared as an adjacency matrix. A structure consists of three $n \times n$ matrices, one for each transition operator, where *n* is the length of the tuple $\tilde{F}$. Let m_{ij} be the (*i*,*j*) entry of a matrix, we have $m_{ij} \geqslant 1$ if the i^{th} and j^{th} fragments are connected, and 0 otherwise. Notably, the diagonal of each matrix is always zero, as self-loops are not admitted. Table 2 shows the corresponding three adjacency matrices, related to the Example 1 shown in Figure 7. In particular, we have:

($\longrightarrow$): the non-deterministic transition for fragment f_i is activated if the i^{th} row has non-zero cells, and the next fragment to be activated is non-deterministically selected in $\{f_j \mid m_{ij} = 1\}$;

$\longrightarrow$	f_0	f_1	f_2
f_0	0	0	1
f_1	0	0	1
f_2	0	0	0

$\twoheadrightarrow$	f_0	f_1	f_2
f_0	0	0	0
f_1	0	0	0
f_2	0	0	0

$\dashrightarrow$	f_0	f_1	f_2
f_0	0	0	0
f_1	0	0	0
f_2	1	2	0

Table 2: Operators translation

($\twoheadrightarrow$): the parallel transition is similar to the non-deterministic one, with the difference that the fragment f_i activates all the fragments f_j with $m_{ij} = 1$;

($\dashrightarrow$): the sequential operator matrix is slightly different from the previous ones, as the values are not only 0 or 1. These values must be interpreted as a sequence defining the order in which the target fragments are activated for each execution of the source fragment. The activation of the sequential operator on a fragment excludes the other operators until the sequence of activation is terminated. In our example, fragment f_2 activates first the execution of f_0 and then the execution of f_1 (see the last row of the matrix at the right-hand side in Table 2).

3.2.2 Fragments

The TA for a generic fragment is depicted in Figure 9; the template is parametric, so that it is a valid representation for each fragment of the application. The execution of the fragment starts from the initial location where it is waiting for the activation. The activation is managed by the array *activated[]* as follows: whenever the element in the array corresponding to the fragment index becomes *true*, the corresponding fragment can move to the *ready* location. In this latter location, it can continue its execution on the mobile device or the cloud, depending on the evaluation of the guards on the transitions. They state that the fragment can be executed locally only if the results of the previous fragment are updated locally (*result[previous[id]][0]==1*), or remotely only if they are updated remotely and the fragment is offloadable (*result[previous[id]][1]==1 and*

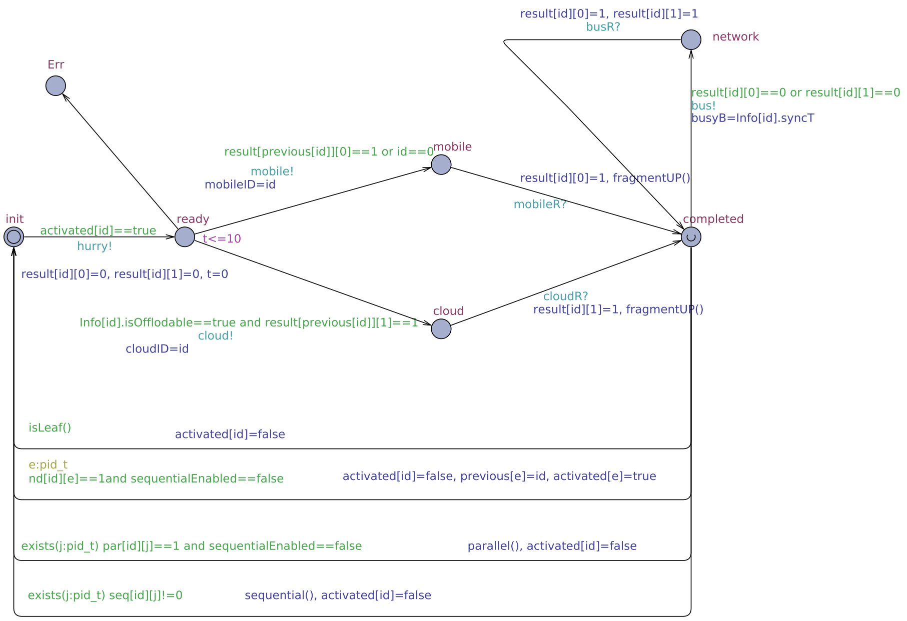

result[id][0]=1, result[id][1]=1
busR?
network
Err
result[id][0]==0 or result[id][1]==0
bus!
busyB=Info[id].syncT
result[previous[id]][0]==1 or id==0
mobile
mobile!
mobileID=id
result[id][0]=1, fragmentUP()
mobileR?
init
activated[id]==true
hurry!
ready
t<=10
completed
result[id][0]=0, result[id][1]=0, t=0
cloudR?
result[id][1]=1, fragmentUP()
cloud
Info[id].isOfflodable==true and result[previous[id]][1]==1
cloud!
cloudID=id
isLeaf()
activated[id]=false
e:pid_t
nd[id][e]==1and sequentialEnabled==false
activated[id]=false, previous[e]=id, activated[e]=true
exists(j:pid_t) par[id][j]==1 and sequentialEnabled==false
parallel(), activated[id]=false
exists(j:pid_t) seq[id][j]!=0
sequential(), activated[id]=false

Figure 9: Fragment translation

Info[id].isOffloadable==true). In case of both the invariants are false, the model will move in the `error` state. When the execution of the fragment is completed, it can proceed towards either the *network* location, in order to synchronize the results locally and remotely (*result[id][0]=1, result[id][1]=1*), or the initial location, by following one operator in the structure. Indeed, the use of each operator is rendered as an outgoing transition from the *completed* location to the *init* one; these transitions are concurrent and enabled according to the corresponding adjacency matrix, defined in the global declaration.

3.2.3 Resources

Each kind of resource (i.e., mobile device, cloud and bus) is translated into a specific TA; since these TA are similar, we show here just the one for the mobile device (Figure 10) and, for the sake of presentation, we describe it in terms of a general resource. A resource can be in the *idle* state, waiting for some fragment, or *inUse*, processing the current fragment. When the resource synchronizes with a fragment, it resets the lo-

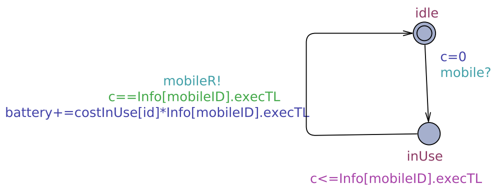


Figure 10: Mobile resource translation

cal clock and remains in the *inUse* state until the clock reaches the value corresponding to the occupation time for the current fragment. Before releasing the resource, the battery level of the mobile device is updated according to the permanence time and the energy consumed by the resource. In this model, we assume that no energy is consumed if there is nothing to compute, and the energy power consumed by the cloud during its execution corresponds to the energy used by the mobile in the idle state waiting for the results.

3.3 Optimal scheduler

In this section, we formalize the notion of optimal scheduler in terms of two cost functions on a System Graph (SG). A SG, like the one in Figure 7, provides a graphical representation of a MobiCa application. Besides this, it is also useful as an intermediate model between the specification and the resulting network of TA generated by the translation.

Definition 6 (System Graph) *Given an application $\langle \tilde{F}; S \rangle$ installed in a system with a mobile device defined by information (c, e_m, e_c, e_b) and a cloud machine defined by c', its* system graph *SG is a tuple $\langle N, \longrightarrow, \dashrightarrow, \twoheadrightarrow, T, E, O \rangle$ where:*

- *$N = \{f \mid f[i, m, s, o] \in \tilde{F}\}$ is a set of fragment names.*

- $\longrightarrow$, $\dashrightarrow$, $\twoheadrightarrow \subseteq N \times N$ *are three transition relations defined as* $f \longrightarrow f'$ *(resp.* $f \dashrightarrow f'$, $f \twoheadrightarrow f'$*) iff* $f' \in \tilde{f}$ *for some* $\tilde{f}$ *such that* $f \longrightarrow \tilde{f} \in S$ *(resp.* $f \dashrightarrow \tilde{f} \in S$, $f \twoheadrightarrow \tilde{f} \in S$*). We use* $f \rightarrowtail f'$ *to denote either* $f \longrightarrow f'$ *or* $f \dashrightarrow f'$ *or* $f \twoheadrightarrow f'$.

- $T : N \times \{M, C, B\} \rightarrow \mathbb{N}$ *gives the execution time of a fragment on a resource (where M is the mobile device, C the cloud, and B the bus); given* $f[i, m, s, o] \in \tilde{F}$*, we have:* $T(f, M) = \lfloor i/c \rfloor$, $T(f, C) = \lfloor i/c' \rfloor$, *and* $T(f, B) = \lfloor s/n \rfloor$.

- $E : \{M, C, B\} \rightarrow \mathbb{N}$ *is the energy, expressed as a natural number, consumed per time unit by the mobile device when a given resource is in use (E(M)=*e_m*, E(C)=*e_c*, E(B)=*e_b*).*

- $O : N \rightarrow \{false, true\}$ *represents the possibility of offloading a fragment (value 1) or not (value 0); given* $f[i, m, s, o] \in \tilde{F}$*, we have* $O(f) = o$.

Notably, an SG is completely derived from the information specified in the corresponding MobiCa system.

A *path* on SG is a finite sequence $\eta = f_0 \rightarrowtail f_1 \rightarrowtail \ldots \rightarrowtail f_k$ ($k \geq 0$). Notably, in a path, parallel activations of fragments are interleaved.

Definition 7 (Scheduler) *Given a system graph SG, a* scheduler *is a partial function* $\Theta : N \times Op \times N \rightarrow \{0, 1\}^2$ *that, given a transition* $f \rightarrowtail f'$ *in SG, returns a pair of values* $\pi_s, \pi_t \in \{0, 1\}$ *which indicate the execution location of the source fragment* f *and of the target fragment* f'*, respectively, where* 0 *denotes a local execution and* 1 *a remote one.*

When a scheduler is applied to a transition of the corresponding SG, it returns information about offloading decisions for the involved fragments. By applying a scheduler Θ to each transition of a sequence of transitions, i.e. a path η, we obtain a *schedule* δ, written $\Theta \cdot \eta = \delta$. A schedule consists of a sequence of triples of the form (f, π, β), each one denoting a fragment f, belonging to the considered path, equipped with its execution location π and the synchronization flag β. Parameters $\pi, \beta \in \{0, 1\}$ indicate the local ($\pi = 0$) and remote ($\pi = 1$) execution and the need ($\beta = 1$) or not ($\beta = 0$) of data synchronization for f. The value of π should respect the offloadability declared by the variable O in the fragment declaration. Formally, $\Theta \cdot \eta = \delta$ is defined as follows: let f and

f' being two consecutive fragments in the path η, there exist in δ two consecutive triples $(f, \pi, \beta) \rightarrowtail (f', \pi', \beta')$ iff $\Theta(f, \rightarrowtail, f') = (\pi_s, \pi_t)$ s.t. $\pi = \pi_s$, $\pi' = \pi_t$ and $\beta = |\pi_s - \pi_t|$. Notice that, as Θ is a partial function, there may be transitions in η that are not in δ; for such transitions the schedule does not provide any information about the offloading strategy to apply, because they are not considered by the scheduler.

Taking inspiration from the approach in (CBC+10a), we define two cost functions. In particular, we consider the cost of executing a given path in the considered SG using the scheduler, i.e. the cost functions are defined on schedules. We describe our approach for determining such optimal schedulers in Chapter 4

Definition 8 (Time and energy costs) *The time and energy costs of a schedule δ for a given $SG = \langle N, \longrightarrow, \dashrightarrow, \twoheadrightarrow, T, E, O \rangle$ are defined as follows:*

$$\mathit{Time}(\delta) = \textstyle\sum_{(f,\pi,\beta)\in\delta} \left(\, (1-\pi) \times T(f, M) \; + \; \pi \times T(f, C) \; + \; \beta \times T(f, B) \, \right)$$

$$\begin{aligned} \mathit{Energy}(\delta) = \textstyle\sum_{(f,\pi,\beta)\in\delta} (\, & (1-\pi) \times T(f, M) \times E(M) \; + \; \pi \times T(f, C) \times E(C) \\ & + \beta \times T(f, B) \times E(B) \,) \end{aligned}$$

The function $Time(\delta)$ calculates the total time required by the schedule δ, i.e. the time for executing a path of SG according to the scheduler that generates δ. For each fragment f in the system, we add the time $T(f, M)$ if the fragment is executed locally ($\pi = 0$), or the time $T(f, C)$ if the fragment is executed remotely ($\pi = 1$). The function considers also the synchronization time $T(f, B)$ if two consecutive fragments are executed at different locations ($\beta = 1$).

The function $Energy(\delta)$ calculates the total energy required to complete the scheduled path. The difference with respect to the previous function is that here the time of permanence of a fragment in a resource is multiplied by the corresponding energy required per time unit.

Relying on the cost functions introduced above, we can have the time-optimal scheduler Θ_T and the energy-optimal scheduler Θ_E for a given SG, which determine the sequence of actions that generates the less costly combination of resources, in terms of $Time(\delta)$ and $Energy(\delta)$ respectively, for a path in SG.

Sch.	*Time*	*Energy*
1	$T_1 = (3+10+16+10) = 39$	$E_1 = (3+10+16+10) \times 5 = 195$
2	$T_2 = (3+25+3+2+3+5) = 41$	$E_2 = (3 \times 5 + 25 \times 2 + 3 \times 1 + 2 \times 1 + 3 \times 1 + 5 \times 2) = 83$
3	$T_3 = (3+10+5+2+3+5) = 28$	$E_3 = (3 \times 5 + 10 \times 5 + 5 \times 2 + 2 \times 1 + 3 \times 1 + 5 \times 2) = 90$

Table 3: Time and energy costs of the schedules for the simple application

Example 2 (Time and battery costs for the small application) *We evaluate here the schedules proposed in Example 1 (Figure 8) using the cost functions introduced above. Notice that in the calculation we consider only the cyclic part of the application omitting the initialization that is not relevant in terms of an infinite scheduler. Table 3 reports the time and energy consumed for the three schedules calculated according to Definition 8. Evaluating the results, the time-optimal scheduling for the application is achieved in Schedule 3, that is* $(f_0, 0, 0) \rightarrowtail (f_2, 0, 1) \rightarrowtail (f_1, 1, 0) \rightarrowtail (f_2, 1, 1)$*, with a total time cost* $T_3 = 28$*. The offloading choices for achieving such result are formalized in terms of the scheduler (written here using a notation based on triples)* Θ_T*=*$\{(f_0 \longrightarrow f_2,0,0),(f_2 \dashrightarrow f_1,0,1),(f_1 \longrightarrow f_2,1,1),(f_2 \dashrightarrow f_0,1,0)\}$*. On the other hand, Schedule 2, that is* $(f_0, 0, 1) \rightarrowtail (f_2, 1, 0) \rightarrowtail (f_1, 1, 0) \rightarrowtail (f_2, 1, 1)$*, is the energy optimal one, with a total energy consumption* $E_2 = 83$*. The corresponding scheduler is* Θ_E*=*$\{(f_0 \longrightarrow f_2,0,1),(f_2 \dashrightarrow f_1,1,1),(f_1 \longrightarrow f_2,1,1),(f_2 \dashrightarrow f_0,1,0)\}$*. From this example it is clear that there may not be a correspondence between energy and time consumption, since we have different cost results. Hence, defining a scheduler optimized for more resources in general is not a simple task.*

3.4 MobiCa implementation

In this section we present the implementation of the MobiCa syntax in terms of Xtext grammar. The proposed grammar looks like the following:

```
grammar org.xtext.MobiCa with org.eclipse.xtext.common.Terminals

generate MobiCa "http://www.xtext.org/MobiCa"

Model:
(devices+=Device | applications+=Application | systems = System)* ;

Device:
Mobile | Cloud ;

Cloud:
'Cloud' name=ID '[' applications+=[Application]','
                cpuInstructions=INT ']'';' ;

Mobile:
```

```
16 'Mobile' name=ID '[' applications+=[Application] ',' battery=INT
17         ',' network=INT ',' memory=INT ']' '[' cpuInstructions=INT ','
18         energyMobile=INT ',' energyCloud=INT ',' energyBus=INT ']'';';
19
20 Application:
21 'Application' name=ID '{' fragments+=Fragment+ structure=Structure '}'';';
22
23 Structure:
24 'Structure' name=ID '[' edges+=Edge+']'';' ;
25
26 Edge:
27 start=[Fragment] operator=Operator stop+=[Fragment](',' stop+=[Fragment])*
28 ';'
29
30 enum Operator:
31 NDC = '-->' | PAR = '--|' | SEQ ='--::' ;
32
33
34 Fragment:
35 'Fragment' name=ID '[' instructions=INT ',' memory=INT ',' sync=INT ','
36          isOfflodable=('true'|'false') ']' (init='init')?';' ;
37
38
39 System:
40 'System' name=ID ':=' devices+=[Device] ('|' devices+=[Device])* ';'
41 ;
42
43 Query:
44 'Query' name=ID ':' (
45  'E[]' costType=('Energy'|'Time') ',' cost=INT ',' reward=INT
46          (',' constraint=INT)? ';' |
47  'E<>' fragments=INT ';' |
48  'E<>' frag+=[Fragment] ',' time=INT ',' battery=INT ','memory=INT
49          (location=('remote'|'local')?) ';'
50 )*
51 ;
```

The first line declares the name of the language and of the grammar that corresponds to the fully qualified name of the .xtext file. The declaration of the grammar also states the use of the grammar `Terminals`, which is part of the Xtext library and defines the grammar rules for common things like quoted strings, numbers, and comments; so that in our language we will not have to define such rules.

The first rule (line 5) defines the root element of the Abstract Syntax Tree. In this example we declare that a MCC `Model` is a collection of devices and applications that are joined together in a unique system. The fact that they are collections is implied by the operator +=. The star operator * after the brackets states that the elements can be an arbitrary number >= 0, this means that we can have more applications and devices but only one system (notice the = after `systems`).

The shape of each component in the grammar is expressed with its own rule. Starting from the beginning, the first rule we encounter after `Model` is `Device` (line 8). This rule defines a general devices that belongs to the MCC system. We can have two types of device: mobile device and cloud system.

The cloud system is described by its own rule (line 11) called `Cloud`. In this rule are present four keywords, namely `"Cloud", ",", "[",` and `"]"`. Keywords of a DSL are defined by string literals (which in Xtext can be expressed with either single or double quotes). Therefore, a valid cloud declaration statement starts with the keyword '`Cloud`' followed by an ID; there is no rule defining ID in the grammar because that is one of the rules inherited from the grammar Terminals. Then, the squared brackets '[' ']' are expected and within them `cpuInstructions` and `applications` are specified. Notice, after the `applications+=` list we do not want just a name, but the name of an existing `Application`. This can be expressed in the grammar using square brackets and the type we want to refer to. This mechanism is one of the powerful features of Xtext, that is, cross-references. Xtext will automatically resolve the cross-reference by searching in the program for an element of that type with the given name. If it cannot find this element, it will automatically raise an error. The automatic code completion mechanism will also take into consideration cross-references, thus proposing elements to refer to.

The rule for the `Mobile` (line 15) is similar to the Cloud, but with some extra elements like battery, network, memory, etc. The `Application` rule (line 20) is composed by a collection of `fragments` and a `structure`. The symbol + after `Fragment` force the user to define an Application with at least one fragment. A `Structure` (line 23) is a collection of edges, and each `Edge` (line 26) is composed by three elements, a start element that contains a `Fragment`, an operator and a stop element that can contain one or more comma separated destination fragments. The `Operator` (line 30) is not a real rule but just an enumeration of elements: the non-deterministic operator (NDC), the parallel execution (PAR) and the sequential progress (SEQ).

A Fragment (line 34) is defined by the keyword `Fragment` followed by a name and some integer parameters. The offloadability of the fragment is expressed with the boolean variable `isOffloadable` that can assume only true/false values. The root fragment of the application is indicated with the optional symbol (?) keyword `init`. The rule `System` (line 41), defines the components involved in a particular instantiation of the system. This rule is composed by a name and a collection of devices. The devices can be instantiated one after the other just using the operator "|". The last rule is dedicated to the verification `Query`. The MobiCa implementation provides three preconfigured queries, that will be translated following the revisited UPPAAL syntax described in Table 4 (custom queries can be specified by the user directly in UPPAAL). Where *f* is a fragment name, *var* is a placeholder for the variable "GlobalTime, battery, memory, etc." defined in UPPAAL and $\phi_1[\phi_2]$ is a concatenation of formulae. Since we are interested in reachability property we restrict the language to one path quantifier and two temporal operators. The quantifier is used in a particular state to specify that at least one path starting from that state satisfies a given property. The temporal operators, instead, describe properties of a path through the generated tree. In particular the quantifier E (Exists) with the operator <> (Eventually), followed by an expression, define the common query that describes the possibility to find a path along which a certain property is eventually satisfied; this query is usually used at runtime. With the operator [ ] (Globally) instead the query describes the possibility to find a path along which a certain property is globally satisfied; this query is used for determining the infinite scheduling. The MobiCa language does not cover all possible queries that can be expressed in UPPAAL; in fact the aim of the query language incorporate in the MobiCa tool is to just speed up the verification procedure on the technique that we propose.

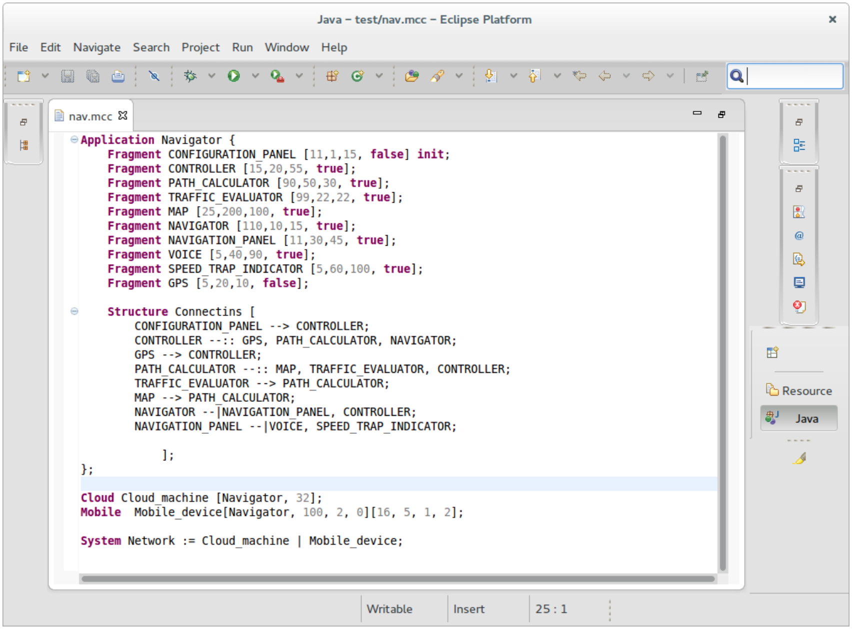
3.4.1 MobiCa editor

Figure 11 shows the MobiCa Eclipse environment generated using the Xtext framework. Using the described grammar, we are able to write

Queries	Q	::=	EF ϕ | EG ϕ
State Fromulas	Ex	::=	f | $true$ | $false$ | var | $\phi_1[\phi_2]$ | Uop ϕ | ϕ_1 Bop ϕ_2
Unary operators	Uop	::=	not
Binary Operators	Bop	::=	$<$ | $<=$ | $==$ | $!=$ | or | and | imply | ....

Table 4: Query syntax

our specifications of MCC systems using the MobiCa DSL. In the Eclipse editor, it is possible to see the MobiCa specification of the navigator case study introduced in Chapter 1.


Java – test/nav.mcc – Eclipse Platform
File Edit Navigate Search Project Run Window Help
nav.mcc
Resource
Java
Writable
Insert
25 : 1

```
Application Navigator {
    Fragment CONFIGURATION_PANEL [11,1,15, false] init;
    Fragment CONTROLLER [15,20,55, true];
    Fragment PATH_CALCULATOR [90,50,30, true];
    Fragment TRAFFIC_EVALUATOR [99,22,22, true];
    Fragment MAP [25,200,100, true];
    Fragment NAVIGATOR [110,10,15, true];
    Fragment NAVIGATION_PANEL [11,30,45, true];
    Fragment VOICE [5,40,90, true];
    Fragment SPEED_TRAP_INDICATOR [5,60,100, true];
    Fragment GPS [5,20,10, false];

    Structure Connectins [
        CONFIGURATION_PANEL --> CONTROLLER;
        CONTROLLER --:: GPS, PATH_CALCULATOR, NAVIGATOR;
        GPS --> CONTROLLER;
        PATH_CALCULATOR --:: MAP, TRAFFIC_EVALUATOR, CONTROLLER;
        TRAFFIC_EVALUATOR --> PATH_CALCULATOR;
        MAP --> PATH_CALCULATOR;
        NAVIGATOR --|NAVIGATION_PANEL, CONTROLLER;
        NAVIGATION_PANEL --|VOICE, SPEED_TRAP_INDICATOR;

            ];
};

Cloud Cloud_machine [Navigator, 32];
Mobile  Mobile_device[Navigator, 100, 2, 0][16, 5, 1, 2];

System Network := Cloud_machine | Mobile_device;
```

Figure 11: MobiCa Editor: Navigator application

For the sake of readability we, also report the code in the Listing 3.1. Looking into detail, a system (line 28) is composed by a network of a `Cloud_machine` (line 25) and a `Mobile_device` (line 26) that collaborate for providing the navigator system. In the proposed 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 values 5, 1, 2 in the second group of parameters are the energy consumed per time unit by the mobile device when is in use, when is in idle and during the synchronization along the bus, respectively. The navigator application (line 1) is defined using the keyword `Application`, and it is composed of a set of `Fragments` (lines 2-11) that are connected according to the transitions defined inside the structure block. As example of fragment the CONFIGURATION_PANEL (line 2), consists of 11 instructions, requires 1 MB of memory at runtime and 15 MB to be synchronized. Moreover, it is not offloadable and it is the root fragment of the application. The `Structure` (lines 13-22) contains the relations between fragments. For example the first transition (line 14) claims that the computation can move from the CONFIGURATION_PANEL to the CONTROLLER in non-deterministic way. Notice that in a system it is possible to have more devices and different applications, but to exploit the benefit of MCC at least we should have one mobile, one cloud and an application shared between them. The definition of the `Query` is the last step. In MobiCa we permit to define a query just introducing the relevant parameters, as the tool will automatically generate the preconfigured query, defined according to the low-level information within the generated TA model of the system. For example, the query generated from the `query3` (line 32) specification will be

```
E[]VOICE.id36 and globalTime<300 and battery>50 and
                    memory<100.
```

Practically, from the Navigator implementation our tool automatically create an XML file that can be taken in input by UPPAAL. For the sake of readability the XML file is omitted here, we refer the interested reader to the Appendix B.

```
Application Navigator {
Fragment CONFIGURATION_PANEL [11,1,15, false] init;
Fragment CONTROLLER [15,20,55, true];
Fragment PATH_CALCULATOR [90,50,30, true];
Fragment TRAFFIC_EVALUATOR [99,22,22, true];
Fragment MAP [25,200,100, true];
Fragment NAVIGATOR [110,10,15, true];
Fragment NAVIGATION_PANEL [11,30,45, true];
Fragment VOICE [5,40,90, true];
Fragment SPEED_TRAP_INDICATOR [5,60,100, true];
Fragment GPS [5,20,10, false];

Structure Connectins [
CONFIGURATION_PANEL --> CONTROLLER;
CONTROLLER --:: GPS, PATH_CALCULATOR, NAVIGATOR;
GPS --> CONTROLLER;
PATH_CALCULATOR --:: MAP, TRAFFIC_EVALUATOR, CONTROLLER;
TRAFFIC_EVALUATOR --> PATH_CALCULATOR;
MAP --> PATH_CALCULATOR;
NAVIGATOR --|NAVIGATION_PANEL, CONTROLLER;
NAVIGATION_PANEL --|VOICE, SPEED_TRAP_INDICATOR;
];
};

Cloud Cloud_machine [Navigator, 32];
Mobile  Mobile_device[Navigator, 100, 2, 0][16, 5, 1, 2];

System Network := Cloud_machine | Mobile_device;

Query query1: E[], Time, 300,41;
Query query2: E<>, 20;
Query query3: E<> VOICE, 300, 50, 100;
```

Listing 3.1: Navigator MobiCa code

Chapter 4

Scheduling Analysis

In this chapter we introduce the main techniques we use to find the optimal scheduling in a MCC application. The main distinction in the scheduling analysis is on the time the analysis is carried out: the first one is at design-time and the other one at run-time. These approaches are similar, but with different characteristics and effort.
The design time approach is the more straightforward method to implement, it only requires the application model and we do not have constraints on the response time. At design time is possible to perform an accurate analysis, but it is not simple to consider all possible scenarios that can happen during the application runtime.
Therefore, we propose also a runtime technique that is able to adapt to the evolution of the application, but this requires a dedicated infrastructure, like a framework equipped with a monitor. This has to follow the application current state and, at the same time, to calculate the strategy to follow for the next states. The main drawback in this technique is related to the response time of the verifier that should be almost in real time. For improving the performance, we adopted some shrewdness that simplify the problem to a smaller one and permit to have a faster answer. Another issue to face is related to the necessity of an ad-hoc infrastructure for runtime analysis and this implies that also the application should be developed with the possibility to extend its behaviour

with a decision support.

In Section 4.1 we present the design time analysis for an optimal infinite scheduling based on the cost/reward horizon method. In Section 4.2 we present the runtime decision support based on the reachability technique. The cost optimal reachability is the problem of finding the minimum cost (e.g. time, energy, fragments) for reaching a given goal location. Finally, in Section 4.3 we present a technique based on Stratego(DJL$^+$15), an UPPAAL version that is a dedicated tool for synthesize strategy on timed automata model.

4.1 Design time approach

In this section, we take into consideration schedulers that produce infinite schedules ensuring the satisfaction of a property for infinite runs of the application. This is particularly useful for MCC, where applications are supposed to provide permanent services, or at least to be available for a long period. In particular, considering that our model is equipped with constraints on duration, costs and rewards, we are interested in identifying the optimal schedulers that permit the achievement of the best result in terms of energy consumption and execution time. In fact, over infinite behaviours, it is possible to recognize a cyclic execution of components that is *optimal* and is determined by means of the limit ratio between accumulated costs and rewards. Consequently, an optimal scheduler is given by maximizing or minimizing the cost/reward ratio.

4.1.1 Cost/reward horizon method

In order to find the optimal scheduler for an application with infinite behavior, as discussed above, we propose a cost/reward horizon method. From the literature (BBL08; RLS06), we know that the optimal ratio is computable for diverging and non-negative reward systems.
In what follows we first present the basic concepts behind our cost/reward method and then we show how the optimal infinite scheduling problem can be solved using timed automata models and the UPPAAL

model checker.
The behavior of the application is the starting point for defining an optimal infinite scheduler. It can be described as a set of paths of the considered system graph SG. For each path UPPAAL will generate all possible schedules and will choose the best one according to a specific ratio (clarified below). The chosen schedule is indeed the less costly one and, hence, it can be used to synthesize the rules that compose the optimal scheduler.

Let's start by defining the ratio of a finite path η of a SG as follows:

$$Ratio(\eta) = Cost(\eta)/Rewards(\eta)$$

where $Cost()$ and $Rewards()$ are two functions keeping track of the accumulated cost/reward along the path η. Now, we extend this ratio to an infinite path γ=$f_0 \rightarrowtail \ldots \rightarrowtail f_n \rightarrowtail \ldots$, with γ_n=$f_0 \rightarrowtail \ldots \rightarrowtail f_n$, the finite prefix of length n; the ratio of γ is:

$$Ratio(\gamma) = \lim_{n\to\infty}(Cost(\gamma_n)/Rewards(\gamma_n))$$

The optimal infinite path is the one with the smallest $Ratio(\gamma)$ among all possible schedules.

Finding the smallest ratio is not always a tractable problem, but it is possible to improve its tractability reducing the problem to a given horizon. From this new point of view, we want to maximize the reward in a fixed cost window. Notice that the cost window should be of an appropriate length, in order to complete the execution of at least one application cycle.
This technique can be implemented in UPPAAL considering the query:

E[] not(f_1.Err, ..., f_n.Err) and (Cost≥C imply Reward≥R)
(4.1)

This query asks if there exists a trace where the system keeps running without errors (identified by $f_i.Err$) and whenever the predefined cost C is reached, the accumulated reward should be at least R (see Figure 12).

For verifying the satisfaction of the above formula, the TA model includes an additional template (Figure 13) implementing the cost window using the reset mechanisms. It consists of one state and a self-loop transition, where each time the simulation reaches the cost C the transition

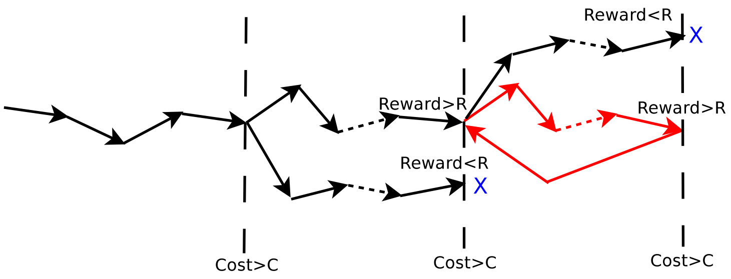


Figure 12: Cost/reward horizon method

will reset the accumulated `Costs` and `Rewards`.
In this way, the behavior of the application is split in cost windows and in each of them the accumulated rewards should satisfy the formula $\texttt{Reward} \geq \texttt{R}$.

Since we are looking for the maximum reward given a predefined cost, for finding the optimal scheduler it is necessary to discover the maximum value of `R` for which the formula (4.1) holds. The resulting trace generated by a satisfiable formula has the structure depicted in Figure 12. The trace starts with some initial actions corresponding to the application start-up and leads to its cyclic behavior. As shown in the figure, the approach does not consider all possible traces, but only the ones that satisfy the constraints of the query. The candidate schedule is the piece of trace that is highlighted in red in the figure, which means that UPPAAL has found a cyclic behavior in the application whose execution satisfies the formula forever. This means that we have found an optimal schedule from which it is possible to derive the set of rules that will gen-

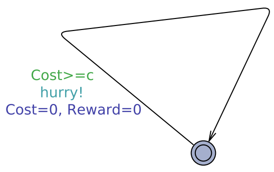


Figure 13: Reset TA

erate the optimal scheduler. Until now the derivation is done by hand, but can be simply automatized just constructing a parser able to read the textual results generated by UPPAAL.

4.1.2 The horizon method at work

In this section we show how the cost/reward horizon method can be applied to MCC systems and, in particular, to the Example 1 presented in Figure 7.

We are interested in finding a time-optimal and/or battery-optimal scheduler. By applying the method presented in Section 4.1.1 to a MCC system, given an infinite path γ, the time- and energy-based ratios become:

$$r_T = \lim_{n\to\infty}(Time(\gamma_n)/Fragments(\gamma_n));$$
$$r_E = \lim_{n\to\infty}(Energy(\gamma_n)/Fragments(\gamma_n)),$$

respectively.

Thus, the accumulated costs are calculated by the functions $Time()$ and $Energy()$ given in Definition 8. The rewards are instead defined by a function $Fragments() : \eta \to \mathbb{N}$ which counts the number of fragments executable in the fixed window. The more fragments we are able to execute with the same amount of time or energy, the better the resources are used.

To find the minimum time-based ratio using the UPPAAL model checker we can ask the system to verify a query of the following form:

```
 E[] forall(i:pid_t) not(Fragment(i).Err)
and (GlobalTime≥300 imply (fragments>41))
```

In this specific case we want to know if, in a fixed window of `300` units of time, it is always possible to execute `41` fragments. To find the minimum ratio we have to iterate on the number of fragments in the denominator in order to find the maximum number for which the query holds. In our running example, the maximum value that satisfies the query is 41, giving a ratio 300/41 = 7.31. The resulting trace generated by the presented query results in an execution sequence that can be synthesized as the Schedule 3 shown in Figure 8.

The query for determining the energy-based ratio is defined as:

```
        E[] forall(i:pid_t) not(Fragment(i).Err)
  and (battery≥900 imply (fragments>43 and battery≤930))
```

In this case, the resulting ratio is $900/43 = 20.93$. Thus, the system requires 20.93 units of battery per fragment. Notice that in this query there is an extra constraint defined as an upper bound on the right side of the `imply` keyword. This is because we can have different schedules satisfying the formula, but we consider only the ones that exceed the battery threshold as little as possible. The resulting trace from the energy query gives us the energy optimal schedule, that in this case can be synthesized as the Schedule 2 in Figure 8.

To assess the truthfulness of the cost/horizon method, we can compare the obtained results with the ones calculated directly in the Gantt chart in Figure 8. The energy ratio for Schedule 2 on one loop is given by $83/4 = 20.75$. The slight difference in the results is due to the size of the cost window. In the Gantt chart we are considering a window that fits perfectly four fragments and, hence, we do not have any incomplete fragment that affects the final result as in the case of the horizon method, but the approximation is really close to the best case.

4.1.3 Evaluation of custom schedulers via SMC

In this section, we present a technique for evaluating the performance of a custom scheduler using the Statistical Model Checking facilities (BDL$^+$12; DLL$^+$11) provided by UPPAAL (the model checker is called UPPAAL-SMC, or SMC for short).

Let us suppose now that a developer wants to define his own scheduler for an application and to know how much the resulting custom scheduler is close to the optimal one or to calculate some statistics. Possible reasons for customizing a scheduler could be problems in the development phase related to hardware cost or less quantitative issue, such as security and privacy that force the developer to introduce a static scheduler.

The new personalized scheduler in UPPAAL is modeled as a TA template called Manager. The duty of this manager is to decide on which resource each fragment should be executed. Considering the model presented in Figure 9, here we do not have anymore the decision in the *ready* location between mobile and cloud transition, but just a transition that is synchronized with the manager. The manager operates as a mediator, between the fragments and the resources. Once the manager receives notice of a new fragment to execute, it decides according to some rules in which resource's waiting list to move it. The resources are modeled following a busy waiting paradigm, where every time the queue is not empty, a new fragment is consumed. Before executing the assigned fragment, the resource checks the execution location of its predecessor; if data synchronization is not required, it just executes the fragment, otherwise it synchronizes the data and then processes it. Once the computation is completed, the resource returns the control back to the executed fragment and passes to the next one.

The manager can contain different kinds of rules. One possibility is to define a rule for each fragment or to specify a more general rule that can be applied to the whole application. For example, to execute a fragment remotely when it is offloadable is a very simple rule that a developer can consider to implement in a MCC system. Figure 14 depicts the TA model

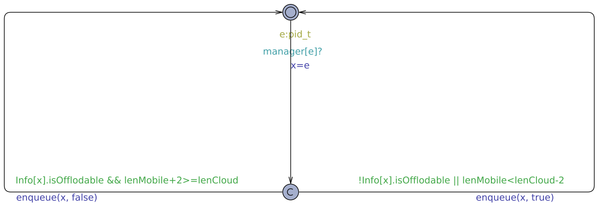

e:pid_t
manager[e]?
x=e
C
Info[x].isOfflodable && lenMobile+2>=lenCloud
enqueue(x, false)
!Info[x].isOfflodable || lenMobile<lenCloud-2
enqueue(x, true)

Figure 14: Manager TA

of the manager implementing this custom strategy. In detail, after a fragment is synchronized with the manager, the latter decides to enqueue the fragment on the corresponding waiting list according to the guard `Info[x].isOffloadable`. In this way, if the fragment `x` is offloadable it is queued locally, otherwise remotely.

Looking more closely at the custom scheduler implemented in the manager, we notice it realises the same behavior of Schedule 2 in Figure 8, which we already know to be energy-optimal.

As previously said, usually a customized scheduler is defined for reasons related to particular conditions of the environment. Now, we perform some statistical analysis to quantify how much the customized scheduler above is far from the optimal one, in order to have a quantitative measure of any performance loss. In particular, below we present some verification activities and compare the obtained results with the time/energy-optimal scheduling we found in Section 4.1.2.

By evaluating the following queries using the SMC tool, we can determine the expected maximum value for the number of fragments that can be executed in a given temporal window:

```
E[time<=300;2000](max:fragments)
E[battery<=900;2000](max:fragments)
```

These two queries aim at finding the expected value over 2000 runs in a window of 300 units of time and 900 units of battery, respectively. The results are: 29 fragments for the first query and 41 for the other one. Comparing these results with those of optimal ones, we can clearly see that the scheduler defined by the developer is almost as efficient as the energy-optimal one. Indeed, they differ only for 2 fragments in the energy case. Instead, the performance of the custom scheduler is very far from that of the time-optimal scheduler, as they differ for 14 fragments.

The proposed strategy can be also evaluated to see if it is closer either to the energy-optimal scheduler or to the time-optimal one. This can be achieved by checking if the probability to reach the time-optimal scheduler is greater than the probability to reach the energy-optimal scheduler.

```
Pr[time<=300](<> fragments>=43)>=
Pr[battery<=900](<> fragments>=41)
```

The result of this query is `false` with probability `0.9`, meaning that the probability of reaching the energy-optimal scheduler is greater than the one for the time-optimal scheduler.

We can also simulate the system behavior by executing the following commands:

```
simulate 1[time<=300]{battery,fragments}
simulate 1[battery<=900]{time,fragments}
```

Their results are shown in Figure 15. On the top graph, we have the number of fragments compared with the consumed battery using a time scale. On the bottom graph, instead, we have the ratio of executed fragments and required time using a battery scale. The graphs simply show the ratio between costs and rewards according to the elapsing of time or battery usage. It can be simply defined drawing a vertical line and divide the obtained time/battery value for the number of fragments executed. Furthermore the figures summarize the consumption of resources according to the progress of the application.

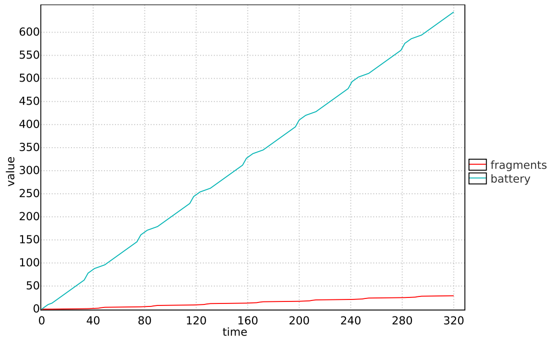
4.1.4 Application to the navigator case study

We present now the results obtained using the cost/reward horizon method applied to the navigator case study. The complexity of this example, shown in Figure 16, makes it a good test bed for our method. Notice that the values of the parameters used in the example are generated ad-hoc as a proof of concept. From real life we expect that the developer can determine information about fragment instructions by performing experimentation, relying on statistics or simply studying the complexity in the code. The diagram in Figure 17 shows the resulting schedules synthesized from the verification of the following queries:

```
     E[] forall(i:pid_t) not(Fragment(i).Err)
  and (time≥100 imply (fragments>16 and time<120))

     E[] forall(i:pid_t) not(Fragment(i).Err)
and (battery≥400 imply (fragments>21 and battery<419))
```

The first query defines the time-optimal schedule with respect to a time window of 100 units and with a maximum number of executed fragments equal to 16. The ratio $r_T = 100/16 = 6.25$ was reached keeping the execution local for almost all fragments except for f_8 and f_9, which are executed in parallel remotely.



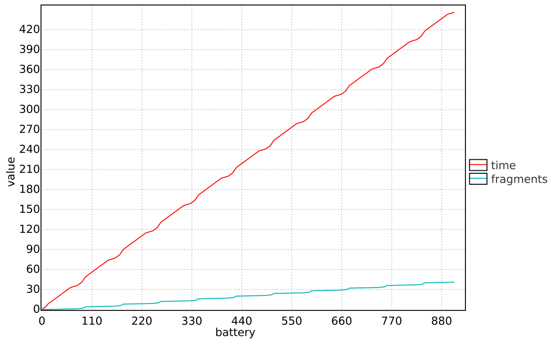


Figure 15: Simulation results

The opposite behavior is identified in the verification of the second query for the energy-optimal case, where only three fragments are executed locally and all the others remotely. Since the fragments f_7 and f_2 are not offloadable, they are maintained locally together with the fragment f_1. The choice to execute f_1 locally is given by the necessity of the scheduler to wait for a suitable moment to move the computation remotely. Clearly, moving the computation between two non-offloadable fragments is not convenient; furthermore, sometimes it is better to anticipate or postpone the offloading when the data synchronization is minimal or less costly. The ratio of this scheduler is $r_E = 400/21 = 19.05$ with a final energy consumption equal to 272 units per cycle.

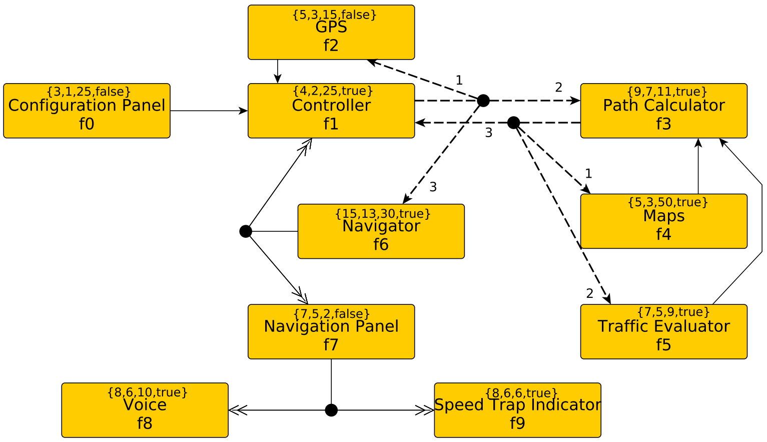


Figure 16: Navigator case study: graphical representation of the MobiCa specification

The cost/reward horizon method fits MCC systems perfectly. In particular, during the sequential behavior of the application it tries to find the best moment for moving the computation remotely, defining also different strategies according to the role of the fragment. Instead, during parallel behavior, where there are no direct relations between fragments, it tries to exploit the benefit derived by allocating the computation both on the mobile and on the cloud.

As a final evaluation we present the results obtained verifying the custom scheduler described in Section 4.1.3 on the navigator case study using SMC. Verifying the expected maximum reward using the query

```
E[battery<=400;2000](max:  fragments)
```

we obtain a cost energy ratio $r_E = 400/16 = 25$. Even worse is the score obtained by trying to optimize the performance using the query

```
E[time<=100; 2000](max:  fragments)
```

which achieves a ratio $r_T = 100/5 = 20$. Thus, comparing the obtained values, it is possible to notice a substantial growth of the ratio for the

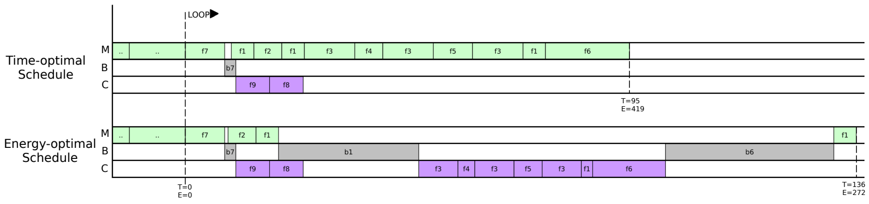


Figure 17: Navigator case study: optimal schedules

custom scheduler. Since a higher ratio means a decrease in performance, we can claim that the strategy defined by the developer is not a good approximation of the optimal one. Furthermore, analyzing the results in more detail, we notice that the custom scheduler is very far from the time optimal, with a ratio that is four time larger than the one achieved by the optimal scheduler. Considering instead the energy case, it is possible to reach a ratio of 25 against the 19.05 of the optimal one. Looking at these results, the developer is aware that using his custom scheduler, he can achieve a good performance if he is interested in energy optimization, although this is not optimal. By performing a simulation, we can see a significant gap between the number of executed fragments and the elapsing of time according to the consumed battery power; there is indeed a symmetric increase of values generated by the cyclic behavior of the application.

The plot in Figure 18, instead, represents a scheduler synthesized using a histogram. Using an appropriate simulation query, which takes into account the fragments in execution on the resources, it is possible to represent each fragment as a column of the same height of its identifier in the specific resource. For the sake of readability, columns referring to cloud (red lines) and mobile (green lines) are depicted on the same level of the graph, while the network columns (blue lines) are reported just below. A peak in the blue line means that the corresponding fragment above requires the synchronization on the bus before its execution.

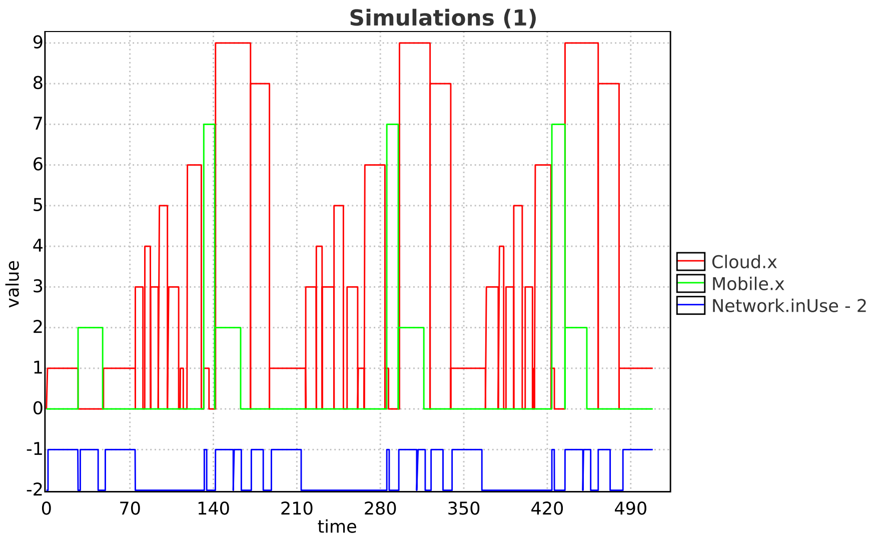


Figure 18: Simulation results

4.2 Run-Time analysis

The analysis at design time presented before can be improved using the runtime approach. This methodology has the ability of suggesting the offloading strategy for future fragments of the system starting from the current state. By means of an ad-hoc framework, this approach has the capacity to identify the best execution strategy of *k* steps ahead. Here, we propose the runtime optimization based on the cost minimization for reachability property. The idea is to use the diagnostic trace, which is automatically generated by the UPPAAL tool during the verification, as a schedule for taking decisions during the application execution. The schedule is a trace enriched with the execution location for each fragment and a flag that indicates the necessity of synchronizing data in case of different execution location for the consecutive fragment.

x	seconds
10	0,17
11	0,29
12	0,49
13	0,71
14	1,38
15	3,19
16	7,88

Table 5: Verification response time

4.2.1 Minimization of reachability properties

UPPAAL can provide three kinds of traces according to the diagnostic trace set-up: some trace, the shortest one, or the fastest one. The queries we present here, are not related only to a specific application, but they have parameters, such as fragments, battery, memory and elapsing of time, that are relevant to MCC at large and not only for this specific example.

The first approach is using the window optimization technique. These analysis can be achieved using the query of the form:

```
E<>fragment>=x
```

Which means ″it is possible to find a path with a number of fragments greater than x″. In particular, fixed a number `x` of fragments to explore, starting from the current state and verifying for the fastest trace, the result can be identified as the time-optimal schedule for the proposed application.

Table 5 contains the execution time expressed in seconds used by UPPAAL to provide the resulting schedule for the navigator example according to a given number of fragments `x` to predict.

There is an exponential growth of seconds according to the increasing of the value `x`. This criticality produces issues related to the right setting of the parameter `x`. Since the complexity is exponential, a good compromise could be to set up a small `x` that guarantees a fast result and then, using an iterative deepening approach, to increase the value step by step until the application execution reaches the depth of x.

The presented query is specific for time optimization. On the same way it is possible to verify other parameters, like battery or memory, implementing this new measure as clocks values.

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

Table 6: UPPAAL verification results

In Table 6, we present some others relevant queries for the navigator case study of the form:

```
E<> FragmentID and Time<x and Energy>y and Memory<z
```

The query presented belongs to the reachability property set, but the difference with respect to the previous one is on the frontier, in this case we want to reach a precise point in the application using a certain amount of time and memory while guaranteeing a certain amount of residual battery. The first query in the table 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". By 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 19 where, for the sake of presentation, 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*, displaying step by step if the fragment is executed locally or remotely. The final resource usage for the mentioned query is shown in Table 6. The time consumed is equal to 299, the residual battery is equal to 87 and the used memory is equal to 21. These results are obtained executing the fragments *CONTROLLER, PATH_CALCULATOR, TRAFFIC_EVALUATOR, MAP, NAVIGATOR*, and *NAVIGATION_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 6, we can then check a query using the results of the previous queries as constraints. In this case, using again 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 6. In addition to the previous constraints, there we want to force the system

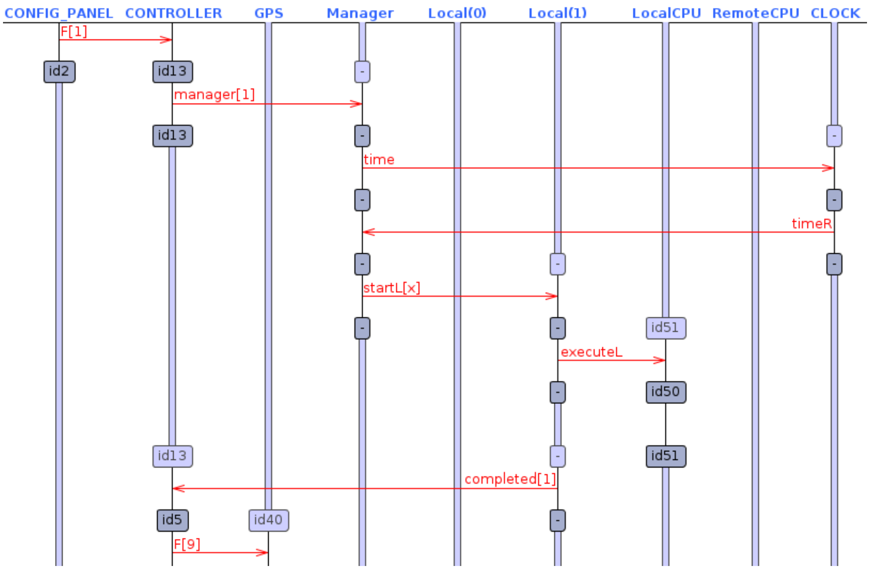


Figure 19: UPPAAL verification

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.

4.2.2 Runtime framework

The runtime approach can be supported by a software framework able to include all the features and characteristics useful for the correct usage of the decision support. The framework will be a middleware shared between the mobile device and the remote cloud in order to provide a common base for the implemented services.

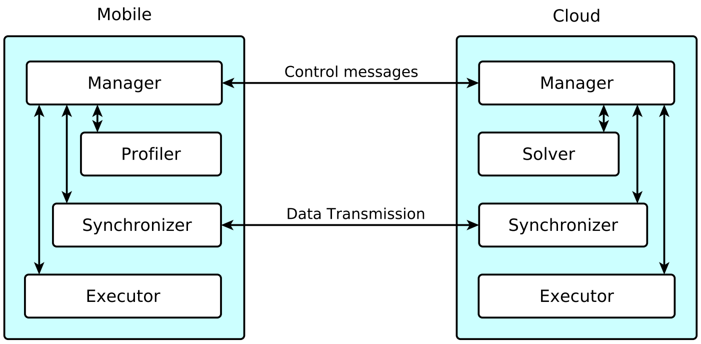


Figure 20: Runtime framework

Figure 20 shows the framework structure. It consists in a series of modules that permit to create the right environment for the system. The `Manager` is the main component of the framework and it is divided in local component and remote one. It is able to coordinate and take decisions regarding the execution location of the fragments to transfer to the other modules. The `profiler` maintains information regarding the execution time of the fragments, the progress of the application execution, the network bandwidth, the remaining battery, the free memory and the CPU loads. All this information collected by the profiler will be exchanged like control messages between local and remote manager, and forwarded to the remote solver by necessity. The `solver` is the heart of the framework where the decision support is located. In the solver resides the application model and the UPPAAL engine that has the duty of calculating the best schedule for the application. The schedule generated by the solver is maintained both on the local and remote manager and is used to coordinate the synchronization and execution of the fragments in the respective modules. The `executor` has the duty of executing the fragments of the application and the `synchronizer` module is used to maintain consistent the state of the system between two consecutive fragments executed in different locations. Another role of the manager is to monitor the variation of the resources' parameters collected by the solver in order to avoid wrong schedule executions. A typical variation in the resources conditions can be generated by the fluctuation of the network bandwidth due to connection instability. During this situation the proposed schedule might be not anymore optimal and so the manager should require a new estimation to the solver. Another situation to manage is the disconnection of the device from the network. So, for guaranteeing reliability, the manager should bring back the application to the last consistent state (checkpoints) and restart the computation locally (rollback). This permits to guarantee continuity in the usage of the application without loss of information. The concepts of roll-back, consistent state and checkpoints are considered out of the scope for this thesis, we refer the interested reader to the following standard works on these topics (KT87; EAWJ02; Ran75).

4.2.3 Invocation interval

The decision support based on model checking suffers from the common problem, as highlighted from Table 5 of state space explosion. For this reason in order to exploit the computation power and the large quantity of available resources, the UPPAAL engine is deployed in the cloud part. Of course this expedient is not enough for providing fast results at runtime, especially for optimization problems being part of the NP-Hard problems. It is worth noticing that the main purpose of this work is not focused on the performance, as we want to show a new methodology where the model checking is able to solve a different problem with respect to what we are used to think. For this reason the verification is based on an iterative deepening method, where the query are bounded in the number of fragments, permitting a faster answer and more flexibility to the problem adaptation. This technique, exploring gradually the state space, allows fast answers using a small exploration depth, bypassing the main issue imputable to the model checking and permitting its application to run-time contexts. At the same time we want to have a prediction optimized for long runs.

Algorithm 1 Decision support algorithm

```
resources=getResources();
depth=10;
k=depth;
schedule==null;
while (!applicationStop) do
  if (schedule==null) then
    schedule=solver(getCurrentState(), resources,
            depth);
  else if (resources==getResources()) then
    schedule=solver(getCurrentState(), resources,
            k++);
  else if (resources!=getResources()) then
    newInst=solver(getCurrentState(),
            getResources(),depth);
    if (newInst!=schedule) then
      schedule=newInst;
      k=depth
    end if
  else if connectionLost then
    schedule=null;
  end if
  if (currentFragmet==k-x) then
    k=depth
  end if
  return schedule;
end while
```

The pseudo-code Algorithm 1 describes the behaviour of the framework according to different conditions that can show up. At the application initialization, the remote `manager` requires the parameters regarding the resources, using the method `getResoruces()` and setting the global variables needed for the system.

The first step, when the scheduler is still null, the solver calculates the first schedule with a prefixed number of fragments given by the variable `depth`. The prefixed depth guarantees a fast solution, given by a small look ahead. After that, the manager shares the results with the client and starts the execution coordination.

After this first initialization, the prediction can continue gradually increasing the depth of exploration in order to obtain a longer schedule. The new discovered schedule will replace the old one without interruption for the application. Notice that `getCurrentState()` is used for keeping the starting point of the prediction consistent with the current state of the application execution.

Notably, the above procedures are valid only in conditions where the resources are stable over time, for example when the system is connected to the wireless network. But this is not always the case, because unstable network can lead to continuous variation of the surrounding environment parameters. At this point the manager will ask the solver to run a new instance of UPPAAL with the most recent parameters and, in case of difference in the results, the old instance will be interrupted and the new schedule will be synchronized. Of course each time that a difference in the parameters is revealed, a new instance in the solver is opened and only the best one will be kept.

The last scenario is the complete disconnection of the mobile device with the remote part. In this case all the fragments are redirected locally, restarting the computation from a consistent state. This until the connection is restored and a new schedule is provided.

In case the application is faster than the solver to predict an extra step (`if(currentFragment==k-x)`), the manager requires again the information to the profiler and restarts the solver with the updated information. The `x` is necessary to restart the computation of the scheduling in time, before that the application reach the prefixed depth and remain without the prediction. In this circumstance the solver will recalculate the optimal schedule decreasing the depth to the constant `depth`. This mechanism guarantees always the longest optimized schedule, until the instant in which the computation of a new schedule is longer than the execution of all predicted fragments. The proposed algorithm is devised on purpose for running on the cloud infrastructure in order to save mobile resources and avoiding the system overhead. Define a precise execution cost is really a complex task, because the algorithm is in loop until the application is stopped and for each iteration an instance of the solver is executed. The solver module is highly influenced by the prediction depth and the dimension of the current explored graph and its complexity is synthesised in Table 5 at page 63.

4.3 Generation offloading strategies via UPPAAL Stratego

In this section we give an intuition of how it is possible to define a strategy using the UPPAAL Stratego. The tool integrates other two branches of UPPAAL: SMC (BDL$^+$12) and Tiga (BCD$^+$07) specifically devised for synthesis for time games.

Stratego is a novel tool which facilitates the generation, optimization, comparison and performance exploration of strategies for stochastic priced timed games in a user-friendly manner. The tool allows an efficient and flexible strategy-space exploration before the adaptation in a final implementation.

UPPAAL Stratego comes with an extended query language where strategies may be constructed, compared, optimized and used in statistical model checking for game under the constraints of a given synthesized strategy. The tool can operate on three different kinds of strategies, all of them memoryless. *Non-deterministic* strategies give a set of actions in each state, with the most permissive strategy ”when it exists”, offering the largest set of choices. In the case of timed games, most permissive strategies exist for safety and time-bounded reachability properties. *De-*

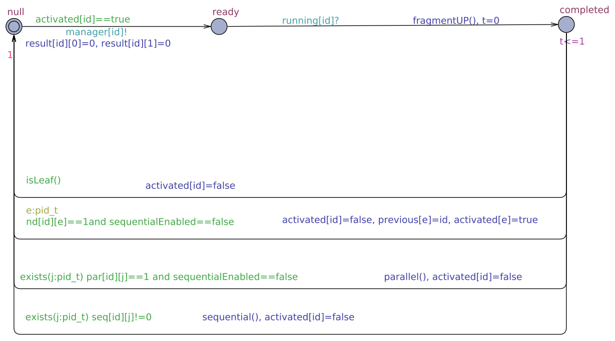

null
activated[id]==true
manager[id]!
result[id][0]=0, result[id][1]=0
1
ready
running[id]?
fragmentUP(), t=0
completed
t<=1
isLeaf()
activated[id]=false
e:pid_t
nd[id][e]==1and sequentialEnabled==false
activated[id]=false, previous[e]=id, activated[e]=true
exists(j:pid_t) par[id][j]==1 and sequentialEnabled==false
parallel(), activated[id]=false
exists(j:pid_t) seq[id][j]!=0
sequential(), activated[id]=false

Figure 21: Fragment example

terministic strategies give one action in each state. *Stochastic* strategies give a distribution over a set of actions in each state.

4.3.1 Navigator case study on Stratego

We describe here how it is possible to generate a schedule for a variation of the navigator case study using the Stratego tool. Thanks to a friendly user interface, Stratego permits to construct the problem in a graphical way, simply drawing the model in the dedicated editor. Doing a comparison with other versions of UPPAAL, in Stratego we have the possibility to define two types of transitions. The solid arrows are transitions managed by the controller and the dashed arrows are managed by the stochastic environment. The translation from MobiCa to the timed automata model for Stratego is hence similar to the statistical case.

Figure 21 depicts the template for fragments implemented in Stratego. In the *null* location the TA has only one output transition, which has the duty to synchronize the computation with the manager in Figure 22 for providing the offloading decision. The manager duty is to divide fragments between local and remote execution; after the reception of a new task, the manager does not force anymore the execution, but restrict only the local execution for non offloadable fragments. This behaviour is managed by the dashed arrows for the execution decision that are now

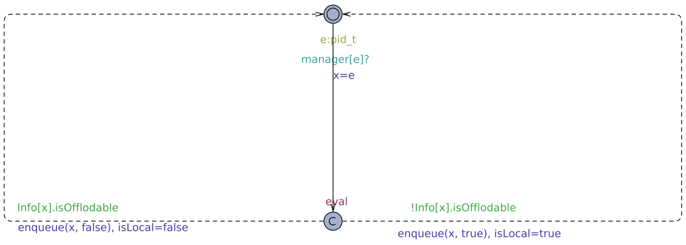


Figure 22: Manager example

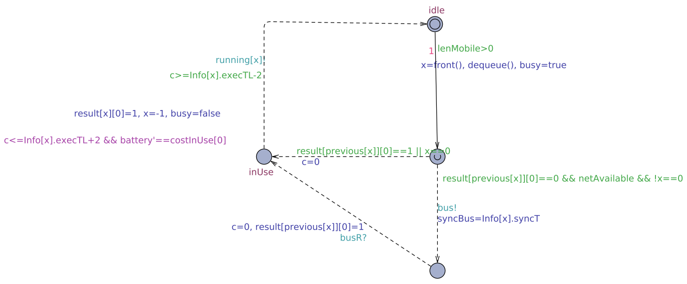
uncontrollable. The same variation is applied to the resources' models (Figure 23), where all transitions except the first one on the reception of a task are uncontrollable.
Once defined the model, the last step is to use the appropriate verification query for defining a strategy. The query syntax for the Stratego version is a bit more complex, because it includes all the model checking functions plus the strategy derivation. For a more complete view, the reader can refer to the Appendix A.6.

Here below we try to define a strategy on the navigator case study using Stratego. As first step it is necessary to define a safety property. The scope of this property is to guarantee to respect of a determined condition for the strategy. Using the query in Listings 4.1, we state that a safe strategy should always reach a determined point in the application, that in our case is the complete execution reaching the fragments 8 and 9. The completion of such fragments is underlined by a true flag on the *result* array.

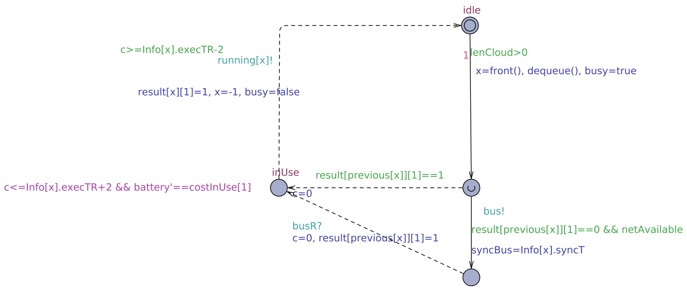
```
strategy safe=control: A<> ((result[9][0]!=0 || result
    [9][1]!=0) && (result[8][0]!=0 || result[8][1]!=0))
```

Listing 4.1: Safe query

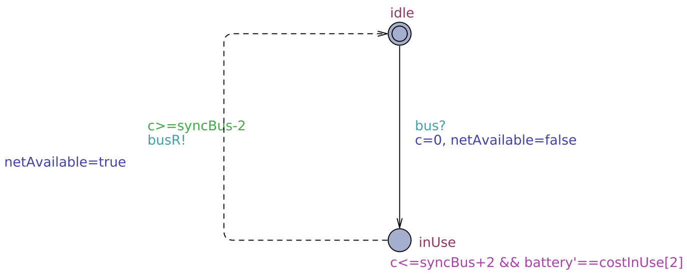
Once defined the precondition safe, it is possible to define the *fast/fastE* strategies (Listing 4.2). Using the *fast/fatsE* properties we require the system to find the strategy that minimizes the time or battery for reaching a point in the application respecting the *safe* condition. If the system is able


idle
1
lenMobile>0
x=front(), dequeue(), busy=true
running[x]!
c>=Info[x].execTL-2
result[x][0]=1, x=-1, busy=false
c<=Info[x].execTL+2 && battery'==costInUse[0]
result[previous[x]][0]==1 || x==0
c=0
inUse
result[previous[x]][0]==0 && netAvailable && !x==0
bus!
syncBus=Info[x].syncT
c=0, result[previous[x]][0]=1
busR?

(a) Mobile


idle
c>=Info[x].execTR-2
running[x]!
1
lenCloud>0
x=front(), dequeue(), busy=true
result[x][1]=1, x=-1, busy=false
inUse
result[previous[x]][1]==1
c<=Info[x].execTR+2 && battery'==costInUse[1]
c=0
bus!
busR?
c=0, result[previous[x]][1]=1
result[previous[x]][1]==0 && netAvailable
syncBus=Info[x].syncT

(b) Cloud



(c) Network

Figure 23: The resources' models

to find a strategy, after the verification procedure will return a message containing: the number of runs, the number of iterations for each run, the number of reset and the total number of iterations.

```
strategy fast=minE(time)[<=100]:<>((result[9][0]!=0 ||
    result[9][1]!=0) && (result[8][0]!=0 || result
    [8][1]!=0)) under safe

strategy fastE=minE(battery)[<=50]:<>((result[9][0]!=0 ||
    result[9][1]!=0) && (result[8][0]!=0 || result
    [8][1]!=0)) under safe
```

Listing 4.2: Fast/FastE query

The found strategy is maintained in memory and accessible for further analysis.
Listing 4.3 shows two queries that try to elaborate the obtained strategy for discovering additional relevant information and drawing the suggested behavior.
The first query provides the expected maximum time for the defined strategy to reach the prefixed point after 200 simulations.
The second query instead is a simulation of the obtained strategy in the application model. The simulation can be personalized according to the user needs and, in this case, we focus on the consumed battery, the execution location for each fragment identified by the ID, and the availability of the network. For graphical reason, we inserted in the query mathematical operations like division and subtraction for avoiding overlap in the results. Using the simulation query for a time period of 60 units of time on the fastE strategy, we obtain the chart in Figure 24. The sky blue line sketches the executed fragments, where the height of the columns corresponds to the ID of the fragments and the width corresponds to the permanence time on a resource. The blue line denote the usage of the network (the synchronization along the bus), and the green one the location execution for the fragment. For example, if we consider a vertical line at time 5, the current fragment in execution is *Fragment1*, the pick in the blue line means the usage of the network for data synchronization of the previous fragment and the green line without the pick means that the execution is remote.

The last line in red defines the battery consumption trend according to the operation executed by the mobile device. The variation of the climb degree indicates different usages of energy according to the action

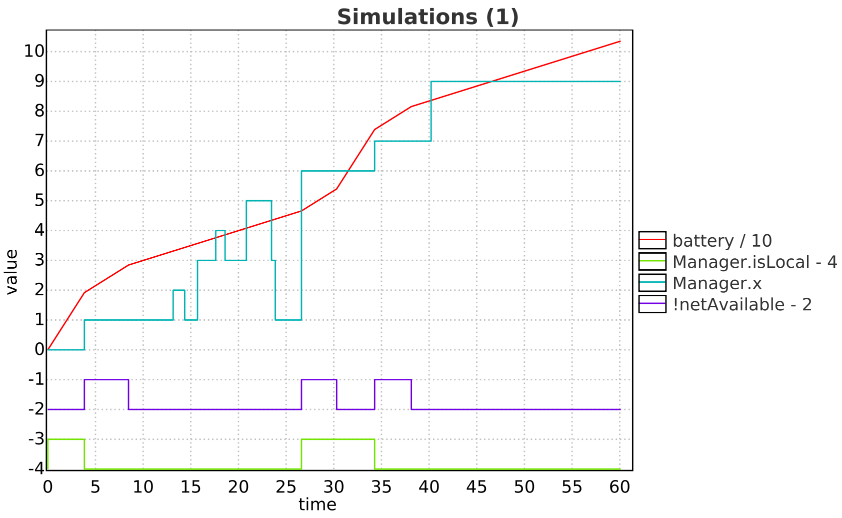


Figure 24: Strategy simulation

executed by the mobile device.

```
E[<=100; 200] (max: time) under fastE

simulate 1[<=60] {battery/10, Manager.isLocal−4, Manager.
    fragmentID, !netAvailable−2} under fastE
```

Listing 4.3: Results queries

Chapter 5

Parallelized MCC Scheduling Algorithms

In this chapter we describe further experiments based on algorithms developed ad-hoc for defining scheduling on MCC applications using parallel computation in Java language.
The definition of the optimal scheduling at run-time has many challenges related to response time and resource usage. In order to improve the performance of the system described in Section 4.2.2 we substitute the model checking engine in the solver module, with a smart and parallel algorithm developed on purpose. The issues related to the proposed algorithms are similar to the limitations of the model checking technique, like the state space explosion and so memory overflow. Here we try to improve the performance using parallel computation and some heuristics for pruning the tree, avoiding the exploration of unreachable and unsafe states. These algorithms have been applied to a simplified version of MobiCa without parallel computation, so reducing the operators only to non-deterministic and sequential ones.

5.1 Parallel Dijkstra implementation

In this section we describe the parallel Dijkstra implementation, that is a variation of the Dijkstra algorithm (Dij59) for the shortest path. The idea is to take in input a MobiCa specification and transform its behaviour in a edge-weight graph. An edge-weighted structure is a graph model, where we associate weights (or costs) to each edge. Such graphs are natural models for many applications. Perhaps the shortest path is the most intuitive graph-processing problem that we encounter regularly in systems, as e.g. when we use a map application or a navigation system to get directions from one place to another. In fact, if the weighted navigator graph represent a road network, each edge represents one direction of a road between two intersections, and the weight of an edge could represent the road's length, the time required to travel the road, or the toll a vehicle pays to use a road. The **weight of a path** is the sum of the weights of the edges on the path, so, if the edge weight represents the travel time, the weight of a path indicates the total distance time to travel along that road. The **shortest path** from vertex u to vertex v is a path whose sum of edge weights is minimum over all paths from u to v. To convert a MCC context into a edge-weight graph, we can model fragments as vertices and dependences between fragments as weighted edges. The weight of each edge is a generic cost that in a MCC context can be identified as the time or energy for executing the source fragment of the edge in a resource (mobile/cloud). In the weighted graph that we derive from MobiCa, we want a shortest path containing the lowest-cost way to get from the source fragment, which we call s to a target fragment t at depth greater than `k`. The `k` value indicates a threshold in the exploration of the graph, generally contains the number of levels to explore, but can also states cost variables like time and energy.

5.1.1 Properties of shortest paths

The shortest path problem requires some properties that should be respected for the formulation of the algorithm.

Paths are directed. A shortest path must respect the direction of its edges.

The weights are not necessarily distances. The weights might represent time, cost or an entirely different kind of variable.

Not all vertices need to be reachable. If t is not reachable from s, there is no path at all, and therefore there is no shortest path from s to t.

Negative weights introduce complications. We assume that edge weights are positive.

Shortest paths are not necessarily unique. There may be multiple paths of the lowest weight from s to t; we choose the first one found.

The runtime optimal scheduling we introduce in Chapter 4 can be reformulated as the single source shortest paths, where the source of the graph corresponds to the current state of our application. The result of the unfolding for the MobiCa specification is a tree, known as the shortest-paths tree (SPT), which gives a shortest path from the current state of the application s to a vertex t after `k` steps. Such path always exists and in general there may be two or more paths of the same length connecting `s` to a vertex `t`.
From here on, since our graph is a tree, we use node instead of vertex.

Example 3 *Taking in consideration the example presented in Figure 7 we can image to unfold the SG behaviour and generate the infinite tree graph presented in Figure 25. The fragments are nodes and the dependence between fragments are edges. A single edge (u, v) inherits the weight (time/energy) of the execution of the fragment u in a specific resource.*

*Notice that we have four possible combinations of execution: fragment can be executed locally on mobile (**M**) remotely on cloud (**C**), locally plus the bus (**M+B**) or remotely plus the bus (**C+B**). The use of the bus is necessary after the execution for the synchronization of the results. By convention, a fragment when is started in a resource should terminate in the same location. In reality, this is not always the case; it can happen that the remote execution is stopped for continuing locally due to connection lost.*

The execution representation for the Example 1 at page 35 starts with an empty root that connects the fragment f_0, followed by fragment f_2, an proceed by an infinite loop $f_2 \rightarrow f_0 \rightarrow f_2 \rightarrow f_1 \rightarrow f_2$. In the unfolding we find at

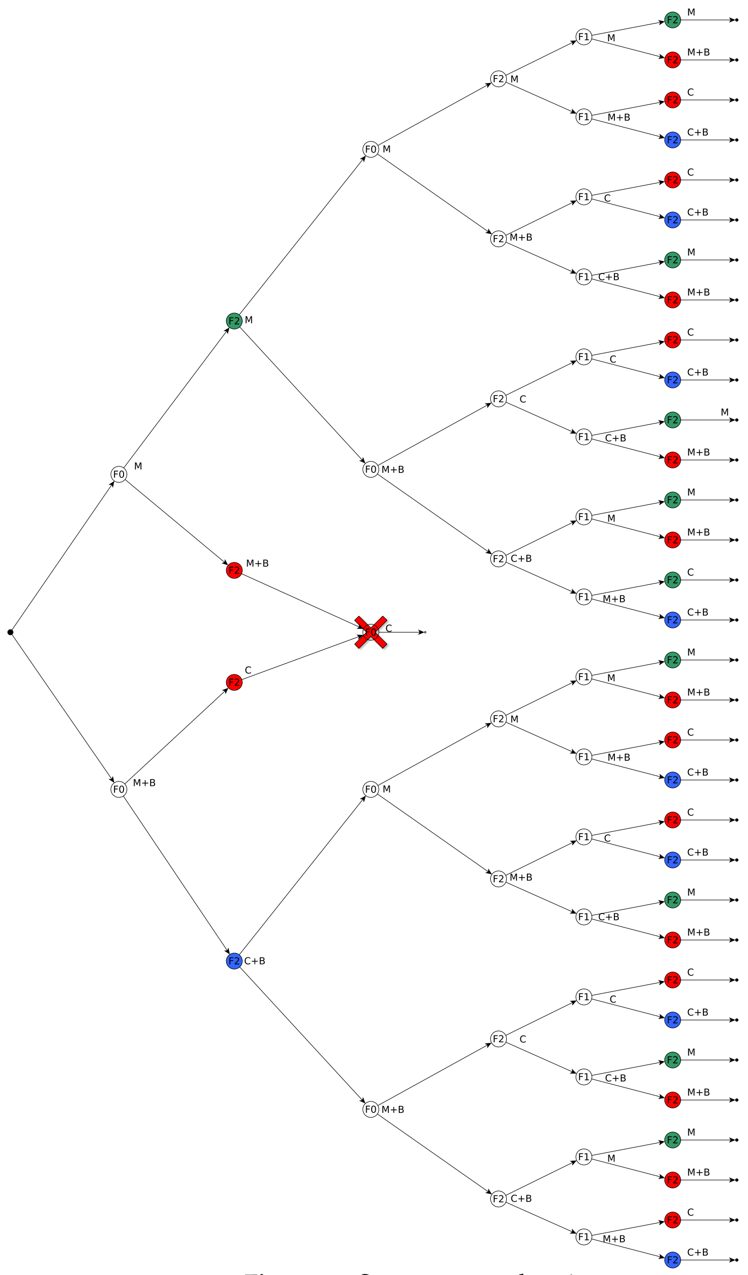


Figure 25: State space exploration

*the first level only two instances of f_0 reached by two transitions labeled with **M** and **M+B**. This because the fragment f_0 is not offloadable and so can be executed only in the mobile with or without synchronization on the bus.*

*The tree from f_0 continues towards f_2, that appear at the second level. Since f_2 is offloadable, potentially we have four possible execution location, indeed, if f_0 was on **M**, f_2 can be executed in **M** or **M+B**, else if it was on **M+B**, f_2 can be executed in **C** or **C+B**.*

*Going ahead with the third level in the tree of our example, we encounter again f_0; here, from f_2 **M** we move towards f_0 **M** and **M+B**, from f_2 **C+B** we have f_0 **M** and **M+B**. Instead from f_2 **M+B** and **C** we finish in a illegal state f_0 **C** denoted with a red cross, because a non-offloadable fragment cannot be executed remotely. The construction of the tree continues towards f2 and f1 for then proceed the cycle restarting from the coloured f2 at level 2 and so on.*

Looking carefully the Figure 25 we can recognise some standard patterns on the construction of the tree, that are generated by some rules introduced in the algorithms in order to reduce the state space.

These rules can be formalized as a set of heuristics: if a fragment is executed in **M**, the successor can be executed only in **M** or **M+B**, this because the system cannot execute two consecutive fragments in a different resource without the synchronization on the bus. For the same reason if a fragment is executed in cloud the successor can be executed only in **C** or **C+B**.

The second heuristics instead is related to the data synchronization. If a fragment after its execution synchronizes the data on the bus, so we are in the case **M+B** or **C+B**, the successor can be only **C** and **C+B** in the first case and **M** and **M+B** in the second case. This because it has not sense to synchronize the data on the bus and maintaining the execution on the same resource, we are loosing time for useless actions.

The last heuristic states that, if a fragment is not offloadable, so executable only in **M** and **M+B**, its predecessor can be executed only on **C+B** or **M** for not lead in illegal states.

The presented heuristics, resumed in Table 7, are very useful in MCC applications in order to simplify the tree and reduce the state space. Of course this solution does not avoid the explosion of the graph state space, but it permits a deeper analysis using less resources in terms of time

Heuristic	Predecessor	Successor	
		offloadable	not offloadable
1	M	M / M+B	M / M+B
	C	C / C+B	illegal
2	M+B	C / C+B	not convenient/illegal
	C+B	M / M+B	M / M+B

Table 7: Unfolding heuristics

and memory. In the Figure 25, for the sake of readability, we limit our exploration at level 6; of course is possible to proceed the exploration by simply applying the defined heuristics. The cyclic behaviour of the Example 1 is reflected in the tree exploration, indeed we can find circular behaviour every four levels of the generated graph. For having a more clear idea of the graph continuation we can imagine to append to each leaf f2 the corresponding subgraph delineated by a coloured root f2 starting at level 2. The correspondence of the colours are important for guaranteeing the respect of the graph construction according to the formulated heuristics.

After having presented the state space exploration, we try to merge this concept with our parallel Dijkstra implementation. Since the exploration of the tree can be infinite, in the algorithm we consider always a bound k imposed by the developer, that limit the exploration to reach a given level in the tree from the current state. This permits to have a good scheduling prediction in relative short time. The tree in the algorithm is constructed on-the-fly for keeping the memory usage under control and avoiding memory overflow.

Before describing the parallel algorithm that we propose, here below we recall the Dijkstra algorithm customized ad-hoc for the shortest path in a MCC tree. By convention, we will name s the source node and t the target one. In our Dijkstra implementation we want to compute three things for each node t. First the weight of a shortest path from s to t

which we denote as total cost *totalcost(t)*. Second, the predecessor of t on a shortest path from s to t, which we call *pred(t)*, and the depth of the node t from s, that we call *depth(t)*. To compute the shortest path from a source node to the target one we apply a set of relaxation steps to the edges of the tree.

Definition 9 ***Relax(u, v)***

- *Inputs: u, v nodes such that there is and edge (u,v).*
- *Output: If totalcost(u) + weight(u, v)<totalcost(v), then totalcost(v)=totalcost(u) + weight(u, v) and pred(v)=u*

The Dijkstra algorithm maintains a set Q of unvisited nodes for which the *totalcost, pred* and *depth* values are not yet known, instead, all nodes not in Q have their final *totalcost, pred* and *depth* values. In the algorithm we implement the set Q of the unvisited nodes as a priority queue. Notice, since we are using this algorithm at runtime, with an infinite state space, we do not know all future nodes of our tree. For this reason the unvisited set is constructed on-the-fly and contain only all neighbourhoods of the visited nodes that are possible candidates for being the next to be relaxed and added to the visited queue.

The algorithm, after initializing *totalcost(v)* to ∞, *pred(v)* and *depth(v)* to `null` for all nodes except the source s, it repeatedly finds the vertex u in set Q with the lowest *totalcost* value, removes that node from Q, adds all the successors to the set Q with *depth* equal *depth(u)+1* and then relaxes all the edge leaving u. The cycle continues until the set Q is empty or until the depth is equal to the threshold `k` setted by the developer.
In Listing 5.1, we list the details of the presented Dijkstra(G, s) algorithm using the pseudocode.

We now show, how the revisited Dijkstra algorithm for MCC application can be implemented in a arbitrary number of threads/processors. The parallel Dijkstra at its initialization requires three parameters, and in particular the number of threads, the number of starting nodes and the depth that we want to reach in computing the shortest path.

```
for each node v ∈ G: totalcost(v) ← ∞, pred(v),depth(v)←null;
add s to Q;
while Q not empty or depth(v)<=threshold
        u←ExtractMin(Q);
        for each nodes v such that u→v
                depth(v)=depth(u)+1;
                prec(v)←u;
                add v to Q
                Relax(u, v)
```

Listing 5.1: Dijkstra algorithm

The algorithm starts the state space exploration with a breath first search (BFS) algorithm until it is reached a level in the tree where the number of discovered nodes are greater than or equal to the number of the starting nodes setted by the user. Referring to the unfolding in Figure 25, if the user sets the number of starting nodes equal to 2, the system stops the BFS after discovering the first two nodes *f0* at level one, if he sets the starting node equal to 3 the system will find four nodes *f0* at level 3, instead if he sets the number to 5 the system will explore the tree until the nodes *f2* at level 4. This happens because the BFS should discover a level in the tree with a number of nodes greater than or equal the number of starting nodes required by the user. During the BFS exploration the algorithm keeps track of the *totalcost*, *pred* and *depth* values for each visited node. When the BFS reaches a depth level sufficient to cover the starting nodes, for each node present in the last level visited, it instantiates a new Dijkstra implementations explained in Listing 5.1 passing the corresponding node. So if we return to the example with starting node equal to, 2 the system will explore until level one and it will execute two instantiations of Dijkstra in parallel, one with *f0* **M**, and one with *f0* **M+B** as source nodes. Once a Dijkstra instantiation reaches the imposed depth it will send back the target nodes achieved to the caller. The caller maintains in memory only the target node *t* with the minimum *totalcost(t)* arrived until that moment, and each time it receives a new performing node *t* it substitutes the node and notifies in broadcast the total cost to all the Dijkstra instantiations still running. This notification permits to interrupt the computation of threads that have not yet reached the threshold but with a total cost greater than that of the candidate node notified by the caller.

The caller, once collected all the target nodes from all instantiations, selects the node t with the lowest *totalcost(t)*. At this point starting from the selected node and looking the *pred(v)* for each node v from t to s, it is able to construct backwards the shortest path. This shortest path is the candidate optimal schedule for the MCC application analyzed.

The proposed algorithm, thanks to the parallel computation and using the unfolding heuristics, has permitted to achieve the same results of the model checking approach presented in Section 4.2.1, but with less time and permitting the exploration to a longer horizon. This guarantees a better prediction strategy for the MCC applications.

The main problem related to this algorithm is connected to the memory usage during the analysis. Indeed in the Dijkstra implementation we have all visited vertex maintained in memory. In some case we run into memory overflow, despite we do not create all the state space but only what is required.

The solution to such problem could be to prune the priority queue, eliminating nodes that are old and with high cost, but we did not find yet a good rule for being sure to of not loosing part of our tree that potentially can lead to an optimal solution.

5.2 Parallel Depth First Search

In this section we propose a different algorithm that tries to save memory, but giving always good performance. The new algorithm is based on the same concepts of the previous one, but with the difference that each tread instead of starting to compute a Dijkstra shortest path applies a depth first search (DFS) in the depth of the tree. Of course the algorithm will construct only the necessary sequence of vertices until the threshold, respecting the same construction heuristics presented in Table 7.

When the algorithm reaches the threshold for the first path, it saves the sequence of vertices with the accumulated cost as candidate optimal scheduling and proceed looking the next one. The system continues its exploration and each time it reaches the end of the tree with a cost lower than the optimal candidate, it replaces the old sequence with the newest one and frees the memory occupied by the rejected path. Once the thread has covered all the committed subgraphs, it sends back the partial optimal result for that portion of the tree. As before, the global optimal path is the one with the lowest cost between all partially optimal paths achieved by each thread.

5.3 Analysis of the proposed algorithms

In this section we asses the performance of the two parallelized algorithms with respect to the UPPAAL method for the minimization of reachability properties described in Section 4.2.1.

The results presented, were executed on a Intel Core i7 Q740 1.73GHz with 4 cores (8 threads) processor, and with a fixed amount of RAM in the JVM equal to 2 Gb.

Table 8 depicts the performance of the three algorithms at work for finding the optimal scheduling on the Example 1 at page 35. The results are divided according to the depth of the schedule. Notice in the first column we report the number of the starting nodes, that corresponds to the number of instantiations for the parallel algorithms and the depth (i.e the pattern 2-10 indicates two starting nodes at depth ten). Comparing the results, we can state that the UPPAAL approach is the less performant. Using 2 Gb of memory it is able to reach a depth of 40 with 221.23 seconds. Forty is the maximum depth achieved by UPPAAL on the presented example before running into memory overflow. Passing to the parallel algorithms we have that the parallel DFS is the best performant, reaching greater depth, with a reasonable amount of spent time. The parallel Dijkstra algorithm instead performs well with low depth, but suffers the state space explosion like in the UPPAAL case; despite this issue it is able to compute the optimal schedule with a depth of 45 in 28.859 seconds.

The presented results give a clear idea of how the proposed algorithms scale with respect to the increasing of the depth. But to assess more concretely the scalability of the proposed algorithms, in Figure 27

StartingNodes-Depth	UPPAAL	P. Dijkstra	P. DFS
2-10	0.855	0.017	0.024
2-20	9.071	0.127	0.122
2-30	70.930	0.304	0.300
2-40	221.230	11.766	0.915
2-45	out	28.859	2.500
2-50	out	out	13.475
2-55	out	out	99.685

Table 8: Performance results in seconds for Example 1

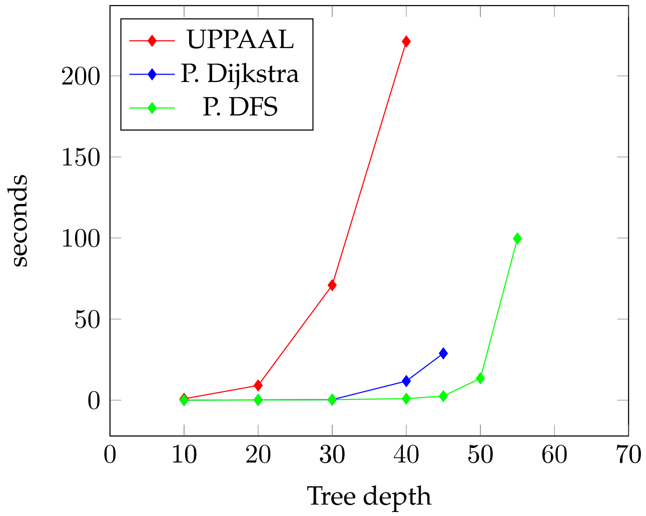


Figure 26: Graphical representation of results in Table 8

we propose a new example, where we increased the number of fragments and iterations. In this example all the fragments are intentionally offloadable in order to grow the complexity of the state space and limit the advantages brought by the usage of the heuristics in the parallel algo-

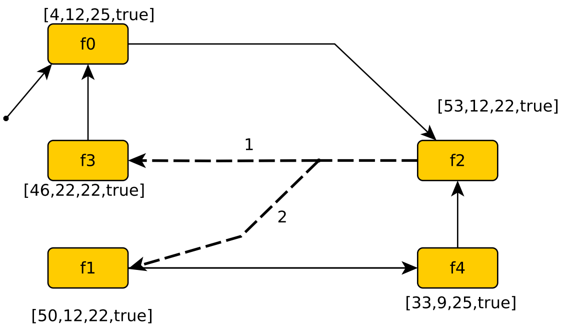


Figure 27: Another example of a MobiCa application

rithms. The results shown in Table 9 reconfirms the global performance obtained in the previous case, with an evident decreasing of performance for all the algorithms, especially for the UPPAAL one. In this case UPPAAL stops for out of memory at depth 30; it is clear that the model checking is penalized by the dimension of the model. For the other two algorithms we do not have sensible performance degradation. What we notice in Dijkstra implementation is the maximum depth reached, that is 40 instead of 45, with an amount of second that is doubled with respect to the previous case. Also the parallel DFS algorithm has shown a degradation in performance with the expanded model; despite of this inconvenient, it is always able to generate an optimal schedule without arising in errors for lacking of memory until much greater depth.

We present now further experiments on the Example 1 using the Dijkstra implementation but with different input parameters. Figure 29 represents an histogram containing on the y-axis the starting nodes and on the x-axis the elapsing of time in seconds. From the histogram it is possible to see that increasing the starting node from 2 to 4 with a depth of 45 we improve the performance from 28.859 of Table 8 to 12.89. Increasing more the number of the starting nodes to 80 the algorithm improves its performance by concluding the computation in 6.72 seconds. Of course we cannot exaggerate in increasing the number of starting nodes, indeed in the 200-45 case, the algorithm lost a bit of performance getting a result

StartingNodes-Depth	UPPAAL	P. Dijkstra	P. DFS
2-10	0.720	0.029	0.029
2-20	31.44	0.144	0.090
2-30	out	0.584	0.348
2-40	out	23.550	6.128
2-45	out	out	46.275
2-50	out	out	49.170
2-55	out	out	1089.988
2-60	out	out	2562.436

Table 9: Performance results in seconds for the example in Figure 27

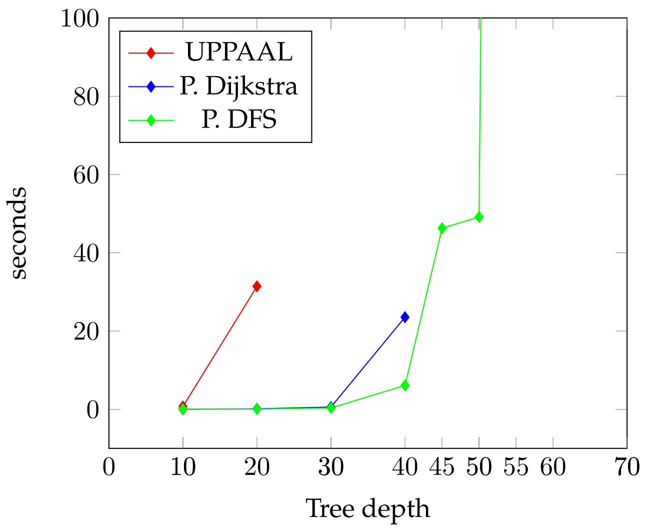


Figure 28: Graphical representation of results in Table 9

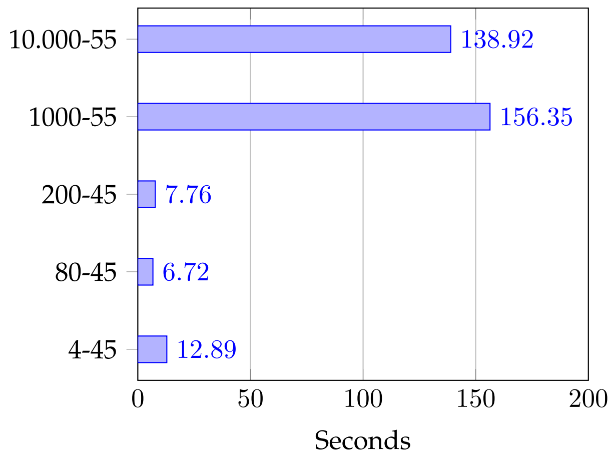


Figure 29: Execution seconds for the P. Djkstra algorithm on Example 1

of 7.76 seconds. In the setting of the parameters we should find the right compromise between exploration and exploitation of the state space in order to obtain the best results. With the increasing of the starting nodes parameter, the algorithm explores more tree using the BFS, reducing the computation assigned to the Dijkstra algorithm for reaching the target node.

Adjusting these settings, we improved the performance also for the 1000-55 and 10.000-55 configurations. Arranging this new value for the starting node, the algorithm reaches a depth of 55 with respect to the 45 achieved in Table 8 with the 2-45 configuration. The better result obtained in the 10.000-55 configuration is due to the good performance of the Dijkstra algorithm in exploring less levels of the state space.

To sum up, the UPPAAL method is less efficient with respect to the algorithms implemented in Java, but at the same time it is more flexible and adaptable to any kind of user necessity. In fact, the analysis is parametric with respect to the verified property, easily expressed as a logical formula. Furthermore, the model checking approach is able to deal with parallel operators in MobiCa, not straightforward to implement in the Java algorithms.

From the other side we have two parallel algorithms developed ad-hoc for defining optimal scheduling in MCC applications, that are able to benefit of construction heuristics defined on purpose for MCC systems. Comparing the two algorithms we found that the Parallel DFS is faster than the Dijkstra implementation. This result is due to a more slim implementation and less demanding data structure. Another advantage is related to the memory use: the DFS algorithms requires very low quantity of memory, each thread can contain in memory at most a number of nodes present in one path. It usually computes better than the Dijkstra implementation because it does not require the ordering of vertices in the priority queue, that for big state space is a real bottleneck. The usage of parallel algorithms have increased the performance for both the Dijkstra and the DFS methodology. Quantify the total benefit it is a really difficult task strongly related to the execution environment and the computation power of the infrastructure. Just to mention the cost/benefit impact of the algorithms, we can consider the results of the P. Dijkstra in Table 8 considering the 2-45 case, the time required for computing the schedule is 28.859 seconds using two threads, comparing this result with the 4-45 case in Figure 29, we have a reduction of time to 12.89 using 4 threads. So the parallel computation generates an improving of performance equal to the execution time on a single processor divided for the number of threads. Notice, in our case since we have a processor with 4 cores the best performance can be achieved with 4 threads. Increasing the number of threads the performance in some cases remains stable or can be decreased due to a more complex management of the context switching for the processor. We register the same result also for the parallel DFS case in the worst case. In the other cases it is not simple to define the ratio between cost/benefit, because the results are influenced also by the random choice of the algorithm that can lead to exploit mainly the optimization on the best result introduced in the algorithm, improving considerably the global system performance.

For the sake of readability the source code of the implemented algorithms is omitted here, we refer the interested reader to the MobiCa website (Mob15).

Chapter 6

Related Work

MCC has been already presented as a promising solution in the mobile application field. In particular, we have shown that it allows both data processing and data storage to happen outside the mobile device at runtime. This new solution can be confused with the cloud computing; the main difference is that the latter aims at providing rich elastic computing resources that are defined at design time, instead the MCC permits to expand the functionality on the go and on demand according the environment parameters.

According the (FLR13) survey the MCC methods are based on three offloading technique that consists in: Client-Server, Virtualization, and Mobile Agent. The former techniques relies on the communication between mobile devices and a remote server/cloud via protocols such as Remote Procedure Calls (RPC). This methodology is largely used in web development, but required a pre-installed system between the participants. This constraint is a clear disadvantage in a MCC scenario because limits the dynamic nature of mobile applications. Spectra (FPS02) and Chroma (BSPO03) are typical example of a Client-Server architecture.
Virtualization is a technique that permits to transfer the memory image of a virtual machine from a source device to a destination server. The advantages of this method is the migration of applications and OS without stopping their functionalities. The main works that rely on virtual-

ization are MAUI (CBC$^+$10b) and CloneCloud (CIM$^+$11). Mobile Agent is another methodology that uses a mobile code approach to partition and distribute jobs; under this paradigm we found also the Cloud Personal Assistant (CPA) introduced in by O'Sullivan et all. (OG13b). The authors assign at the CPA the responsibility for discovering an appropriate service searching on a repository, invoke it with the corresponding parameters and notify the results back from the cloud to the mobile.

MCC has many advantages, but at the same time has to face with many issues due to the integration of different architectures. Thus, it inherits also all the challenges behind the cloud computing and the application mobility. The main issues in MCC are related to:

- Low bandwidth: It is the most important parameter for the success of the code offloading and for taking advantages from the MCC techniques.

- Availability: Service availability can compromise the computation executed until that moment and sometimes this can cause loss of data.

- Heterogeneity: MCC will be used in heterogeneous network, like 3G and wi-fi, and the application should support different communication protocol. Furthermore the heterogeneity of networks with different power consumption will require different analysis for decide if to offload or not.

- Computation offloading: This is the key concept in MCC. However it is a very delicate phase in the application life and it is influenced by the external environment. The offloading can be applied in a static environment where the connection is stable and so it is possible to have also a static application partition. Moreover it can also be applied in a dynamic environment, where the connection is fluctuating and, for taking advantage from the offloading the system should decide the granularity of the partitioning and which components to offload.

- Security: It is another important issue in MCC. The MCC paradigm inherits advantages from cloud computing and mobile device, but of course has a drawback in data and user security. Among the security risks for mobile users we found all the possible attacks that can be done to a mobile phone like trojan, virus, etc. For the data, instead, we have problems of privacy connected to the reliability of cloud that can be due also to the integrity of the data.
- Quality of services: It is a concept that acquires importance with the cloud computing. Quality of services is regulated by means of contracts that guarantee no limitation in congestions, bandwidth reduction and disconnections.
- Pricing: In MCC there are two players on which a mobile phone can rely: mobile service provider and cloud service provider. Of course they have different characteristics, method and price.

In the literature we found many of mobile frameworks that use a range of approaches to achieve these points. To better explain how the existing frameworks approach them, we structure the rest of the section as follow: first in Section 6.1 we examine the different optimization metrics to consider and the resources used to quantify these choices; in Section 6.2 we describe the different partitioning methods used and in Section 6.3 we discuss the proposed offloading strategies proposed in the literature.

6.1 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 (RRPK99), which is one of the earliest systems to support dynamic application partition, attempts to improve battery lifetime by migrating large tasks on server machines. On the other hand, more innovative frameworks like Odessa (RSM$^+$11) try to improve system performance using a greedy algorithm to manage data parallelism, stage offloading and pipeline parallelism. CloneCloud (CIM$^+$11) can minimize both execution time and energy used for a target computation. MAUI (CBC$^+$10b) and Cuckoo (KPKB12) consider both energy and execution time. Chroma (BSPO03) weighs both execution time and fidelity when making decisions. Instead Spectra (FPS02) 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.

6.2 Partitioning methods

The partitioning is the step after the optimization of the metrics, where the developer should decide a method for splitting the application between offloadable and not offloadable parts. Theoretically, the number of possible application partitions is very large since partitioning could employ granularities as fine as single instructions. However, from a practical point of view, very fine-grained partition are very ineffective.

In partitioning methods research area there is a considerable difference of opinion about how best divide the applications. There are some approaches like Chroma that ask to the developer to specify a given application partition. These approaches are based on the assumption that the number of fragments in which the application is split is small and, as consequence, the granularity of the application is typically larger. The authors of Chroma created a little language called Vivendi (BGSH07) that allows the developers to specify compact static partitions of the application. Using Vivendi, developers specify code components called "remoteops" that may benefit from remote execution given the right circumstances. Developers also specify "tactics", each of which represents a different way of combining remote procedure calls that are selected at runtime by the Chroma system.

Other frameworks like Spectra ask the developer to specify candidate partitions explicitly. The same for MAUI that considers candidate partitions specified at methods granularity. However, MAUI asks to the developers to specify which methods it should consider executable remotely by annotating those methods with the custom attribute feature provided by Microsoft Common Language Runtime. Then, at runtime, the system according to a solver decides if to offload or not the candidate method selected by the developer. A more extreme variant of this approach is presented in RPF, which considers only two candidate partitions, one performed locally and another remotely.

Other more innovative methods instead choose the partitions granularity without requiring any programmer assistance. These approaches takes advantage of modern virtual machine runtime language to identify application components that can be remotely executed and the data on which these components operate. CloneCloud offers the best example of this approach; it identifies candidate partitions through static analysis of code compiled to run in Android's Dalvik virtual machine. The static analysis in this framework distinguishes three kinds of situations: 1) tasks that access device hardware must execute on the platform that has those features; 2) tasks that access the same location of memory should be executed on the same device; 3) if a task is offloaded also the methods that it invokes should be offloaded. As another example, Odessa models the application as a data low level graph in which the nodes are computational components called stages; it dynamically decides where stages should be placed and when to employ a parallel computation. Cuckoo is a Java based framework relying on the programming activity/service model existing in Android, it makes a distinction between compute intensive part (services) and interactive parts of the application (activities). Cuckoo requires the programmer a local implementation and then generates code for the same implementation on a remote server. Generally, remote and local methods are the same but may also be different, since the remote implementation can run in a different architecture.

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.

6.3 Offloading strategies

The last way for categorizing the MCC frameworks is according to the offloading strategy. Given a set of candidate partitions produced during the partition methods, the MCC system selects the one that it believes will best achieve its goals. This should be the partition that maximizes the system's chosen metric or utility function. For achieving the offloading strategy, the mentioned frameworks should rely on some mobile environment conditions like: network bandwidth, latency and server load but also application behavior like amount of required computation.

RPF relies on direct observation. In particular it requires that the application alternatively executes each partition and, by using an interface, is able to directly measure the drain of mobile device battery. After it has gathered sufficient measurements, RPF uses the partition with the lowest estimated energy cost most of the time, and occasionally executes the other partitions for seeing if the costs are changed. The drawback in this approach is that the system is not scalable, moreover the strategy requires sufficient measurements for all partitions before that RPF can start to choose the best one. Other problems in this approaches are that the prediction is application domain, and sometimes the prediction can be wrong due to the fact that the system has learned the cost of a partition on the same input for a period of time or because the data of the interesting partition are stale due to a not frequently usage by the user.

These problem in the direct observation have led to other approach that separately predict *resource supply* (resource available) and *resource demand* (resource required by the partition). Under this kind of approaches, it is possible to find MAUI. 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. For the resources supply MAUI estimates network according to the direct observation, sending in regular interval of time an amount of bytes and collecting the time used. Based on this data, MAUI predicts how much time will be spent for transferring an application state if it chooses to execute the method remotely. While the approach used by MAUI can adapt quickly to changes of network conditions, it is less agile to adapt to the changes of the application states. Often, the amount of states that must be transferred between the mobile and remote computers is input-dependent. When inputs vary, the size of the state transferred will change in response. MAUI suffers this limitation, like RPF that recognizes the state size only with the direct observation.

Other approaches that use the supply-demand method (NFS00) overcome this limitation with a history-based technique, which has been used in other MCC systems such as Spectra and Chroma. When history-based prediction is used, the application is sampled to explore its behavior under different inputs during its computation. Based on the profile data, machine learning algorithms are used to predict how application resource usage varies with fidelity and inputs. History-based prediction also requires some hints from the application developer. In particular, the developer must specify the inputs and other information related to the complexity of the partition, in order to have a final evaluation of the cost in terms of memory and CPU.

A similar approach is used by Cuckoo. The framework intercepts a method invocation using a proxy and then it will decide if invoking the method locally or remotely, using heuristics, context information and history.

A novel approach is proposed in Odessa, in which the system tries to identify first the performance bottleneck using an application profiler. Its decision engine uses simple predictors based on processor frequencies and recent network measurement to estimate whether offloading a component from the mobile computer to a remote server or increasing the level of parallelism for processing the component would improve the performance. This greedy incremental approach is also used in MARS (CLK$^+$11) for deciding which component to run remotely. It sorts tasks by the ratio of predicted local execution time to predict remote execution time. It offloads components with the highest such ratio first, subject to an energy constraint. This strategy is particular indicated for applications that exhibit high degree of parallelism.

Compared to above-mentioned works, the MobiCa approach takes decisions 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 configurations.

6.4 Scheduling via Timed Automata

The standard state-of-the-art approach to solve scheduling problems is to first represent the process graphically as a state task network or a resource task network and then to transform it into an optimization problem. The mathematical model of the optimization problem results normally in a Mixed Integer Nonlinear Programming (MINLP or MILP) since the corresponding mathematical models involve both discrete and continuous variables that must satisfy a set of linear or non-linear equality and inequality constraints. Few cases exist that can be modeled as linear problems (LP) or as integer problems (IP) with only real valued variables and discrete variables, respectively. The quality of the solution of the optimization problem strongly depends on the formulation of the model and a crucial role is played by the representation of time in the models. Based on the representation of time the mathematical programs can be generally classified into discrete time and continuous time models. The discrete time models are defined by dividing the scheduling horizon into finite number of time intervals with equal time durations and restricting the events that represent starting and finishing of operations to take place only at the margins of the intervals. The main disadvantages of discrete time models are the unnecessary increased number of binary variables which leads to an increased model size and the introduction of model inaccuracies due to the approximation of time. The inherent limitations of mathematical models with discrete time led to the development of models with continuous time representations. Due to the representation of time by continuous variables the models are more accurate compared to the discrete time models and also lead to the elimination of inactive event-time assignments.

An alternative method that has been discussed to model scheduling problems is to model the problem by sets of TA and to solve them by reachability analysis. Initially the purpose of TA was to model discrete event timed systems and to qualitatively analyze them. Reachability analysis is a verification technique to determine the reachable subset of states of the system modeled as TA, and evaluating whether formal properties such as safety or liveness are satisfied for the reachable states. The strength of the modeling approach is its graphical representation and modularity that helps in modeling complex systems with ease and clarity. Efficient and user-friendly tools such as Uppaal ((BLR05)) and Kronos ((Yov97)) have been developed for modeling and analysis of TA. In the context of scheduling, the modular nature of the modeling approach is exploited by modeling the resources and timing constraints as individual and independent timed automata and representing the interaction between the individual TA by synchronization labels. The individual automata modeled can then be composed automatically to represent the complete model of the scheduling problem. The composed automaton (i.e. the complete model of the scheduling problem) is represented by the reachability graph that includes an initial location where no operations have been started and at least one target location in which all operations are finished. The reachability analysis is performed to find a path from the initial state to the target state that fulfils an optimization criterion. In the context of scheduling, each path from the initial state to the target state represents a valid schedule and the path that fulfils an optimization criterion represents the optimal schedule.
The extension of timed automata to priced timed automata ((BFH$^+$01)) which include costs for transitions and cost rates for period of stay in locations, further motivated us to use the approach for scheduling problems as it allows to formulate more complex objective functions. Cost-optimal reachability analysis for priced TA aims at finding a path from the initial state to a target state with optimal cost. Specialized software

tools like Uppaal-CORA ((BLR05)) and TAOpt ((SE05)) were developed for cost-optimal reachability analysis of priced TA. Similar to many other approaches that solve scheduling problems, the TA-based approach also suffers from the problem of combinatorics thus demanding for efficient pruning procedures to reduce the search space. Reduction techniques to reduce the state-space were proposed in (SE05) for pure job-shops. In this work the authors introduced the idea of computing lower bounds for makespan minimization problems in job-shops by embedding linear programming (LP) into the cost-optimal reachability analysis to reduce the search space. This technique improved the overall performance to a great extent particularly for large scale problems as a huge part of the search tree was pruned due to the lower bounds computed by the LP.

Despite TA are able to describe very complex system, it is very complex and annoying to represent a MCC systems with such formalism. Furthermore the TA have to face with a really dynamic environment, composed by simple and structured constraints. For this reason, we develop a high-level language able to capture all these aspects, but at the same time to inherit all the advantages related to the expressiveness and the construction modularity of TA. Reachability is a very powerful property for defining optimal scheduling, and it gives the best results at runtime, because only at runtime it is able to foresee all the possible configurations that can start from a high dynamic environment like MCC systems. In our work we use reachability properties inside an ad-hoc framework, able to create the right environment for defining the optimal scheduling in every situation.

Chapter 7

Concluding Remarks

In this thesis we have presented a formal methodology useful for defining offloading strategies for MCC applications, to be used at design or run time. We have engaged the topic merely from the point of view of the developer interested in optimizing the system configuration. In particular we propose a framework able to improve the system performance, reduce the energy usage and so give a rich user experience.
As explained in this thesis, we have based our framework on the definition of a new language, MobiCa, able to express a MCC system in each of its relevant aspects. With this language, equipped with a formal semantics, we are able to automatically generate a global model that represents the entire MCC scenario under consideration. This model, created adhoc for being used as input in the UPPAAL model checker, will give the possibility to generate scheduling at run-time and design time and assess the results using quantitative analysis on the designed system. The developer both at design-time and at run-time, can rely on the results provided by the model checking technique, in order to perform the best offloading strategy for the designed application.
MCC is still a research field where many open challenges have not yet addressed. However, the need of MCC appears fundamental. This new technology connects two strong trends in the modern IT community: every day more people access data and run computations from their mobile

devices; data and computation is increasingly distributed among cloud infrastructures. Thus in this period where desktop computers and workstations are replaced by mobile and cloud infrastructures there is the necessity to move towards a more dynamic environment that is able to mitigate the set of limitations related to mobile devices.

7.1 Challenges and Future Work

This thesis faces a series of challenges in the MCC field, however there are many topics related to this work that are considered out of the scope for now and that will be integrated in this work in the future.
Security and privacy: Aspects that we have not touched are about security and privacy during the offloading mechanism. MCC users are naturally uncomfortable with the concept of storing personal data or performing sensitive computation on computer systems outside of their immediate control. Existing approaches for providing security and privacy are incomplete or unrealisable. The security depends on the manner in which MCC infrastructures are deployed. If the offloading is done towards a cloud provider, whom the mobile user already has a trust relationship, then the user may be more prone to send sensitive data or computations. However the user may be unwilling to send data to remote services without security and privacy guarantees. One possible solution for deploying data and computation in a secure infrastructure is to initially limit the functionalities provided by the cloud and then expand services gradually always controlled by means of a policy language or with some kind of contracts such as service level agreements.
Expand the benefit to more applications: as discussed, it is currently difficult to earn benefit for applications that do not require large amount of computation. If the computation of a fragment is small, the time and energy saved by performing the computation remotely is less than the time and energy used to synchronously communicate inputs to the cloud and receive back the results. This, unfortunately, often precludes many popular applications from taking substantial advantages in MCC. A promising idea is to develop new models for executing computation remotely

removing the need for synchronous communication. So, a possible future work could be to predict what inputs will be needed before the execution of a computation and send those inputs to the remote part in advance. Static analysis and code profiling potentially can help to make such prediction more accurate.
Failure handling: MCC systems have rudimentary support for handling failures and performance degradation. In this thesis we address the problem providing a reasonable solution in the run-time decision support using roll-back system. Another promising research direction could be replication. A mobile application can improve performance by executing the same computation in multiple location. The application can proceed its execution with the first result generated. Lateness executions can be aborted and their results discarded. This methodology of executing multiple instances of the same application consumes extra resources, so a MCC system should be able to judge when is fruitful to employ redundancy. In this direction we can consider to investigate smart procedures able to predict possible disconnections in order to minimize inconsistent situations and so avoid costly and heavy procedures of roll-back.
Towards the technology: The services provided by cloud infrastructures are still not widespread. Without MCC applications, there is little demand of remote services and viceversa; without services, there is a little incentive to develop MCC systems. For this reason, it is necessary that the development of MCC applications and the deployment on cloud infrastructures should be done simultaneously. The most likely motivation is the need to deliver better performance for existing applications than statically partition their functionality between mobile and cloud. A possibility should be a formal language like MobiCa, where once defined the system using the model specification, it automatically derived the global structure of the system, providing the skeleton of the implementation for both mobile and cloud part and, just leaving the developer to fill up the already optimized implementation templates. A promising work for the future is to extend the MobiCa implementation in Xtext introducing a language semantics that permit the code generation starting from the syntax presented here. This future work is very demanding in terms of

architecture knowledges, because the generated code should be multi-platform and self-adaptive to the environment.
Run-time Middleware: The framework integration is undoubtedly the next step that has to be taken in consideration for the future. The decision support algorithm presented in Section 4.2.3 should be integrated in a multi platform middleware able to include the runtime decision support developed using UPPAAL and the parallel algorithms presented in Chapter 5, that have been proved being faster than the model checking analysis. Under this optimization view, our intention is to investigate more on the algorithms analysis in order to verify, if the good results obtained for the MCC field can be broaden to a large area of scheduling optimization and strategy synthesis.
Aware Monitor The framework could be extend with a monitor accessible on the web where the final user can check the resource usage trend and define personalised policies in order to help the system to foresee possible disconnection, availability of a stable connection or the possibility to recharge the battery. Under this assumption the scheduler can optimize the system relying on information provided and so postpone or anticipate actions according to the received information.

Appendix A

UPPAAL Syntax

A.1 Declaration

```
Declarations   ::= (VariableDecl | TypeDecl | Function
                    |ChanPriority)*
VariableDecl   ::= Type VariableID (',' VariableID)* ';'
VariableID     ::= ID ArrayDecl* [ '=' Initialiser ]
Initialiser    ::= Expression|
                      '{' Initialiser (',' Initialiser)* '}'
TypeDecls      ::= 'typedef' Type ID ArrayDecl* (',' ID
                     ArrayDecl*)* ';'
```

A.2 Type

```
Type       ::= Prefix TypeId
Prefix     ::= 'urgent' | 'broadcast' | 'meta' | 'const'
TypeId     ::= ID | 'int' | 'clock' | 'chan' | 'bool'
              |  'int' '[' Expression ',' Expression ']'
              |  'scalar' '[' Expression ']'
              |  'struct' '{' FieldDecl (FieldDecl)* '}'
FieldDecl ::= Type ID ArrayDecl* (',' ID ArrayDecl*)* ';'
ArrayDecl ::= '[' Expression ']'|  '[' Type ']'
```

A.3 Function

```
Function        ::= Type ID '(' Parameters ')' Block
Block           ::= '{' Declarations Statement* '}'
Statement       ::= Block
                    | ';'
                    |  Expression ';'
                    |  ForLoop
                    |  Iteration
                    |  WhileLoop
                    |  DoWhileLoop
                    |  IfStatement
                    |  ReturnStatement
ForLoop         ::= 'for' '(' Expression ';' Expression ';'
                 Expression ')' Statement
Iteration           ::= 'for' '(' ID ':' Type ')' Statement
WhileLoop       ::= 'while' '(' Expression ')' Statement
DoWhile         ::= 'do' Statement 'while' '
                ('  Expression ')'';'
IfStatment      ::= 'if' '(' Expression ')' Statement [
                'else' Statement ]
ReturnStatement ::= 'return' [ Expression ] ';'
```

A.4 Espression

```
Expression ::= ID
               |  NAT
               |  Expression '[' Expression ']'
               |  Expression '''
               |  '(' Expression ')'
               |  Expression '++' | '++' Expression
               |  Expression '--' | '--' Expression
               |  Expression Assign Expression
               |  Unary Expression
               |  Expression Binary Expression
               |  Expression '?' Expression ':' Expression
               |  Expression '.' ID
               |  Expression '(' Arguments ')'
               |  'forall' '(' ID ':' Type ')' Expression
               |  'exists' '(' ID ':' Type ')' Expression
```

```
                |  'sum' '(' ID ':' Type ')' Expression
                |  'deadlock' | 'true' | 'false'

Arguments  ::= [ Expression ( ',' Expression )* ]

Assign     ::= '=' | ':=' | '+=' | '-=' | '*=' | '/='
                | '|=' | '&=' | '^=' | '<<=' | '>>='
Unary      ::= '+' | '-' | '!' | 'not'
Binary     ::= '<' | '<=' | '==' | '!=' | '>=' | '>'
                |  '+' | '-' | '*' | '/' | '%' | '&'
                |  '|' | '^' | '<<' | '>>' | '&&' | '||'
                |  '<?' | '>?' | 'or' | 'and' | 'imply'
```

A.5 Query language

```
Prop ::= 'A[]' Expression
                  | 'E<>' Expression
                  | 'E[]' Expression
                  | 'A<>' Expression
                  | Expression --> Expression
                  | 'sup' ':' List
                  | 'sup' '{' Expression '}' ':' List
                  | 'inf' ':' List
                  | 'inf' '{' Expression '}' ':' List
                  | Probability
                  | ProbUntil
                  | Probability ( '<=' | '>=' ) PROB
                  | Probability ( '<=' | '>=' ) Probability
                  | Estimate

List         ::= Expression | Expression ',' List
Probability  ::= 'Pr[' ( Clock | '#' ) '<=' CONST ']' '('
                  ('<>'|'[]')     Expression ')'
ProbUntil    ::= 'Pr[' ( Clock | '#' ) '<=' CONST ']' '('
                 Expression 'U' Expression ')'
Estimate     ::= 'E[' ( Clock | '#' ) '<=' CONST ';' CONST
                 ']''('('min:' | 'max:') Expression ')'
```

A.6 Stratego query language

```
//Strategy generators using Uppaal Opt
Minimize objective ::= strategy DS = minE (expr) [bound]:
                          <> prop under NS
Maximize objective ::= strategy DS = maxE (expr) [bound]:
                          <> prop under NS

//Strategy generators using Uppaal Tiga
Guarantee objective ::= strategy NS = control:  A<> prop
Guarantee objective ::= strategy NS = control:  A[] prop

//Statistical Model Checking Queries
Hypothesis testing ::= Pr[bound](<> prop)>=0.1 under SS
Evaluation         ::= Pr[bound](<> prop) under SS
Comparison         ::= Pr[bound](<> prop1) under SS1 >=
                        Pr[<=20](<> prop2) under SS2
Expected value     ::= E[bound;int](min:  prop) under SS
Simulations        ::= simulate int [bound]{expr1,expr2}
                        under S

//Model checking queries
Safety   ::= A[] prop under NS
Liveness ::= A<> prop under NS
```

Appendix B

UPPAAL case study

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE nta PUBLIC '-//UppaalTeam//DTDFlatSystem1.1
EN''http://www.it.uu.se/research/group/darts/
uppaal/flat-1_2.dtd'>
<nta>
<declaration>// Place global declarations here.
const int N = 10;          // Number of fragments.
typedef int[0,N-1] pid_t;

urgent broadcast chan hurry;
int battery=0;

int fragments=0;

const int P = 3;          // Number of fragments.
typedef int[0,P-1] pid_p;

urgent chan mobile, cloud,bus;
chan mobileR, cloudR, busR;

//const int costIdle[pid_p]={2,0,1};
const int costInUse[pid_p]={5,1,2};

typedef struct
{ int[0,150] execTL;
int[0,150] execTR;
int[0,200] memory;
```

```
int[0,100] syncT;
bool isOfflodable;
} Fragments;
const Fragments Info[N]={
{3,1,2,4,false},
{5,3,2,4,true},
{3,1,2,4,true},
{3,1,2,4,true},
{3,1,2,4,true},
{3,1,2,4,true},
{3,1,2,4,true},
{3,1,2,4,true},
{3,1,2,4,true},
{3,1,2,4,true}
};

bool activated[N]={true, false, false, false,false,false,
                false, false,false,false};
//0 never act., 1 act. local, 2 act. remotely
pid_t previous[pid_t];
int busyB;
pid_t mobileID, cloudID;
int result[pid_t][2];
clock time;
int[0,100]  memory=0;
bool loadM[pid_t];

const bool nd[pid_t][pid_t]={
        {0,1,0,0,0,0,0,0,0,0},{0,0,0,0,0,0,0,0,0,0},
        {0,1,0,0,0,0,0,0,0,0},{0,0,0,0,0,0,0,0,0,0},
        {0,0,0,1,0,0,0,0,0,0},{0,0,0,1,0,0,0,0,0,0},
        {0,0,0,0,0,0,0,0,0,0},{0,0,0,0,0,0,0,0,0,0},
        {0,0,0,0,0,0,0,0,0,0},{0,0,0,0,0,0,0,0,0,0}};
const bool par[pid_t][pid_t]={
        {0,0,0,0,0,0,0,0,0,0},{0,0,0,0,0,0,0,0,0,0},
        {0,0,0,0,0,0,0,0,0,0},{0,0,0,0,0,0,0,0,0,0},
        {0,0,0,0,0,0,0,0,0,0},{0,0,0,0,0,0,0,0,0,0},
        {0,1,0,0,0,0,0,1,0,0},{0,0,0,0,0,0,0,0,1,1},
        {0,0,0,0,0,0,0,0,0,0},{0,0,0,0,0,0,0,0,0,0}};
const int seq[pid_t][pid_t]={
        {0,0,0,0,0,0,0,0,0,0},{0,0,1,2,0,0,3,0,0,0},
        {0,0,0,0,0,0,0,0,0,0},{0,3,0,0,1,2,0,0,0,0},
        {0,0,0,0,0,0,0,0,0,0},{0,0,0,0,0,0,0,0,0,0},
        {0,0,0,0,0,0,0,0,0,0},{0,0,0,0,0,0,0,0,0,0},
        {0,0,0,0,0,0,0,0,0,0},{0,0,0,0,0,0,0,0,0,0}};


</declaration>
<template>
< breaklinesname x="5" y="5">Fragment</ breaklinesname>
<parameter>pid_t id</parameter>
<declaration>// Place local declarations here.
bool sequentialEnabled=false;
int counter=1;
clock t;

void parallel(){
        for(k:pid_t){
                if(par[id][k]){
                        activated[k]=true;
                        previous[k]=id;
                }
        }
}

void sequential(){
        sequentialEnabled=true;
        for(k:pid_t){
                if(seq[id][k]==counter){
                        activated[k]=true;
                        previous[k]=id;
                }
        }
        counter++;
        if(not exists(j:pid_t) seq[id][j]==counter){
                sequentialEnabled=false;
                counter=1;
        }
}

bool isLeaf(){
        return forall (i : pid_t) nd[id][i]==0 &amp;&amp;
        forall (i : pid_t) par[id][i]==0 &amp;&amp;
        forall (i : pid_t) seq[id][i]==0;
}

```

```
void fragmentUP(){
        fragments++;
}</declaration>
<location id="id0" x="-357" y="-136">
< breaklinesname x="-367" y="-170">Err</ breaklinesname>
</location>
<location id="id1" x="416" y="-192">
</location>
<location id="id2" x="416" y="48">
< breaklinesname x="406" y="18">completed</ breaklinesname>
<urgent/>
</location>
<location id="id3" x="112" y="160">
< breaklinesname x="102" y="130">runningR</ breaklinesname>
</location>
<location id="id4" x="112" y="-40">
< breaklinesname x="102" y="-70">runningL</ breaklinesname>
</location>
<location id="id5" x="-200" y="48">
< breaklinesname x="-210" y="18">ready</ breaklinesname>
<label kind="invariant" x="-178" y="42">t&lt;=100</label>
</location>
<location id="id6" x="-408" y="48">
< breaklinesname x="-418" y="18">null</ breaklinesname>
</location>
<init ref="id6"/>
<transition>
<source ref="id5"/>
<target ref="id0"/>
<label kind="guard" x="-339" y="-78">t==100</label>
</transition>
<transition>
<source ref="id2"/>
<target ref="id6"/>
<label kind="guard" x="-390" y="272">isLeaf()</label>
<label kind="assignment" x="-212" y="280">
        activated[id]=false</label>
<nail x="416" y="306"/>
<nail x="-408" y="306"/>
</transition>
<transition>
<source ref="id2"/>
<target ref="id6"/>
```

```
<label kind="guard" x="-391" y="476">exists(j:pid_t)
        seq[id][j]!=0</label>
<label kind="assignment" x="-127" y="476">sequential(),
        activated[id]=false</label>
<nail x="416" y="510"/>
<nail x="-408" y="510"/>
</transition>
<transition>
<source ref="id2"/>
<target ref="id6"/>
<label kind="guard" x="-399" y="416">exists(j:pid_t)
        par[id][j]==1 and sequentialEnabled==false</label>
<label kind="assignment" x="144" y="416">parallel(),
        activated[id]=false</label>
<nail x="416" y="442"/>
<nail x="-408" y="442"/>
</transition>
<transition>
<source ref="id1"/>
<target ref="id2"/>
<label kind="synchronisation"x="288" y="-216">busR?</label>
<label kind="assignment" x="208" y="-232">result[id][0]=1,
        result[id][1]=1</label>
<nail x="184" y="-192"/>
<nail x="264" y="-112"/>
</transition>
<transition>
<source ref="id2"/>
<target ref="id1"/>
<label kind="guard" x="416" y="-136">result[id][0]==0 or
        result[id][1]==0</label>
<label kind="synchronisation" x="416" y="-120">bus!</label>
<label kind="assignment" x="416" y="-105">
        busyB=Info[id].syncT</label>
</transition>
<transition>
<source ref="id2"/>
<target ref="id6"/>
<label kind="select" x="-390" y="317">e:pid_t</label>
<label kind="guard" x="-390" y="334">nd[id][e]==1and
        sequentialEnabled==false</label>
<label kind="assignment"x="-8" y="331">activated[id]=false,
        previous[e]=id, activated[e]=true</label>
```

```
<nail x="416" y="368"/>
<nail x="-408" y="368"/>
</transition>
<transition>
<source ref="id4"/>
<target ref="id2"/>
<label kind="synchronisation"x="200" y="0">mobileR?</label>
<label kind="assignment" x="208" y="-32">result[id][0]=1,
        fragmentUP()</label>
</transition>
<transition>
<source ref="id3"/>
<target ref="id2"/>
<label kind="synchronisation" x="236" y="112">cloudR?
        </label>
<label kind="assignment" x="224" y="128">result[id][1]=1,
        fragmentUP()</label>
</transition>
<transition>
<source ref="id5"/>
<target ref="id3"/>
<label kind="guard" x="-328" y="144">Info[id].isOfflodable
        ==true and result[previous[id]][1]==1</label>
<label kind="synchronisation" x="-184" y="160">cloud!
        </label>
<label kind="assignment"x="-272" y="176">cloudID=id</label>
</transition>
<transition>
<source ref="id5"/>
<target ref="id4"/>
<label kind="guard" x="-152" y="-64">
        result[previous[id]][0]==1 or id==0</label>
<label kind="synchronisation" x="-120" y="-40">mobile!
        </label>
<label kind="assignment" x="-176" y="-24">mobileID=id
        </label>
</transition>
<transition>
<source ref="id6"/>
<target ref="id5"/>
<label kind="guard" x="-376" y="32">activated[id]==true
        </label>
<label kind="synchronisation" x="-331" y="51">hurry!
```

```
        </label>
<label kind="assignment" x="-399" y="85">result[id][0]=0,
        result[id][1]=0, t=0</label>
</transition>
</template>
<template>
< breaklinesname>reset</ breaklinesname>
<location id="id7" x="0" y="0">
<label kind="invariant" x="-10" y="17">time&lt;=300</label>
</location>
<init ref="id7"/>
<transition>
<source ref="id7"/>
<target ref="id7"/>
<label kind="guard" x="-126" y="-93">time&gt;=300</label>
<label kind="assignment" x="-126" y="-59">battery=0,
        time=0, fragments=0</label>
<nail x="-144" y="-119"/>
<nail x="68" y="-144"/>
</transition>
</template>
<template>
< breaklinesname>Mobile</ breaklinesname>
<parameter> pid_p id</parameter>
<declaration>clock c;
void memoryUp(int i){
if(loadM[i]==0){
memory+=Info[i].memory;
loadM[i]=1;
}
}</declaration>
<location id="id8" x="-80" y="40">
< breaklinesname x="-104" y="56">inUse</ breaklinesname>
<label kind="invariant" x="-208" y="80">c&lt;=
        Info[mobileID].execTL</label>
</location>
<location id="id9" x="-80" y="-64">
< breaklinesname x="-90" y="-94">idle</ breaklinesname>
</location>
<init ref="id9"/>
<transition>
<source ref="id8"/>
<target ref="id9"/>
```

```
<label kind="guard" x="-496" y="8">c==Info[mobileID].execTL
        </label>
<label kind="synchronisation" x="-424" y="-16">mobileR!
        </label>
<label kind="assignment" x="-697" y="25">battery
        +=costInUse[id]*Info[mobileID].execTR
</label>
<nail x="-352" y="40"/>
<nail x="-352" y="-64"/>
</transition>
<transition>
<source ref="id9"/>
<target ref="id8"/>
<label kind="synchronisation" x="-68" y="-34">mobile?
        </label>
<label kind="assignment" x="-68" y="-51">c=0</label>
</transition>
</template>
<template>
< breaklinesname>Cloud</ breaklinesname>
<parameter>pid_p id</parameter>
<declaration>clock c;</declaration>
<location id="id10" x="-112" y="0">
< breaklinesname x="-122" y="-30">inUse</ breaklinesname>
<label kind="invariant" x="-122" y="15">c&lt;=
        Info[cloudID].execTR</label>
</location>
<location id="id11" x="-112" y="-128">
< breaklinesname x="-122" y="-158">idle</ breaklinesname>
</location>
<init ref="id11"/>
<transition>
<source ref="id10"/>
<target ref="id11"/>
<label kind="guard" x="-320" y="-104">c==
        Info[cloudID].execTR</label>
<label kind="synchronisation" x="-312" y="-87">cloudR!
        </label>
<label kind="assignment" x="-206" y="0">battery+=
        costInUse[id]*Info[mobileID].execTL</label>
<nail x="-224" y="0"/>
<nail x="-224" y="-128"/>
</transition>
```

```
<transition>
<source ref="id11"/>
<target ref="id10"/>
<label kind="synchronisation" x="-104" y="-87">cloud?
        </label>
<label kind="assignment" x="-104" y="-72">c=0</label>
</transition>
</template>
<template>
< breaklinesname>Network</ breaklinesname>
<parameter>pid_p id</parameter>
<declaration>clock c;
</declaration>
<location id="id12" x="-232" y="48">
< breaklinesname x="-212" y="42">inUse</ breaklinesname>
<label kind="invariant" x="-242" y="63">c&lt;=busyB</label>
</location>
<location id="id13" x="-232" y="-168">
< breaklinesname x="-242" y="-198">idle</ breaklinesname>
</location>
<init ref="id13"/>
<transition>
<source ref="id12"/>
<target ref="id13"/>
<label kind="guard" x="-496" y="-80">c==busyB</label>
<label kind="synchronisation" x="-496" y="-65">busR!
        </label>
<label kind="assignment" x="-646" y="-42">battery+=
        costInUse[id]*busyB</label>
<nail x="-416" y="48"/>
<nail x="-416" y="-168"/>
</transition>
<transition>
<source ref="id13"/>
<target ref="id12"/>
<label kind="synchronisation" x="-216" y="-80">bus?</label>
<label kind="assignment" x="-216" y="-65">c=0</label>
</transition>
</template>
<template>
< breaklinesname>Battery</ breaklinesname>
<location id="id14" x="-56" y="-32">
<label kind="invariant" x="-96" y="0">battery&gt;=0
```

```
        and memory&lt;70</label>
</location>
<init ref="id14"/>
</template>
<system>// Place template instantiations here.
Mobile_dev=Mobile(0);
Cloud_dev=Cloud(1);
Net=Network(2);
// List one or more processes to be composed into a system.
system Fragment, Mobile_dev, Cloud_dev, Net, Battery,reset;
}</system>
<queries>
<query>
<formula>A[]not deadlock
</formula>
<comment>
</comment>
</query>
<query>
<formula>A[]1==1
</formula>
<comment>
</comment>
</query>
<query>
<formula>E&lt;&gt;cycle&gt;=0 and Fragment(9).completed
</formula>
<comment>
</comment>
</query>
<query>
<formula>E[]forall(i:pid_t) not(Fragment(i).Err) and
        (time&gt;=300 imply (battery&lt;500
        and fragments&gt;10))
</formula>
<comment>
</comment>
</query>
<query>
<formula>E[](time==100 imply (battery&lt;250
        and fragments&gt;39))
</formula>
<comment>
```

```
</comment>
</query>
<query>
<formula>E[] forall(i:pid_t) not(Fragment(i).Err) and
        (battery&gt;=1000 imply time&gt;=1000)
</formula>
<comment>
</comment>
</query>
</queries>
</nta>
```

References

[AD94] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994. 8

[AILS07] Luca Aceto, Anna Ingólfsdóttir, Kim Guldstrand Larsen, and Jiri Srba. *Reactive systems: modelling, specification and verification*. Cambridge University Press, 2007. 13

[ALMT15] Luca Aceto, Kim G. Larsen, Andrea Morichetta, and Francesco Tiezzi. A cost/reward method for optimal infinite scheduling in mobile cloud computing. In *Formal Aspects of Component Software - 12th International Conference, FACS 2015, Niterói, Brazil, October 14-16, 2015, Revised Selected Papers*, pages 66–85, 2015. xii

[AMT15] L Aceto, A. Morichetta, and F. Tiezzi. Decision support for mobile cloud computing applications via model checking. In *MobileCloud*, volume 1, pages 296–302. IEEE, 2015. xii, 10

[BBL08] Patricia Bouyer, Ed Brinksma, and Kim G Larsen. Optimal infinite scheduling for multi-priced timed automata. *Formal Methods in System Design*, 32(1):3–23, 2008. 51

[BCD+07] Gerd Behrmann, Agnes Cougnard, Alexandre David, Emmanuel Fleury, Kim G Larsen, and Didier Lime. Uppaal-tiga: Time for playing games! In *Computer Aided Verification*, pages 121–125. Springer, 2007. 71

[BDL04] Gerd Behrmann, Alexandre David, and Kim G Larsen. A tutorial on uppaal. In *Formal methods for the design of real-time systems*, pages 200–236. Springer, 2004. 17

[BDL+12] Peter Bulychev, Alexandre David, Kim Gulstrand Larsen, Marius Mikučionis, Danny Bøgsted Poulsen, Axel Legay, and Zheng Wang. Uppaal-smc: Statistical model checking for priced timed automata. *arXiv preprint arXiv:1207.1272*, 2012. 55, 71

[Bet13] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd, 2013. 24

[BFH+01] Gerd Behrmann, Ansgar Fehnker, Thomas Hune, Kim Larsen, Paul Pettersson, and Judi Romijn. *Efficient guiding towards cost-optimality in uppaal*. Springer, 2001. 101

[BGSH07] Rajesh Krishna Balan, Darren Gergle, Mahadev Satyanarayanan, and James Herbsleb. Simplifying cyber foraging for mobile devices. In *Proceedings of the 5th International Conference on Mobile Systems, Applications and Services*, MobiSys '07, pages 272–285, 2007. 96

[BKMS13] Marco V Barbera, Sokol Kosta, Alessandro Mei, and Julinda Stefa. To offload or not to offload? the bandwidth and energy costs of mobile cloud computing. In *Proc. of IEEE INFOCOM*, 2013. 2

[BLR05] Gerd Behrmann, Kim G Larsen, and Jacob I Rasmussen. Optimal scheduling using priced timed automata. *ACM SIGMETRICS Performance Evaluation Review*, 32(4):34–40, 2005. 101, 102

[BSPO03] Rajesh Krishna Balan, Mahadev Satyanarayanan, So Young Park, and Tadashi Okoshi. Tactics-based remote execution for mobile computing. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*, MobiSys '03, pages 273–286, 2003. 92, 95

[CBC+10a] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. MAUI: Making smartphones last longer with code offload. In *MobiSys*, pages 49–62. ACM, 2010. 42

[CBC+10b] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: Making smartphones last longer with code offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 49–62, 2010. 93, 95

[CIM+11] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: Elastic execution between mobile device and cloud. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 301–314, 2011. 93, 95

[CLK+11] Asaf Cidon, Tomer M. London, Sachin Katti, Christos Kozyrakis, and Mendel Rosenblum. Mars: Adaptive remote execution for multi-threaded mobile devices. In *Proceedings of the 3rd ACM SOSP Workshop on Networking, Systems, and Applications on Mobile Handhelds*, MobiHeld '11, pages 1:1–1:6, 2011. 99

[Dij59] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959. 78

[DJL$^+$15] Alexandre David, Peter Gjøl Jensen, Kim Guldstrand Larsen, Marius Mikučionis, and Jakob Haahr Taankvist. Uppaal stratego. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 9035 of *Lecture Notes in Computer Science*, pages 206–211. Springer Berlin Heidelberg, 2015. 51

[DLL$^+$11] Alexandre David, Kim G Larsen, Axel Legay, Marius Mikučionis, Danny Bøgsted Poulsen, Jonas Van Vliet, and Zheng Wang. Statistical model checking for networks of priced timed automata. In *Formal Modeling and Analysis of Timed Systems*, pages 80–96. Springer, 2011. 55

[DLNW13] Hoang T. Dinh, Chonho Lee, Dusit Niyato, and Ping Wang. A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless Communications and Mobile Computing*, 13(18):1587–1611, 2013. 1

[EAWJ02] Elmootazbellah Nabil Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002. 68

[EXP] The mobile app economy is exploding. `http://venturebeat.com/2013/01/18/the-mobile-app-\economy-is-exploding-so-what-else-is-new/`. 1

[Fli12] Jason Flinn. Cyber foraging: Bridging mobile and cloud computing. *Synthesis Lectures on Mobile and Pervasive Computing*, 7(2):1–103, 2012. 1

[FLR13] Niroshinie Fernando, Seng W. Loke, and Wenny Rahayu. Mobile cloud computing: A survey. *Future Generation Computer Systems*, 29(1):84 – 106, 2013. Including Special section: AIRCC-NetCoM 2009 and Special section: Clouds and Service-Oriented Architectures. 1, 92

[FPS02] Jason Flinn, SoYoung Park, and M. Satyanarayanan. Balancing performance, energy, and quality in pervasive computing. In *Distributed Computing Systems, 2002.*, pages 217–226, 2002. 92, 95

[GK99] Flavius Gruian and Krzysztof Kuchcinski. Low-energy directed architecture selection and task scheduling for system-level design. In *EUROMICRO*, pages 1296–1302. IEEE, 1999. 35

[KL10] K. Kumar and Yung-Hsiang Lu. Cloud computing for mobile users: Can offloading computation save energy? *Computer*, 43(4):51–56, 2010. 2

[KPKB12] Roelof Kemp, Nicholas Palmer, Thilo Kielmann, and Henri Bal. Cuckoo: A computation offloading framework for smartphones. In Martin Gris and Guang Yang, editors, *Mobile Computing, Applications, and Services*, volume 76 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 59–79. Springer, 2012. 95

[KT87] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. *Software Engineering, IEEE Transactions on*, (1):23–31, 1987. 68

[MN10] Antti P Miettinen and Jukka K Nurminen. Energy efficiency of mobile clients in cloud computing. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 4–4. USENIX Association, 2010. 3

[Mob15] MobiCa. Mobica web page. `http://sysma.imtlucca.it/mobica/`, 2015. 91

[NFS00] D. Narayanan, Jason Flinn, and M. Satyanarayanan. Using history to improve mobile application adaptation. In *Mobile Computing Systems and Applications, 2000 Third IEEE Workshop on.*, pages 31–40, 2000. 99

[OG13a] Michael J. O'Sullivan and Dan Grigoras. User experience of mobile cloud applications - current state and future directions. In *IEEE 12th International Symposium on Parallel and Distributed Computing, ISPDC 2013, Bucharest, Romania, June 27-30, 2013*, pages 85–92, 2013. 6

[OG13b] Michael Joseph O'Sullivan and Dan Grigoras. The cloud personal assistant for providing services to mobile clients. In *Service Oriented System Engineering (SOSE), 2013 IEEE 7th International Symposium on*, pages 478–485. IEEE, 2013. 93

[Ran75] Brian Randell. System structure for software fault tolerance. In *ACM SIGPLAN Notices*, volume 10, pages 437–449. ACM, 1975. 68

[RLS06] Jacob Illum Rasmussen, Kim Guldstrand Larsen, and K Subramani. On using priced timed automata to achieve optimal scheduling. *Formal Methods in Syst. Design*, 29(1):97–114, 2006. 51

[RRPK99] Alexey Rudenko, Peter Reiher, Gerald J. Popek, and Geoffrey H. Kuenning. The remote processing framework for portable computer power saving. In *Proceedings of the 1999 ACM Symposium on Applied Computing*, SAC '99, pages 365–372, 1999. 94

[RSM$^+$11] Moo-Ryong Ra, Anmol Sheth, Lily Mummert, Padmanabhan Pillai, David Wetherall, and Ramesh Govindan. Odessa: Enabling interactive perception applications on mobile devices. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 43–56, 2011. 95

[SE05] Sebastian Panek Olaf Stursberg and Sebastian Engell. Job-shop scheduling by combining reachability analysis with linear programming. In *Discrete Event Systems 2004 (WODES'04): A Proceedings Volume from the 7th IFAC Workshop, Reims, France, 22-24 September 2004*, page 195. Elsevier, 2005. 102

[Yov97] Sergio Yovine. Kronos: A verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):123–133, 1997. 101