Contents

Chapter 1

Meeting Deadlines, Elastically

Einar Broch Johnsen
University of Oslo

Ka I Pun
University of Oslo

Martin Steffen
University of Oslo

S. Lizeth Tapia Tarifa
University of Oslo

Ingrid Chieh Yu
University of Oslo

1.1 Introduction

A cloud consists of virtual computers that are accessed remotely for data storage and processing. The cloud is emerging as an economically interesting model for enterprises of all sizes, due to an undeniable added value and compelling business drivers [11]. One such driver is *elasticity*: businesses pay for computing resources when needed, instead of provisioning in advance with huge upfront investments. New resources such as processing power or memory can be added to a virtual computer on the fly, or an additional virtual computer can be provided to the client application. Going beyond shared storage, the main potential in cloud computing lies in its scalable virtualized framework for data processing. If a service uses cloud-based processing, its capacity can be automatically adjusted when new users arrive. Another driver is *agility*: new services can be deployed on the market quickly and flexibly at limited cost. This allows a service to handle its users in a flexible manner without requiring initial investments in hardware before the service can be launched.

Today, software is often designed while completely ignoring deployment or based on very specific assumptions, e.g., the size of data structures, the amount of random access memory, and the number of processors. For the software developer, cloud computing brings new challenges and opportunities [21]:

- **Empowering the Designer.** The elasticity of software executed in the cloud gives designers far reaching control over the execution environment's resource parameters, e.g., the number and kind of processors, the amount of memory and storage capacity, and the bandwidth. In principle, these parameters can even be adjusted at runtime. The owner of a cloud service can not only deploy and run software, but also control trade-offs between the incurred cost and the delivered quality-of-service.

- **Deployment Aspects at Design Time.** The impact of cloud computing on software design goes beyond scalability. Deployment decisions are traditionally made at the end of a software development process: the developers first design the functionality of a service, then the required resources are determined, and finally a service level agreement regulates the provisioning of these resources. In cloud computing, this can have severe consequences: a program which does not scale usually requires extensive design changes when scalability was not considered a priori.

To realize cloud computing's potential, software must be *designed for scalability*. This leads to a new *software engineering challenge*: how can the validation of deployment decisions be pushed up to the modeling phase of the software development chain without convoluting the design with deployment details?

The EU project Envisage addresses this challenge by extending a design by contract approach to service-level agreements for resource-aware virtualized services. The functionality is represented in a *client layer*. A *provisioning layer* makes resources available to the client layer and determines how much memory, processing power, and bandwidth can be used. A *service level agreement* (SLA) is a legal document that clarifies what resources the provisioning layer should make available to the client service, what they will cost, and the penalties for breach of agreement. A typical SLA covers two different aspects: (i) the mutual legal obligations and consequences in case of a breach of contract, which we call the *legal contract*; (ii) the technical parameters and cost figures of the offered services, which we call the *service contract*.

This paper discusses some initial ideas about applying program verification techniques to models of virtualized services. We consider response time aspects of service contracts and extend JML-like interfaces with response time annotations. This is formalized using μABS; μABS is a restricted version of

ABS [25], an executable object-oriented modeling language used in the Envisage project to specify resource-aware virtualized services [4, 26, 27]. In particular, the work discussed in this paper is restricted to sequential computation and synchronous method calls whereas ABS is based on concurrent objects and asynchronous method calls. In future work, we hope to alleviate these restrictions.

Paper organization. Section 1.2 introduces service interfaces with response-time annotations; Sect. 1.3 introduces the syntax of μABS, the modeling language considered in this paper; Sect. 1.4 demonstrates the approach on an example; Sect. 1.5 develops a Hoare-style proof system for μABS; Sect. 1.6 discusses related work; and Sect. 1.7 concludes the paper.

1.2 Service-Level Interfaces

Service level agreements express non-functional properties of services (service contracts), and their associated penalties (legal contracts). Examples are high water marks (e.g., number of users), system availability, and service response time. Our focus is on service contract aspects of client-level SLAs, and on how these can be integrated in models of virtualized services. Such an integration allows a formal understanding of service contracts and of their relationship to the performance metrics and configuration parameters of the deployed services. Today, client-level SLAs do not allow the potential resource usage of a service to be determined or adapted when unforeseen changes to resources occur. This is because user-level SLAs are not explicitly related to actual performance metrics and configuration parameters of the services. The integration of service contracts and configuration parameters in service models enables the design of resource-aware services which embody application-specific resource management strategies [21].

The term *design by contract* was coined by Bertrand Meyer referring to the contractual obligations that arise when objects invoke methods [33]: only if a caller can ensure that certain behavioral conditions hold before the method is activated (the precondition), it is ensured that the method results in a specified state when it completes (the postcondition). Design by contract enables software to be organized as encapsulated services with interfaces specifying the contract between the service and its clients. Clients can "program to interfaces"; they can use a service without knowing its implementation. We aim at a design by contract methodology for SLA-aware virtualized services, which *incorporates SLA requirements in the interfaces at the application-level* to ensure the QoS expectations of clients.

We consider an object-oriented setting with service-level interfaces given in a style akin to JML [10] and Fresco [46]; **requires**- and **ensures**-clauses express each method's functional pre- and postconditions. In addition, a *response time*

```
type Photo = Rat; // size of the file

interface PhotoService {
    @requires ∀ p:Photo · p ∈ film && p < 4000;
    @ensures reply == True;
    @within 4*length(film) + 10;
    Bool request(List<Photo> film);
}
```

FIGURE 1.1: A photo printing shop in μABS.

guarantee is expressed in a **within**-clause associated with the method. The specification of methods in interfaces is illustrated in Figure 1.1.

1.3 A Kernel Language for Virtualized Computing

The μABS language supports modeling the deployment of objects on virtual machines with different processing capacities, simplifying ABS [4, 25, 27]: conceptually, each object in μABS has a dedicated processor with a given processing capacity. In contrast to ABS, execution in μABS is sequential and the communication between named objects is synchronous, which means that a method call blocks the caller until the callee finishes its execution. Objects are dynamically created instances of classes, and share a common thread of execution where at most one task is *active* and the others are waiting to be executed on the task stack. μABS is strongly typed: for well-typed programs, invoked methods are understood by the called object. μABS includes the types Capacity, Cost, and Duration which all extend Rat with an element infinite: Capacity captures the processing capacity of virtual machines per time interval, Cost the processing cost of executions, and Duration time intervals.

Figure 1.2 presents the syntax of μABS. A *program* P consists of interface and class definitions, and a main block $\{\overline{T\ x};\ sr\}$. Interfaces IF have a name I and method signatures Sg. Classes CL have a name C, optional formal parameters $\overline{T}\ \overline{x}$, and methods $\overline{M}$. A method signature Sg has a list of specifications $\overline{Spec}$, a return type T, a method name m, and formal parameters $\overline{x}$ of types $\overline{T}$. In specifications (see Sect. 1.2), assertions ϕ express properties of local variables in an assertion language extending the expressions e with logical variables and operators in a standard way; a reserved variable *reply* captures the method's return value. A method M has a signature Sg, a list of local variable declarations $\overline{x}$ of types $\overline{T}$, and statements sr. Statements may access local variables and the formal parameters of the class and the method.

Statements are standard, except **job**(e) which captures an execution requiring e processing cycles. A job abstracts from actual computations but may

Syntactic categories	*Definitions*		
C, I, m in Names	P	$::=$	$\overline{IF}\ \overline{CL}\ \{\overline{T\ x};\ sr\}$
s in Statement	T	$::=$	$C \mid I \mid$ Capacity $\mid$ Cost $\mid$ Duration $\mid$ Bool $\mid$ Rat
x in Variables	IF	$::=$	**interface** $I\ \{\ \overline{Sg}\ \}$
k in Capacity	Sg	$::=$	$\overline{Spec}\ T\ m\ (\overline{T\ x})$
c in Cost	$Spec$	$::=$	**@requires** $\phi; \mid$ **@ensures** $\phi; \mid$ **@within** $\phi;$
d in Duration	CL	$::=$	**class** $C\ (\overline{T\ x})\ \{\ \overline{M}\}$
b in Bool	M	$::=$	$Sg\ \{\ \overline{T\ x};\ sr\ \}$
i in Rat	sr	$::=$	$s;$ **return** $e \mid$ **return** e
	s	$::=$	$s; s \mid x = rhs \mid$ **job**$(e) \mid$ **if** $e\ \{s\}$ **else** $\{s\}$
	rhs	$::=$	$e \mid$ **new** $C(\overline{e})$ **with** $e \mid e.m(\overline{x})$
	e	$::=$	**this** $\mid$ **capacity** $\mid$ **deadline** $\mid x \mid v \mid e\ op\ e$

FIGURE 1.2: μABS syntax for the object level. Terms $\overline{e}$ and $\overline{x}$ denote possibly empty lists over the corresponding syntactic categories.

depend on state variables. *Right-hand sides rhs* include expressions e, object creation **new** $C\ (\overline{e})$ **with** e and synchronous method calls $e.m(\overline{x})$. Objects are created with a given *capacity*, which expresses the processing cycles available to the object per time interval when executing its methods. Method calls in μABS are *blocking.* Expressions e include operations over declared variables x and values v. Among values, b has type Bool, i has type Rat (e.g., 5/7), k has type Capacity, c has type Cost, and d has type Duration. Among binary operators *op* on expressions, note that division c/k has type Duration. Expressions also includes the following reserved read-only variables: **this** refers to the object identifier, **capacity** refers to the processing speed (amount of resources per time interval) of the object, and **deadline** refers to the local deadline of the current method. (We assume that all programs are well-typed and include further functional expressions and data types when needed in the example.)

Time. μABS has a dense time model, captured by the type Duration. The language is not based on a (global) clock, instead each method activation has an associated local counter **deadline**, which decreases when time passes. Time passes when a statement **job**(e) is executed on top of the task stack. The effect of executing this statement on an object with capacity k, is that the local deadline of every task on the stack decreases by c/k, where c is the value resulting from evaluating e. The initial value of the **deadline** counter stems from the service contract; thus, a local counter which becomes negative represents a breach of the local service contract. For brevity, we omit the formal semantics.

1.4 Example: A Photo Printing Shop

Let us consider a *photo shop* service which *retouches* and *prints* photos. It is cheaper for the photo shop service to retouch and print photos locally,

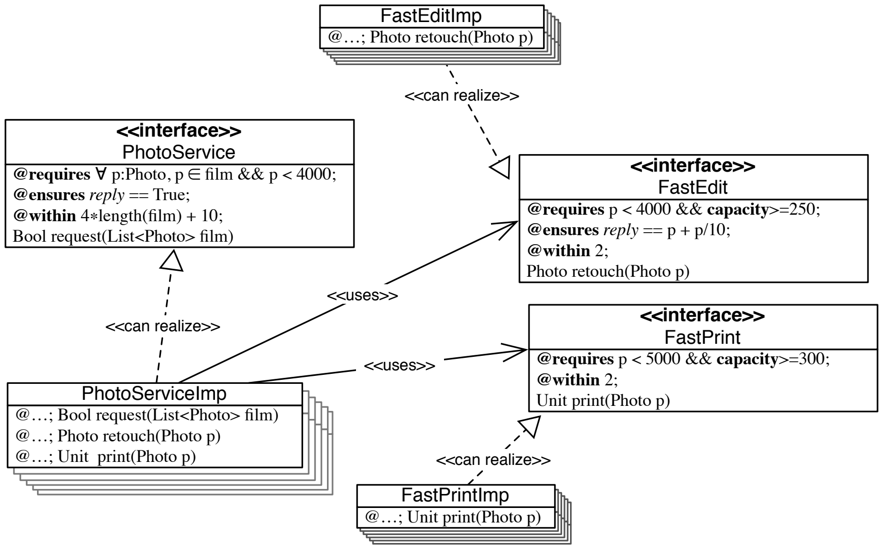

FIGURE 1.3: A class diagram for a photo printing shop

but it can only deal with low resolution photos in time. For larger photos, the photo shop service relies on using a faster and more expensive laboratory in order to guarantee that all processing deadlines are met successfully.

In this example, a film is represented as a list of photos and, for simplicity, a photo by the size of the corresponding file. As shown in the class diagram of Figure 1.3, an interface PhotoService provides a single method request which handles customer requests to the photo shop service. The interface is implemented by a class PhotoServiceImp, which has methods retouch for retouching and print for printing a photo, in addition to the request method of the interface. For faster processing, two interfaces FastEdit and FastPrint, which also provide the methods retouch and print, may be used by PhotoServiceImp. The sequence diagram in Figure 1.4 shows how a photo is first *retouched*, then *printed*. The tasks of retouching and printing are done locally if possible, otherwise they are forwarded to and executed by objects with higher capacities.

The μABS model of the example (Figure 1.5) follows the design by contract approach and provides a contract for every method declaration in an interface and method definition in a class. These specifications are intended to guarantee that a request to a PhotoService object will not break the specified contract. Looking closer at the contract for request, we see that the response time of a request(film) call depends on the length of the film and assumes that the size of every photo contained in the film is smaller than 4000. The implementation of the request method is as follows: Take the first photo in the film (by applying the function head(film)) and check if this photo is low resolution compared

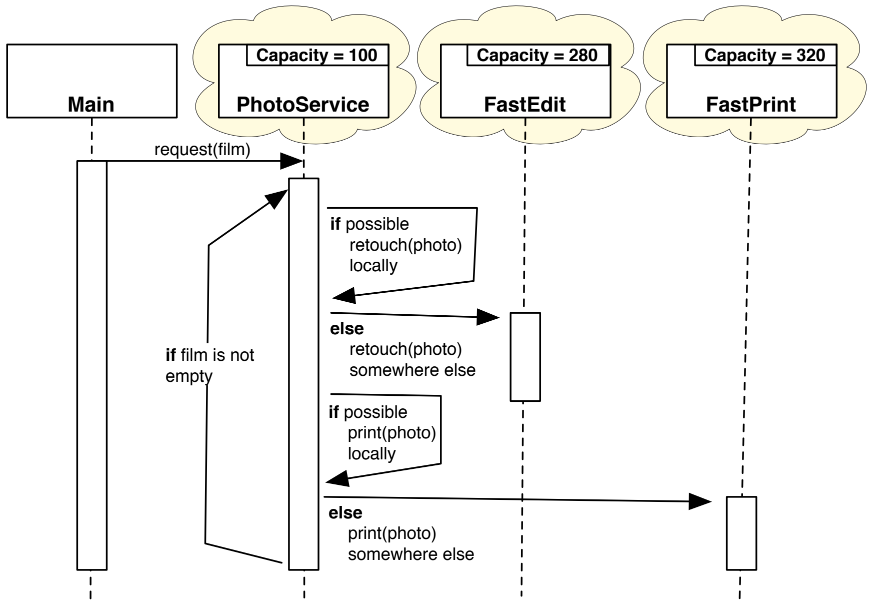
Main
Capacity = 100
PhotoService
Capacity = 280
FastEdit
Capacity = 320
FastPrint
request(film)
if possible retouch(photo) locally
else retouch(photo) somewhere else
if possible print(photo) locally
else print(photo) somewhere else
if film is not empty

FIGURE 1.4: A sequence diagram for a photo printing shop

to the capacity of the PhotoService object, represented by a size smaller than 500 and a capacity of at least 100, respectively. In this case, the retouch can be done locally, otherwise retouch is done by an auxiliary FastEdit object. A similar procedure applies to printing the retouched photos. Thus, photos of small sizes are retouched and printed locally, while photos with bigger sizes are sent to be retouched and printed externally. The implementations of the different methods are abstractly captured using **job** statements.

1.5 Proof System

The proof system for μABS is formalized as Hoare triples [5, 22] $\{\phi\}\ s\ \{\psi\}$ with a standard partial correctness semantics: if the execution of s starts in a state satisfying the precondition ϕ and the execution terminates, the result will be a state satisfying the postcondition ψ. In this paper, we are particularly interested in assertions about the *deadline* variables of method activations.

The reasoning rules for μABS are presented in Figure 1.6. Reasoning about sequential composition, conditional, and assignment is standard, and captured by the rules COMP, COND, and ASSIGN, respectively. Time passes when **job**(e)

```
type Photo = Rat; // size of the file

interface FastEdit {
    @requires p < 4000 && capacity>=250; @ensures reply == p + p/10; @within 2;
    Photo retouch(Photo p);}
class FastEditImp {
    @requires p < 4000 && capacity>=200; @ensures reply == p + p/10; @within 2;
    Photo retouch(Photo p) {job(200); return (p + p/10)}}

interface FastPrint {
    @requires p < 5000 && capacity>=300; @within 2;
    Unit print(Photo p);}
class FastPrintImp {
    @requires p < 5000 && capacity>=250; @within 2
    Unit print(Photo p) {job(250);return unit}}

interface PhotoService {
    @requires ∀ p:Photo, p ∈ film && p < 4000;
    @ensures reply == True; @within 4*length(film) + 10;
    Bool request(List<Photo> film);}
class PhotoServiceImp(FastEdit edit,FastPrint print) {
    @requires ∀ p:Photo, p ∈ film && p < 4000;
    @ensures reply == True; @within 4*length(film)+1;
    Bool request(List<Photo> film) {
         Photo p = 0;
         if (film != Nil){
                p = head(film);
                if (p < 500 && capacity>=100){ p = this.retouch(p);}
                else{p = edit.retouch(p);}
                if ( p < 600 && capacity>=100){this.print(p);}
                else{print.print(p);}
                this.request(tail(film));}
         else{ job(1);}
         return (deadline >= 0) }

    @requires p < 500 && capacity>=100; @ensures reply == p + p/20; @within 1;
    Photo retouch(Photo p) {job(100); return (p + p/20)}

    @requires p < 600 && capacity()>=100; @within 1;
    Unit print(Photo p) { job(100); return unit}}
```

FIGURE 1.5: A photo printing shop in μABS

is executed; **job**(e) has a duration e/cap on an object with capacity cap. The assertion in Rule Job ensures that this duration is included in the response time after executing **job**(e). The subsumption rule allows to strengthen the precondition and weaken the postcondition. For method definitions, the premise of Rule Method assumes that the execution of sr starts in a state where the **requires**-clause ϕ is satisfied and that the expected response time ($deadline$) is larger than expression e, where e is the specified response time guarantee from the **within**-clause. When the execution of sr terminates, the result will satisfy the **ensures**-clause ψ and the expected response time remains non-negative. For method invocations in Rule Call, the specification of the method is updated by substituting the formal parameters $\overline{fp}$ by the input expressions $\overline{e}$. The logical variables for the return value of the method (*reply*) and of the expected response time are renamed with fresh variables α and

$$\textsc{(Method)}\quad \frac{\{\phi \wedge \mathit{deadline} \geq e\}\ \mathit{sr}\ \{\psi \wedge \mathit{deadline} \geq 0\}}{\begin{array}{c}\textsf{@requires}\ \phi;\ \textsf{@ensures}\ \psi;\ \textsf{@within}\ e;\\ \mathsf{T}''\ m\ (\overline{\mathsf{T}}\ \overline{x})\ \{\overline{T'}\ \overline{x'}; \mathit{sr}\}\end{array}}$$

$$\textsc{(Return)}\quad \frac{\{\phi\}\ s; \mathit{reply} = e\ \{\psi\}}{\{\phi\}\ s; \textsf{return}\ e\ \{\psi\}}$$

$$\textsc{(Comp)}\quad \frac{\{\phi\}s_1\{\psi'\} \quad \{\psi'\}s_2\{\psi\}}{\{\phi\}s_1; s_2\{\psi\}}$$

$$\textsc{(Cond)}\quad \frac{\{\phi \wedge b\}s_1\{\psi\} \quad \{\phi \wedge \neg b\}s_2\{\psi\}}{\{\phi\}\ \textsf{if}\ b\ \{s_1\}\ \textsf{else}\ \{s_2\}\ \{\psi\}}$$

$$\textsc{(Subsumption)}\quad \frac{\{\phi'\}\ s\ \{\psi'\} \quad \phi' \Rightarrow \phi \quad \psi' \Rightarrow \psi}{\{\phi\}\ s\ \{\psi\}}$$

$$\textsc{(Assign)}\quad \{\phi[x \mapsto e]\}\ x = e\ \{\phi\}$$

$$\textsc{(Job)}\quad \{\phi[\mathit{deadline} \mapsto \mathit{deadline} - (e/\mathrm{cap})]\}\ \textsf{job}(e)\ \{\phi\}$$

$$\textsc{(New)}\quad \frac{\begin{array}{c}\mathit{fresh}(\alpha)\\ \phi' = \phi[x \mapsto \alpha]\\ T = \mathit{typeOf}(x)\\ \phi' \Rightarrow \mathit{implements}(C, T, e)\end{array}}{\{\phi'\}\ x = \textsf{new}\ C(\overline{e})\ \textsf{with}\ e\ \{\phi\}}$$

$$\textsc{(Call)}\quad \frac{\begin{array}{c}\mathit{fresh}(\alpha, \beta) \quad T = \mathit{typeOf}(e)\\ \phi' \Rightarrow \mathit{requires}(T, m)[\overline{\mathit{fp}} \mapsto \overline{e}]\\ \phi' = \phi[x \mapsto \alpha, \mathit{deadline} \mapsto \mathit{deadline} - \beta]\\ \phi_1 = \mathit{ensures}(T, m)[\overline{\mathit{fp}} \mapsto \overline{e}, \mathit{reply} \mapsto \alpha]\\ \phi_2 = \mathit{within}(T, m)[\overline{\mathit{fp}} \mapsto \overline{e}, \mathit{deadline} \mapsto \beta]\end{array}}{\{\phi' \wedge \phi_1 \wedge \phi_2\}\ x = e.m(\overline{e})\ \{\phi\}}$$

FIGURE 1.6: Proof system for μABS

β, respectively. To avoid name clashes between scopes, we assume renaming of of other variables as necessary. Object creation (in Rule New) is handled similarly to assignment. The precondition ensures that the newly created object of a class C with capacity e correctly implements interface T, where T is the type of x. (Note that the class instance may or may not implement an interface, depending on its capacity.) If a method has a return value, expression e in the return statement will be assigned to the logical variable *reply* in Rule Return, and can be handled by the standard assignment axiom in Rule Assign.

We show in Equation 1.3 the skeleton of the proof for the method `request` in Figure 1.5 by using the proof system presented in Figure 1.6. Let *sr* refers to the method body of `request` and s is *sr* without the return statement. In addition,

$$\begin{gathered}\psi = \mathit{reply} == \texttt{True}, \quad \psi_1 = \psi \wedge \mathit{deadline} \geq 0,\\ \phi = \forall p : \mathrm{Photo},\ p \in \mathit{film}\ \wedge\ p < 4000, \quad \text{and} \quad e = 4 * \texttt{length}(\mathit{film}) + 10\end{gathered} \tag{1.1}$$

We further assume that

$$\psi_2 = \mathit{reply} == \mathit{deadline} \geq 0 \wedge \mathit{deadline} \geq 0 \tag{1.2}$$

be the postcondition of the assignment $reply = deadline \geq 0$.

By Rule Method, the assertions ϕ and $deadline > e$ serve as the precondition of the whole method body sr, where ϕ and e are defined in the **requires**- and **within**-clauses in the definition of the method `request` in Figure 1.5. The postcondition of the method body consists of ψ, which is specified in **ensures**-clause as $reply == \texttt{True}$, and the expression $deadline \geq 0$. Rule Return converts the **return** statement into a statement where the expression $deadline \geq 0$ is assigned to the logical variable *reply*. Then, by the assignment axiom Assign, and with the postcondition ψ_2 assumed in Equation 1.2, the precondition ψ_3 is the postcondition with the logical variable *reply* substituted with the expression $deadline \geq 0$, and thus $\psi_3 = \texttt{True} \wedge deadline \geq 0$. By using Rule Subsumption, the postcondition ψ_2 is weakened to the given postcondition ψ_1. By Rule Comp, the assertion ψ_3 is also the postcondition of the statement s.

$$\cfrac{\cfrac{\cfrac{\cfrac{\vdots}{\{\phi \wedge deadline > e\}\ s\ \{\psi_3\}} \qquad \cfrac{\{\psi_3\}\ reply = deadline \geq 0\ \{\psi_2\} \quad \psi_2 \Rightarrow \psi_1}{\{\psi_3\}\ reply = deadline \geq 0\ \{\psi_1\}}}{\{\phi \wedge deadline > e\}\ s; reply = deadline \geq 0\ \{\psi_1\}}}{\{\phi \wedge deadline > e\}\ s; \textbf{return}(deadline \geq 0)\ \{\psi_1\}}}{\begin{array}{l}\textbf{@requires}\ \phi;\ \textbf{@ensures}\ \psi;\ \textbf{@within}\ e; \\ \qquad \textsf{Bool}\ \texttt{request}(\texttt{List}\langle\texttt{Photo}\rangle\ \texttt{film})\{sr\}\end{array}} \tag{1.3}$$

For brevity, the rest of the proof is omitted in the paper, which can be completed by repeatedly applying the corresponding rules from the above proof system.

1.6 Related Work

The work presented in this paper is related to the ABS modeling language and its extension to virtualized computing on the cloud in the Envisage project. The ABS [25] language and its extensions with time [9], deployment component and resource-awareness [27] provide a formal basis for modeling virtualized computing. ABS has been used in two larger case studies addressing resource management in the cloud by combining simulation techniques and cost analysis, but not by means of deductive verification techniques; a model of the Montage case study [13] is presented in [26] and compared to results from specialized simulation tools and a large ABS model of the Fredhopper Replication Server has been calibrated using COSTABS [3] (a cost analysis tool for ABS) and compared to measurements on the deployed system in [4, 12]. Related techniques for modeling deployment may be found in an extension of VDM++ for embedded real-time systems [45]. In this extension, static architectures are explicitly modeled using CPUs and buses. The approach uses fixed resources targeting the embedded domain. Whereas ABS has been designed to support compositional verification based on traces [14], neither ABS nor VDM++ supports deductive verification of non-functional properties today.

Assertional proof systems addressing timed properties, and in particular upper bounds on execution times of systems, have been developed, the earliest example perhaps being [41]. Another early example to reason about real-time is Nielson's extension of classical Hoare-style reasoning to verify timed properties of a given program's execution [36, 37]. Soundness and (relative) completeness for of the proof rules of a simple while-language are shown. Shaw [40] presents Hoare logic rules to reason about the passage of time, in particular to obtain upper and lower bounds on the execution times of sequential, but also of concurrent programs.

Hooman employs assertional reasoning and Hoare logic [23] to reason about concurrent programs, covering different communication and synchronization patterns, including shared-variable concurrency and message passing using asynchronous channels. The logic introduces a dense time domain (i.e., the non-negative reals, including ∞) and assumes conceptually, for the purpose of reasoning, a single, global clock. The language for which the proof system is developed, is a small calculus, focussing on time and concurrency, where a **delay**-statement can be used to let time pass. This is comparable to the **job**-expression in our paper, but directly associates a duration with the job. In contrast, we associate a cost with the job, and the duration depends on the execution capacity of the deployed object. Timed reasoning using Dijkstra's weakest-precondition formulation of Hoare logic can be found in [19]. Another classical assertional formalism, Lamport's *temporal logic of actions* TLA [1, 32], has likewise been extended with the ability to reason about time [31]. Similar to the presentation here, the logical systems are generally given by a set of derivation rules, given in a classical pre-/post-condition style. Thus, the approaches, in the style of Hoare-reasoning, are compositional in that timing information for composed programs, including procedure calls, is derived from that of more basic statements. While being structural in allowing syntax-directed reasoning, these formalisms do not explore a notion of timed interfaces as part of the programming calculus. Thus they do not support the notion of design-by-contract compositionality for non-functional properties that has been suggested in this paper.

Besides the theoretical development of proof systems for real-time properties, corresponding reasoning support has also been implemented within theorem provers and proof-assistants, for instance for PVS in [15] (using the duration calculus), and HOL [18]. An interesting approach for *compositional* reasoning about timed system is developed in [16]. As its logical foundation, the methodology uses TRIO [17], a general-purpose specification language based on first-order linear temporal logic. In addition, TRIO supports object-oriented structuring mechanisms such as classes and interfaces, inheritance, and encapsulation. To reason about open systems, i.e., to support modular or compositional reasoning, the methodology is based on a rely/guarantee formalization and corresponding proof rules are implemented within PVS. Similarly, a rely/guarantee approach for compositional verification in linear-time temporal logics is developed in [28, 44]. A further compositional approach for the verification of real-time systems is reported in [24], but without making use of a rely/guarantee framework.

Refinement-based frameworks are another successful design methodology for complex system, orthogonal to compositional approaches. Aiming at a correct-by-construction methodology, their formal underpinning often rests on various refinement calculi [6, 34, 35]. Refinement-based frameworks have also been developed for timed systems. In particular, Kaisa Sere and her co-authors [8] extended the well-known formal modeling, verification, and refinement framework Event-B [2] with a notion of time, resulting in a formal transformational design approach where the proof-obligations resulting from the timing part in the refinement steps are captured by timed automata and verified by the Uppaal tool [7].

The Java modeling language JML [10] is a well-known interface specification language for Java which was used as the basis for the interface specification of service contracts in our paper. Extensions of JML have been proposed to capture timed properties and to support component-based reasoning about temporal properties [29, 30]. These extensions have been used to modularly verify so-called performance correctness [42, 43]). For this purpose, JML's interface specification language is extended with a special **duration**-clause, to express timing constraints. The JML-based treatment of time is abstract insofar as it formalizes the temporal behavior of programs in terms of abstract "JVM cycles". Targeting specifically safety critical systems programmed in SCJ (Safety-critical Java), SafeJML [20] re-interprets the **duration**-clause to mean the worst-case execution time of methods concretely in terms of absolute time units. For a specific hardware implementation for the JVM for real-time applications, [39] presents a different WCET analysis [38] for Java. The approach does not use full-fledged logical reasoning or theorem proving, but is a static analysis based on integer linear programming and works at the byte-code level. We are not aware of work relating real-time proof systems to virtualized software, as addressed in this paper.

1.7 Concluding Remarks

Cloud computing provides an elastic but metered execution environment for virtualized services. Services pay for the resources they lease on the cloud, and new resources can be elastically added as required to offer the service to a varying number of end users at an appropriate service quality. In order to make use of the elasticity of the cloud, the services need to be *scalable*. A service which does not scale well may require a complete redesign of its business code. A *virtualized* service is able to adapt to the elasticity provided by the cloud. We believe that the deployment strategy of virtualized services and the assessment of their scalability should form an integral part of the service design phase, and not be assessed a posteriori after the development of the business code as it is done today. The design of virtualized services provides new challenges for software engineering and formal methods.

Virtualization empowers the designer by providing far-reaching control over the resource parameters of the execution environment. By incorporating a resource management strategy which fully exploits the elasticity of the cloud into the service, *resource-aware* virtualized services are able to balance the service contracts that they offer to their end users, to the metered cost of deploying the services. For resource-aware virtualized services, the integration of resource management policies in the design of the service at an early development stage seems even more important.

In this paper, we pursue a line of research addressing the formal verification of service contracts for virtualized services. We have considered a very simple setting with an interface language which specifies services, including their service contracts in the form of response time guarantees, and a simple object-oriented language for realizing these services. To support non-functional behavior, the language is based on a real-time semantics and associates deadlines with method calls. Virtualization is captured by the fact that objects are dynamically created with associated execution capacities. Thus, the time required to execute a method activation depends not only on the actual parameters to the method call, but also on the execution capacity of the called object. This execution capacity reflects the processing power of virtual machine instances, which are created from within the service itself. The objective of the proof system proposed in this paper is to apply deductive verification techniques to ensure that *all local deadlines are met during the execution of a virtualized service*. This proof system builds on previous work for real-time systems, and recasts the deductive verification of timing properties to a setting of virtualized programs. The extension of service interfaces with response-time guarantees, as proposed in this paper, allows a compositional design-by-contract approach to service contracts for virtualized systems.

Several challenges to the proposed approach are left for future work, in particular the extension to concurrency and asynchronous method calls, but also the incorporation of code which reflects the actual computations (replacing the job-statements of this paper). In this case, the abstraction to job-statements could be done by incorporating a worst-case cost analysis [3] into the proof system. Another interesting challenge, which remains to be investigated, is how to incorporate the global requirements which we find in many service-level agreements into a compositional proof system, such as the maximum number of end users.

Bibliography

[1] Martín Abadi and Leslie Lamport. An old-fashioned recipe for real-time. In Jaco W. de Bakker, Cees Huizing, and Willem-Paul de Roever, editors, *Real-Time: Theory in Practice (REX Workshop)*, volume 600 of *Lecture Notes in Computer Science*, pages 1–27. Springer, 1991.

[2] Jean-Raymond Abrial. *Modelling in Event-B.* Cambridge University Press, 2010.

[3] Elvira Albert, Puri Arenas, Samir Genaim, Miguel Gómez-Zamalloa, and Germán Puebla. COSTABS: a cost and termination analyzer for ABS. In Oleg Kiselyov and Simon Thompson, editors, *Proc. Workshop on Partial Evaluation and Program Manipulation (PEPM'12)*, pages 151–154. ACM, 2012.

[4] Elvira Albert, Frank S. de Boer, Reinar Hähnle, Einar Broch Johnsen, Rudolf Schlatte, S. Lizeth Tapia Tarifa, and Peter Y. H. Wong. Formal modeling of resource management for cloud architectures: An industrial case study. *J. of Service-Oriented Computing and Applications*, 2013. Springer Online First, DOI 10.1007/s11761-013-0148-0.

[5] Krzysztof R. Apt, Frank S. de Boer, and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Systems.* Texts and Monographs in Computer Science. Springer, 3rd edition, 2009.

[6] Ralph-Johan R. Back. *On the Correctness of Refinement in Program Development.* PhD thesis, Department of computer Science, University of Helsinki, 1978.

[7] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Petterson, and Wang Yi. UPPAAL — a tool-suite for the automatic verification of real-time systems. In R. Alur, T. Henzinger, and E. D. Sontag, editors, *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science.* Springer, 1996.

[8] Jesper Berthing, Pontus Boström, Kaisa Sere, Leonidas Tsiopoulos, and Jüri Vain. Refinement-based development of timed systems. In John Derrick, Stefania Gnesi, Diego Latella, and Helen Treharne, editors, *Integrated Formal Methods – 9th International Conference, IFM 2012, Pisa,*

Italy, June 18-21, 2012. Proceedings, volume 7321 of *Lecture Notes in Computer Science*, pages 69–83. Springer, 2012.

[9] Joakim Bjørk, Frank S. de Boer, Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. User-defined schedulers for real-time concurrent objects. *Innovations in Systems and Software Engineering*, 9(1):29–43, 2013.

[10] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005.

[11] Rajkumar Buyya, Chee S. Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Comp. Sys.*, 25(6):599–616, 2009.

[12] Frank S. de Boer, Reiner Hähnle, Einar Broch Johnsen, Rudolf Schlatte, and Peter Y. H. Wong. Formal modeling of resource management for cloud architectures: An industrial case study. In *Proc. European Conference on Service-Oriented and Cloud Computing (ESOCC)*, volume 7592 of *Lecture Notes in Computer Science*, pages 91–106. Springer, September 2012.

[13] Ewa Deelman, Gurmeet Singh, Miron Livny, G. Bruce Berriman, and John Good. The cost of doing science on the cloud: The Montage example. In *Proceedings of the Conference on High Performance Computing (SC'08)*, pages 1–12. IEEE/ACM, 2008.

[14] Crystal Chang Din, Johan Dovland, Einar Broch Johnsen, and Olaf Owe. Observable behavior of distributed systems: Component reasoning for concurrent objects. *Journal of Logic and Algebraic Programming*, 81(3):227–256, 2012.

[15] Simon Fowler and Andy Wellings. Formal analysis of a real-time kernel specification. In Bengt Jonsson and Joachim Parrow, editors, *Proceedings of FTRTFT'96*, volume 1135 of *Lecture Notes in Computer Science*, pages 440–458. Springer, 1996.

[16] Carlo A. Furia, Matteo Rossi, Dino Mandrioli, and Angelo Morzenti. Automated compositional proofs for real-time systems. *Theoretical Computer Science*, 376(3):164–184, 2007.

[17] Carlo Ghezzi, Dino Mandrioli, and Angelo Morzenti. TRIO, a logic language for executable specifications of realtime systems. *Journal of Systems and Software*, 12(2):255–307, 1990.

[18] Mike Gordon. A classical mind: Essays in honour of C.A.R. Hoare. In Anthony W. Roscoe, editor, *A Classical Mind: Essays in Honour of C.A.R. Hoare*, chapter A Mechanized Hoare Logic of State Transitions, pages 143–159. Prentice Hall, 1994.

[19] Volkmar H. Haase. Real-time behavior of programs. *IEEE Transactions on Software Engineering*, 7(5):594–501, September 1981.

[20] Ghaith Haddad, Faraz Hussain, and Gary T. Leavens. The design of SafeJML, a specification language for SCJ with support for WCET specifications. In *Proceedings of JTRES'10*, pages 155–163. ACM, 2010.

[21] Reinar Hähnle and Einar Broch Johnsen. Resource-aware applications for the cloud. *IEEE Computer*, May 2015. To appear.

[22] Charles A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[23] Jozef Hooman. Extending Hoare logic to real-time. *Formal Aspects of Computing*, 6(6A):801–826, 1994.

[24] Jozef Hooman. Compositional verification of real-time applications. In Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference (Compos '97)*, volume 1536 of *Lecture Notes in Computer Science*, pages 276–300. Springer, 1998.

[25] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In B. Aichernig, F. de Boer, and M. M. Bonsangue, editors, *Proc. 9th Intl. Symp. on Formal Methods for Components and Objects*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2011.

[26] Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. Modeling resource-aware virtualized applications for the cloud in Real-Time ABS. In *Proc. Formal Engineering Methods (ICFEM'12)*, volume 7635 of *Lecture Notes in Computer Science*, pages 71–86. Springer, November 2012.

[27] Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. Integrating deployment architectures and resource consumption in timed object-oriented models. *Journal of Logical and Algebraic Methods in Programming*, 2014. Available online.

[28] Bengt Jonsson and Yih-Kuen Tsay. Assumption/guarantee specifications in linear-time temporal logic. *Theoretical Computer Science*, 167(2):47–72, 1996.

[29] Joan Krone, William F. Ogden, and Murali Sitaraman. Modular verification of performance constraints. In *ACM OOPSLA Workshop on Specification and Verification of Component-Based Systems (SAVCBS)*, pages 60–67, 2001.

[30] Joan Krone, William F. Ogden, and Murali Sitaraman. Profiles: A compositional mechanism for performance specification. Technical Report RSRG-04-03, Department of Computer Science, Clemson University, Clemson, SC 29634-0974, June 2004.

[31] Leslie Lamport. Hybrid systems in TLA$^+$. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Workshop on Theory of Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 77–102. LNCS 736, Springer, 1993.

[32] Leslie Lamport. Introduction to TLA. Technical report, SRC Research Center, December 1994. Technical Note.

[33] Bertrand Meyer. Design by contract: The Eiffel method. In *TOOLS (26)*, page 446. IEEE Computer Society, 1998.

[34] Carrol C. Morgan. *Programming from Specifications*. Prentice Hall, 1990.

[35] Joseph M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, December 1987.

[36] Hanne Riis Nielson. *Hoare-Logic for Run-Time Analysis of Programs*. PhD thesis, Edinburgh University, 1984.

[37] Hanne Riis Nielson. A Hoare-like proof-system for run-time analysis of programs. *Science of Computer Programming*, 9, 1987.

[38] Peter Puschner and Alan Burns. A review of worst-case execution time analysis (editorial). *Real-Time Systems*, 18(2/3):115–128, 2000.

[39] Martin Schoeberl and Rasmus Pedersen. WCET analysis for a Java processor. In *Proceedings of the 4th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES'06)*, pages 202–211, 2006.

[40] Alan C. Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, 15(7):875–889, 1989.

[41] Mary Shaw. A formal system for specifying and verifying program performance. Technical Report CMU-CS-79-129, Carnegie Mellon University, June 1979.

[42] Murali Sitaraman. Compositional performance reasoning. In *Proceedings of the Fourth ICSE Workshop on Component-Based Software Engineering: Component-Certification and System Prediction*, may 2001.

[43] Murali Sitaraman, Greg Kulczycki, Joan Krone, William F. Ogden, and A. L. N. Reddy. Performance specification of software components. In *ACM Sigsoft Symposium on Software Reuse*, 2001.

[44] Yih-Kuen Tsay. Compositional verification in linear-time temporal logic. In Jerzy Tiuryn, editor, *Proceedings of FoSSaCS 2000*, volume 1784 of *Lecture Notes in Computer Science*, pages 344–358. Springer, 2000.

[45] Marcel Verhoef, Peter Gorm Larsen, and Jozef Hooman. Modeling and validating distributed embedded real-time systems with VDM++. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *Proceedings of the 14th International Symposium on Formal Methods (FM'06)*, volume 4085 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2006.

[46] Alan Wills. Capsules and types in Fresco: Program verification in Smalltalk. In Pierre America, editor, *Proc. European Conference on Object-Oriented Programming (ECOOP)*, volume 512 of *Lecture Notes in Computer Science*, pages 59–76. Springer, 1991.