Softw Syst Model (2010) 9:161–201
DOI 10.1007/s10270-009-0115-6

REGULAR PAPER

Formal specification of non-functional properties of component-based software systems

A semantic framework and some applications thereof

Steffen Zschaler

Received: 13 September 2007 / Revised: 3 November 2008 / Accepted: 22 January 2009 / Published online: 15 February 2009
© Springer-Verlag 2009

Abstract Component-based software engineering (CBSE) is viewed as an opportunity to deal with the increasing complexity of modern-day software. Along with CBSE comes the notion of component markets, where more or less generic pieces of software are traded, to be combined into applications by third-party application developers. For such a component market to work successfully, all relevant properties of components must be precisely and formally described. This is especially true for non-functional properties, such as performance, memory foot print, or security. While the specification of functional properties is well understood, non-functional properties are only beginning to become a research focus. This paper discusses semantic concepts for the specification of non-functional properties, taking into account the specific needs of a component market. Based on these semantic concepts, we present a new specification language QML/CS that can be used to model non-functional product properties of components and component-based software systems.

Keywords Non-functional properties · Formal specification · Component-based software engineering · QML/CS

Contents

1 Introduction . . . 161
2 Prerequisites: TLA$^+$ . . . 162
3 A semantic framework for specifying non-functional properties 164
3.1 Overview of the Framework . . . 164
3.2 Core framework concepts . . . 168
3.3 Component networks . . . 177
3.4 Specifying multiple interacting properties . . . 182
3.5 Summary . . . 185
4 Application of the framework . . . 185
4.1 A new specification language for non-functional properties 185
4.2 Specifying the Interface of analysis techniques . . . 191
4.3 Summary . . . 191
5 Related work . . . 192
5.1 Application structuring techniques . . . 192
5.2 Non-functional properties . . . 193
5.3 Related projects . . . 197
5.4 Summary . . . 198
6 Conclusions and outlook . . . 198

Communicated by Dr. Robyn Lutz.

S. Zschaler (✉)
Computing Department, Lancaster University, Lancaster, UK
e-mail: szschaler@acm.org

1 Introduction

Modern software systems are growing ever more complex. This complexity leads to increased time to market or to larger numbers of errors introduced into the software. It has, therefore, become necessary to counter the increase in complexity and provide development techniques that can help cope with complexity. Component-based software engineering (CBSE) [75] is viewed as one such technique. It helps deal with complexity by following a divide-and-conquer approach, modularising large software systems into smaller, reusable units—called (software) components. CBSE is supposed to be particularly effective in the context of so-called component markets, where components are developed by independent third-party developers and bought by application builders to be composed into complete applications.

If components are to be traded on component markets, they must be accompanied by a precise description of all of their relevant properties. Such a description must be expressed by component developers without knowledge of the context in

المنارة للاستشارات
www.manaraa.com

which their components will be used. On the other hand, it must be understandable by application builders and it must be possible for them to compose specifications of different components and reason about properties of the final system. For example, application builders need to know whether an application built from a certain set of components and deployed on a system with a certain amount of available resources will meet required performance goals, how much memory or network bandwidth will be consumed, or whether data quality will be up to required standards. Ideally, such reasoning would be supported by development tools. This means that we require formal means of specification.

A large amount of literature exists regarding the formal and precise specification of functional properties of applications and components. At the same time, for non-functional properties, work is only just beginning. However, the non-functional properties of a component or application are just as important as its functional ones. In addition to a formal specification of functional properties, we, therefore, require a formal specification of non-functional properties. Moreover, as also expressed in [24], any such language must be generic to be usable in the context of a component market. Many different users will use the language to express a very large number of properties. It is infeasible to require that they learn a new language for every property that they may find relevant. Some generic languages for the expression of non-functional properties exist (most notably, the component quality modelling language (CQML) [1]), but they lack formality and precise semantics. Furthermore, any such specification language must make provisions for the fact that, in a component market, every player (e.g., a component developer) possesses only partial knowledge of the properties of a complete system. Therefore, non-functional specifications of components must necessarily differ from non-functional specifications of systems.

Service-level agreements (SLAs) are used in many areas to formally or semi-formally describe the non-functional properties a certain service should provide (see [70] for an example that has some relations to the approach presented here). However, SLAs only specify the properties required of a service as a whole. The properties of individual components and how they compose to provide certain properties of a service or system are not in the scope of SLAs.

This paper presents a new formal semantic framework for the precise specification of non-functional properties of component-based applications. The formalism is based on extended temporal logic of actions (TLA$^+$) [44]. This semantic framework is then used to construct a formally founded new specification language—QML/CS. To the best of our knowledge, this is the first generic specification language for non-functional properties of component-based systems that has a formal foundation and semantics. Further, the paper presents an approach for formally specifying the interface of analysis techniques for non-functional properties. Analysis techniques are viewed as operations on system specifications and their interface is described using pre- and post-conditions. The semantic framework is a summary presentation of work more extensively discussed in [81]; presentation here aims to make this work available to a broader part of the research community. The specification language is a new contribution that has not been previously published.

The remainder of this paper is structured as follows: To ease understanding of the more formal parts of the paper, Sect. 2 gives a quick introduction to TLA$^+$. Next, Sect. 3 presents the semantic framework, starting with a high-level overview of the main concepts, followed by an in-depth discussion of these concepts and their formal embodiment in TLA$^+$. Section 4 presents applications of the semantic framework. This section introduces QML/CS, a new specification language for non-functional properties of component-based systems, and presents an approach for formally specifying the interface of analysis techniques for non-functional properties—for example in the context of Model-Driven Architecture (MDA) tool components [9]. Section 5 gives an extensive overview of related work and Sect. 6 concludes the paper. The article discusses various examples. The complete TLA$^+$ representation of the main example is not included in the article for space reasons, but can be obtained as a technical report [82] instead.

There are two ways to read this article: Readers not desiring a deep understanding of TLA$^+$ and other formal matters can read Sects. 3.1, 4.1.1, 5, and 6. The remaining sections cover the formal foundations.

2 Prerequisites: TLA$^+$

In this article, we use temporal logic as the formalism to describe our semantic framework. In particular, we use extended TLA$^+$ [44], a temporal logic introduced by Abadi and Lamport. It is important to note that this design decision slightly limits the expressiveness of our approach, because although temporal logic is capable of expressing a very broad range of properties, it cannot be easily used to express stochastic properties. However, the advantages of temporal logic—comparative ease of use, support of proof rules and model checking analysis techniques, compatibility to formal notions commonly used in specifying functional properties of components (e.g., state machines), and relatively large range of specifiable systems—in our opinion outweigh this limitation. In the following, we give a quick introduction to TLA$^+$ to ease understanding of the material in this article. For further details as well as the formal definition of the concepts used, please refer to the literature.

In TLA$^+$, systems are characterized by the set of behaviours they can perform. A behaviour is an infinite sequence

المنارة للاستشارات
www.manaraa.com

of states. For a given set of states Σ, the set of all behaviours over Σ is denoted by Σ^∞. A sequence of states is also called a trace. The set of all finite traces over Σ is denoted by Σ^*. We typically use σ and τ to refer to traces (or behaviours) and σ_i to refer to the ith state in σ. $\sigma|_n$ refers to a sub-trace of σ that only contains the first n states. In the other direction, we can use $\sigma \circ \tau$ to produce a new trace by concatenating traces σ and τ. A pair of consecutive states in a trace is called a step.

A state is defined by the values of so-called flexible variables. Two states where all such variables have the same values are considered equal. Σ is, thus, the cross-product of the domains of all flexible variables. A step where both states are equal in variables v is called a stuttering step in v. Two traces are called stuttering equivalent in v ($\simeq_v$) iff[1] they are equal once all stuttering steps in v have been removed.

Systems are described by TLA$^+$ formulas, which express constraints over flexible variables. The notation $\sigma \models F$ indicates that formula F holds for behaviour σ. There are two basic types of formulas:

State functions: A state function maps a state to a value. Of particular interest are *predicates;* that is, boolean-valued state functions.

Transition functions: A transition function maps a pair of states to a value. Of particular interest are *actions;* that is, boolean-valued transition functions.

For a predicate P, $\sigma \models P$ holds iff $P(\sigma_1) = \text{TRUE}$; that is, iff P holds for the first state in σ. For an action A, $\sigma \models A$ iff $A(\sigma_1, \sigma_2) = \text{TRUE}$; that is, iff A holds for the first pair of states in σ. In any transition function, we use v' to refer to the value of variable v in the second state.

Based on these basic types of formulas, we can construct temporal-logic specifications of systems. In particular, we will use the following constructions in this article (note that we do not associate specific meaning with characters A, B, etc.):

- Conjunction and disjunction as known from standard predicate logic. TLA$^+$ uses a special notation to save parentheses: Alignment of junctors can be used to group subexpressions. Thus:

$$
\begin{aligned}
(A \vee B) \wedge C \equiv \;& \wedge \vee A \\
& \quad\; \vee B \\
& \wedge C
\end{aligned}
$$

- UNCHANGED v is the same as saying $v' = v$; that is, it is the action asserting that v does not change.
- $\Box[A]_v$ where A is an action and v is a variable or a sequence of variables. This is the same as saying $\Box(A \vee \text{UNCHANGED } v)$ (Using the standard temporal-logic meaning of $\Box$, which is 'always'). In other words, it asserts that each step where v changes is an A step. Note that in TLA$^+$ it is illegal to use the operator $\Box$ on an action in any form other than this construction. $\Box$ can be used directly on predicates to express invariants, however.
- $A \stackrel{+}{\triangleright} B$ asserts that B holds *at least as long as* A. In other words, $\sigma \models A \stackrel{+}{\triangleright} B$ iff for any prefix $\sigma|_n$ for which A holds, B holds also. In fact, the formal definition of $\stackrel{+}{\triangleright}$ requires that B holds for at least one step longer than A, if A ever stops to hold. This is useful for modelling open systems, that provide a service B as long as the environment behaves in a certain way, described by A.
- $\exists v : A$ is a means of hiding variable v in A. It asserts that we can find a sequence of values for v so that A holds.
- **if** A **then** B **else** C evaluates to B if A evaluates to TRUE and to C, otherwise.
- Functions are expressed as follows:
  - $[S \to T]$ denotes the set of all functions from S to T.
  - $[x \in S \mapsto e]$ is the function f with domain S so that $f[x] = e$; that is, the value of f for argument x is the value of expression e (which may refer to x).
  - To update a function at one argument without changing the function value for any other argument, we can use the EXCEPT construction: $f' = [f \text{ EXCEPT } ![c] = e]$ which replaces the value of f at argument c by the value of expression e. e may contain the special character @ which means $f[c]$.
- We use $\wp(X)$ to denote the power set of a set X; that is, the set of all subsets of X.
- The formula E_{+v} asserts that, if the temporal formula E ever becomes false, then the state function v stops changing (see [4, Sect. 3.5.1] for further details).

Finally, we use $\triangleq$ to give names to formulas.

A system is described in TLA$^+$ by a specification, which typically represents a state machine. A state machine is a triple $S = (\Sigma, F, N)$ with Σ the set of states, $F \subseteq \Sigma$ a set of initial states, and $N \subseteq \Sigma \times \Sigma$ the next-state relation representing legal state transitions. A state machine can be represented by a TLA$^+$ formula of the form

$$
\begin{aligned}
S \triangleq \;& \wedge INIT \\
& \wedge \Box[NEXT]_{vars}
\end{aligned}
$$

where $INIT$ is a predicate describing the initial states F, $NEXT$ is an action describing N, and $vars$ is a set of all variables, describing Σ. A state machine specification S describes a property Π, such that $\forall \sigma : \sigma \models S \Leftrightarrow \sigma \in \Pi$. A property is a set of behaviours closed under stuttering equivalence. In other words: $\forall \sigma, \tau : ((\sigma \simeq_{vars} \tau) \wedge (\sigma \in \Pi)) \Rightarrow (\tau \in \Pi)$.

[1] if and only if.

www.manaraa.com

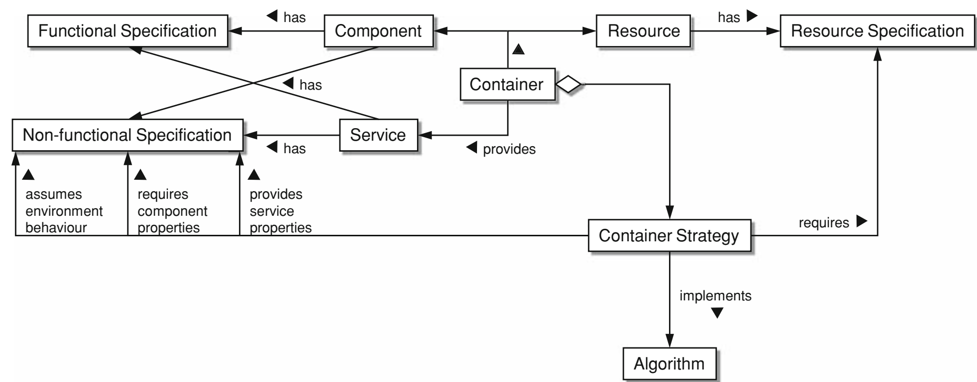

Fig. 1 System model expressed as a UML class diagram

TLA$^+$ specifications are organised into modules. A module can instantiate another module by using $MVar \triangleq$ **instance** $ModuleName$ **with** $VarReplacements$. This creates the name $MVar$ which is a reference to module $ModuleName$ with all variables replaced by variables of the using module in accordance with the replacement rules given in $VarReplacements$. We can then refer to formula F in module $ModuleName$ through $MVar!F$.

3 A semantic framework for specifying non-functional properties

This section presents our semantic framework and gives formal definitions for the individual parts of the framework. We begin with a high-level overview followed by an in-depth discussion in Sect. 3.2. Sections 3.3 and 3.4 extend our core concepts to networks of components and to specifications of more than one non-functional property.

3.1 Overview of the Framework

As pointed out by one of the anonymous reviewers, the most important challenge in defining the specification framework is to provide sufficient and appropriate layers of abstractions allowing independent specification of components and the services they eventually provide. To this end, we have defined five specification types to be discussed in the following subsections: service, component, resource, container, and measurement specifications. Figure 1 gives a graphical overview of these specifications and their relations.

There are two sides to developing component-based systems with defined non-functional properties:

1. Component developers must *implement* components in such a way that they have determinable non-functional properties.
2. Application designers and the runtime system must *use* these components so that the non-functional properties required from the application can be guaranteed.

Example 1 (Implementation versus usage) We can never make any guarantees about the memory consumption for a FIFO queue component which was implemented using a linked list without any limits on its maximum size. But even if the queue was implemented with a fixed-size array of length 64 kB, the runtime system can still use this implementation in such a way that it consumes 256 kB of memory: by creating four instances.

We are not primarily interested in how components must be implemented so that their non-functional properties become determinable. Instead, we assume components with determinable non-functional properties to be available. Based on this, we provide a semantic framework, which allows

– component developers to describe the non-functional properties of the components they have developed, and
– application designers to describe how these components are used to provide guaranteed non-functional properties of an application.

In the following, we give an overview of the various specifications defined by our approach. Each of these specifications is created by a different player in the component market and each of them represents a different important concept. All of them share the common terminology provided by a repository of formal measurement definitions.

3.1.1 Services

Users view a system in terms of the services it provides. They do not care about how these services are implemented, whether from monolithic or from component-structured software. A *service* is a causally closed part of the complete functionality provided by a system. As various authors [43,66] have pointed out, we can model services as partial specifications of a system. Multiple services can then be combined into a total specification of the system's functionality. Users associate non-functional properties with individual services—for example, they will talk about the frame rate provided by a video-player service independently of the response time of

Springer
www.manaraa.com

a cast query even if the two functions might share some components in their implementation. So, from the user's perspective, the non-functional properties of individual services should be described independently. Note that this does not imply that the non-functional properties of two services cannot interact—for example, because their respective implementations run on the same system and share the same resources. However, although users may be able to specify preferences on services, indicating which service should prevail in case of resource contention, they need to be able to describe their non-functional requirements independently for each service.

3.1.2 Components

Services are implemented by *Components*. A component can implement multiple services [43,66]. In addition, services can be implemented by networks of multiple cooperating components. In this case, the service's functionality is composed from the functionality of the individual components. In our video player example above, we might have a component for decoding video frames, one for retrieving film metadata (such as the film's cast) from a database, and a number of other components. For each of these components we might know the worst case execution time for decoding an individual frame or for retrieving a single record for one film, respectively.

3.1.3 Resources

The term *resource* is used in the literature essentially to refer to everything in the system which is required by an application in order to provide its services (e.g., [30,76]). More specifically, Goscinski defines a resource as:

> "[…] each reusable, relatively stable hardware or software component of a computer system that is useful to system users or their processes, and because of this […] is requested, used and released by processes during their activity." [30, Page 440*f.*]

The most important properties of a resource are that it can be allocated to, and used by, applications,[2] and that each resource has a maximum capacity. We do not consider resources with unlimited availability, because they do not have any effect on the non-functional properties of an application. We distinguish between the actual resource (e.g., CPU, memory) and the aspect it enables (e.g., execution of program code/computation, availability of space to store data).

[2] In our terminology, the term *application* encompasses everything required to provide a certain service to a user. That is, both components and containers are part of an application.

In our video player example, the most relevant resources are: CPU cycles required to perform all kinds of computations, network bandwidth for sending compressed or decoded video images to clients, and hard disk data transfer bandwidth for reading both encoded videos and film metadata from the database.

3.1.4 Container

Components require a runtime environment to execute them so that their services can be made available to the user. We call this runtime environment the *container*, inspired by the terminology used, for example, in the Java Enterprise computing framework. The container instantiates components, connects these instances to other instances according to the functional specification, and provides various middleware services to the components, including access to the underlying platform. In short, the container manages and uses the components such that it can provide the services clients require. Extending this notion to non-functional properties, we see that the container needs to use components and resources in such a way that it can guarantee the required non-functional properties of the services it provides.

Additionally, the system's environment also plays an important role. In particular, the container may need to make assumptions about the environment in order to provide its services. In this case, the container will only be able to provide a certain level of non-functional properties as long as its assumptions about the environment are still valid. Environment assumptions may include information on the interarrival times of requests (for time-based properties), assumptions about the abilities of system attackers (for security properties), usage profiles (for dependability properties), and so on.

In the video player example, the container is a piece of middleware that needs to instantiate all components and to allocate CPU cycles, network and hard-disk bandwidth to provide video-playing and cast-query services in the quality the user expects. Important environmental factors to take into account when doing these allocations are the number of films to be sent out concurrently and the rate with which users will issue cast-query requests to the system.

There exists a quite diverse set of non-functional properties that could be of interest for an application. *Container strategies* encapsulate *algorithms* which consider a specific set of non-functional properties and resources, and describe what needs to be done to support them. A container knows a set of such strategies that it applies when setting up an application. Each container strategy maps a set of non-functional properties of components and a set of constraints over resources (resource specifications in Fig. 1) to a set of non-functional properties of the corresponding service. Container strategies must be *functionality preserving;* that is, the

المنارة للاستشارات
www.manaraa.com

functionality of the service is derived only from the functionality of the components.

Example 2 (Response time) Given a component with one operation with a worst-case execution time, and information about the stream of incoming requests for this operation in the form of a jitter-constrained stream [32], the strategy described in [33] computes the required number of component instances, the amount of memory required to buffer incoming requests for the operation's service, the task set to be scheduled by the underlying CPU, and the worst-case response time for each operation invocation.

3.1.5 Measurements

We use the concept of a *measurement* to represent non-functional dimensions of systems. Non-functional specifications can then be expressed as constraints over measurements. A concept equivalent to our measurement is usually called *characteristic* in the literature (most notably [1,38]). We prefer the term measurement, because the concept is indeed based on the same concept from measurement theory (e.g., [27]) where a measurement is a mapping from physical or empirical objects to formal objects. The "physical or empirical objects" in our case are the systems under discussion—represented by state-based models of these systems—thus measurements can be represented as state functions. We use *context models* (state-based specifications of the parts of a system which are relevant for the definition of a measurement) to specify measurements independently of the concrete applications on which they are to be used.

We distinguish two kinds of measurements:

1. *Extrinsic measurements* describe non-functional dimensions which are applicable to a service and are relevant from a user perspective. They view the system as a whole and do not make distinctions to allow for other services, other components, or resource contention. In effect, extrinsic measurements can be used to describe users' non-functional requirements on a service. For example, the response time of the cast-query service is an extrinsic measurement.
2. *Intrinsic measurements* describe non-functional dimensions of component implementations. The value of an intrinsic measurement for a specific implementation depends principally on the way the implementation is realised. If two implementations differ in their values for an intrinsic measurement, they use different algorithms or implementation techniques to provide their functions. Definitions of intrinsic measurements account for the presence of other components, and for resource contention; that is, for the environment in which the component will be executed. In effect, intrinsic measurements can be used to describe the properties of an actually existing implementation independently of how this implementation is used. An example for an intrinsic measurement is execution time of an operation. This measurement has been used in Sect.3.1.2 to describe the properties of video-decoding components, for example. Note that the extrinsic measurement response time is not only a function of the execution times of the components involved. It is also affected by the CPU scheduling strategy of the operating system and the availability of other resources.

3.1.6 Non-functional properties

Non-functional properties are specified as constraints over measurements. Examples are properties like "The response time of service operation `queryCast` is always less than 50 ms", or "The execution time of component operation `queryCast` is always less than 30 ms." We distinguish four kinds of non-functional specifications (and corresponding non-functional properties).

Intrinsic specifications Component implementation properties are described using constraints over intrinsic measurements. These constraints describe relations between the various intrinsic measurements relevant for the component implementation. The most simple example is a statement like: "The execution time of component operation `queryCast` is less than 30 ms". More complex properties constrain the relation between multiple measurements. For example, for the video decoding component the decoding time per frame might depend on the decoding quality provided by the component. Intrinsic specifications describe the effect of the algorithms and implementation techniques used to create a component implementation.

Note, that component implementation specifications do not explicitly mention the resources required to provide the component's services. It is not useful to express resource demand of a component as an intrinsic property, because it depends largely on how the component is used. For example, CPU demand depends on both the intrinsic property execution time, and the number of requests per second the component has to serve. For this reason, component implementation specifications constrain intrinsic properties only, but some of these intrinsic properties (e.g., execution time) correspond to an aspect enabled by a certain resource (e.g., CPU). The relation between resource specifications and component implementation specifications is established by the container specification.[3]

[3] Execution time may vary depending on the underlying hardware. This issue is not supported by the work presented in this article, but is discussed further in Sect. 6 under the heading of *machine dependence*.

Springer

www.manaraa.com

Extrinsic specifications Service specifications constrain extrinsic measurements for a single service. These constraints express how users expect the system to behave or how a system as a whole behaves. The property "The response time of service operation `castQuery` is always less than 50 ms" from above is an example for an extrinsic specification.

Resource specifications Resources enable some non-functional aspect, if their capacity is sufficient to serve the specified load. This leads directly to resource specifications that consist of two parts: (a) an antecedent describing the capacity limits, and (b) a consequence describing the non-functional aspect enabled by the resource. For example, for a CPU with a scheduler based on rate-monotonic scheduling (RMS) [47] the capacity limit can be expressed by the following formula:

$$\sum_{i=1}^{n} \frac{t_i}{p_i} \leq n \cdot \left(\sqrt[n]{2} - 1\right) \quad (1)$$

where n is the number of tasks, and t_i and p_i refer to the worst case execution time and period of the ith task, resp. The non-functional aspect enabled by this resource is that the n tasks described by these parameters can be scheduled to execute jobs with a period p_i which are allowed to execute for at least t_i units of time between the begin and end of their respective period. This is in essence a constraint over execution time.

Container specifications The container uses resources and components to provide a service with certain non-functional properties under certain environment conditions. In order to reason about the extrinsic properties of a system based on the intrinsic properties of its components and the available resources we need to specify precisely how the container uses the components and resources. A container specification is written in rely–guarantee style [40] with the antecedent asserting that:

1. the available component implementations have the provided intrinsic properties. This pre-condition essentially enumerates the intrinsic properties the container takes into account.
2. The system's environment guarantees certain properties. Depending on the algorithms implemented by the container, the container will make different assumptions about the system environment. A typical example of the kind of guarantees given by the system environment is the distribution of request interarrival times. This can be used together with queuing-theory-based techniques to determine the optimal number of components and buffer size to achieve a required response time using components with a known execution time [33].
3. The available resources will enable the required non-functional aspects. What non-functional aspects are required depends on the intrinsic properties of the available components, the extrinsic property to be provided, the guarantees given by the system environment, and the algorithms implemented by the container. This antecedent is the central part of the container specification which describes the mapping from extrinsic and intrinsic non-functional properties of services and components to the lower-level concepts of resource specifications.

Provided these conditions hold, the container guarantees that it will deliver a certain service with specified extrinsic properties.

Example 3 (Container specification) Example 2 presents a container strategy for response-time guarantees. A corresponding container specification would indicate:

– that it requires components with a certain known worst-case execution time E;
– that it requires the environment to send requests at most every R milliseconds;
– that it requires sufficient memory to allocate corresponding buffers, and
– that it requires a CPU that can schedule tasks for the number of component instances the container determines as necessary.

The implication would state that the container provides a service with a maximum response time of R milliseconds.

3.1.7 Feasible systems

In the last section we have described four types of specifications. All these specifications are only useful if we can compose them to obtain a global view of the system which we can use for analysis. One useful analysis is to test whether the available resources are sufficient to provide the required extrinsic properties given the available components and the container specification. This is equivalent to proving that the composition of resource specifications, container specification, intrinsic specifications, and system environment guarantees implies the extrinsic specification. We define a *feasible system* to be a system for which this condition can be proved.

Example 4 (Feasible video-player system) Any combination of a video-decoding component, a container, and a set of resources is called a feasible system iff the component properties and the available resources are appropriate so that the container can provide a service with the properties actually required by a user. For example, if a user requires a frame

المنارة للاستشارات
www.manaraa.com

rate of 30 images/s but the available decoder component has a worst-case execution time of 50 ms and the container cannot compensate this by instantiating the component more than once, the overall video decoder is not a feasible system.

After this abstract overview of the major concepts of our semantic framework, we are now going to discuss the details of the framework itself.

3.2 Core framework concepts

This section discusses the fundamental concepts of the semantic framework. For the moment, we assume applications built from one component and with only one relevant non-functional property. The following sections will lift these restrictions.

3.2.1 Application and context models

Recall from Sect. 3.1.5 that we use the term *measurement* to denote non-functional dimensions. Measurements can then be constrained to describe non-functional properties of a system. A measurement can only be defined relative to some functional characterisation of the system to be constrained. This characterisation does not need to be complete, but it must contain the elements, structures, and behaviours on which the measurement relies. Because such a model gives the context of a measurement, we call it a *context model*. A context model explicitly expresses the assumptions a measurement makes about the functional environment to which it will be applied. Using context models, we can specify measurements independently of their usage in concrete applications.

Definition 1 (*Context model and application model*) A context model S_{Ctx} is given by a state machine $S_{Ctx} = (\Sigma_{Ctx}, F_{Ctx}, N_{Ctx})$, an application model S_{App} is analogously given by $S_{App} = (\Sigma_{App}, F_{App}, N_{App})$.

Context models are distinguished from application models by the fact that they have been constructed for the definition of a specific measurement m and model only those structures which are relevant to m.

Context models are more generic than application models. They do not define specific components, operations, or attributes, but the *concept* of a component, an operation, or an attribute. They abstract from the details of an application model, leaving only those concepts relevant for the definition of a specific measurement. By basing measurement definitions on a context model, such measurements can then be applied to a number of application models, as long as the context model can be viewed as an abstraction of each of these application models. There is, however, no formal distinction between context models and application models: they are both normal TLA$^+$ specifications.

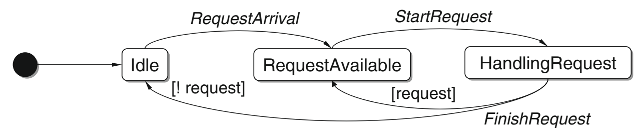

Fig. 2 An example of a context model defining the relevant steps in an operation call

Example 5 (Context model and application model) Figure 2 shows an example of a context model defining the relevant steps in an operation call. The notation is based on UML state diagrams.

It can be seen that this model only gives a very abstract view on an operation call. In particular, there is no reference to any concrete functionality. Instead, only the 'mechanics' of the service are expressed. The service is in state 'Idle' as long as no request for the execution of an operation has arrived. As soon as such a request arrives, the service changes to state 'RequestAvailable'. At some point, the service begins handling the request. As long as it is in the process of handling the request, the service is in state 'HandlingRequest'. The specification does not state how a request is handled. When the request has been handled completely, the service returns to state 'Idle' or 'RequestAvailable', depending on whether a new request has arrived in the meantime. This way of modelling a service operation is just one possible way of doing it. Depending on the measurement, other aspects of an operation call may be important and need to be modelled in a different manner.

In contrast, Fig. 3 shows an application model for a simple Counter component. The component offers two operations: `inc`, represented by the upper branch, which increments an internal value variable, and `getValue`, represented by the lower branch, which returns the current value of that variable. The component can be used to build applications that allow the counting of occurences of all kinds. Initially, the counter starts with a value of zero. Whenever an interesting event occurs, clients send an `inc` request to the component. Eventually, a `getValue` request is used to obtain the total value.

Notice that this model focuses on the actual functionality. It describes one application, where the context model of Fig. 2 describes a large number of applications. In fact, each of the two operations in the counter application can be viewed as a separate specialisation of the context model. For example, the states 'StartingIncrement' and 'FinishedIncrement' can be viewed as a refinement of 'HandlingRequest', 'ReceivedIncrement' as a refinement of 'RequestAvailable' and all other states as a refinement of 'Idle'.

3.2.2 Measurements

The formal specifications used to define measurements must not influence the behaviour of the systems to which they

المنارة للاستشارات
www.manaraa.com

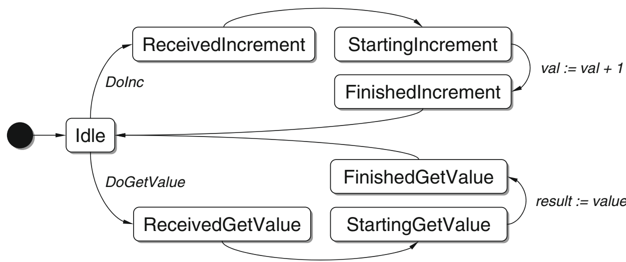

Fig. 3 An example of an application model describing a simple Counter component

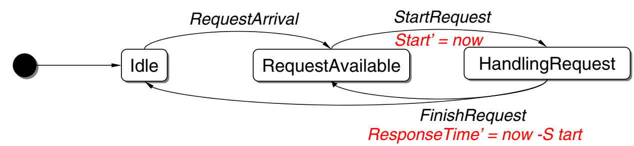

Fig. 4 State-machine representation of response time. Variable *Start* is used to record the time of the last invocation of the operation. Variable *ResponseTime* will hold the response time of the last completed invocation. Variable *now* is taken from Abadi/Lamport's technique for modelling time from [3]

are applied. Abadi/Lamport [2] formally define the concept of a *history variable* to represent state components that can be added to a specification without changing the externally observable behaviour of this specification. In [3] they apply history variables to define timers which can be used to express timing constraints in TLA$^+$specifications. We use history variables as the basis of our definition of a measurement:[4]

Definition 2 (*Measurement*) A measurement variable m is given by state functions f, g, and v, such that

$$Hist(m, f, g, v) \triangleq (m = f) \wedge \Box[m' = g \wedge v' \neq v]_{\langle m, v\rangle}$$

m does not occur free in either f or v, and m' does not occur free in g.

Measurements are defined relative to a context model S^m_{Ctx} with externally visible property Π^m_{Ctx}. For every measurement

$$\forall \sigma \in \Sigma^\infty : \sigma \in \Pi^m_{Ctx} \Rightarrow \sigma \models \exists m : S^m_{Ctx} \wedge Hist(m, f, g, v)$$

must hold. The complete measurement specification is then given by $S^m_{Ctx} \wedge Hist(m, f, g, v)$.

Intuitively, f specifies the initial value for the measurement, g defines the actual process of measuring—that is, how the measurement reacts to actions in the context model S^m_{Ctx}. v is a sequence of context-model variables that the measurement watches. Measurement updates can occur iff any one of the variables in v changes. Abadi/Lamport showed in [3] that conjoining $Hist(m, f, g, v)$ to S^m_{Ctx} does not affect the behaviour represented by S^m_{Ctx}.

Example 6 (Measurement) Figure 4 shows a state-machine representation of the definition of the response-time measurement. Its TLA$^+$ representation can be found in [82]. The measurement is defined by adding two history-determined variables to the context model of a service operation that can be seen in Fig. 2.

3.2.3 Applying measurements to concrete applications

Once we have specified non-functional dimensions using measurements, we need to apply these definitions to a concrete application. This requires that we map the concepts from the measurement's context model to the structures and behaviours present in the application model. In other words, we need to explain that, and how, the application model can be seen as an instance of the measurement's context model. This serves two purposes: (i) to check whether the measurement can indeed be applied in this specific case, and (ii) to bind parameters; that is to express specifically, to which part of our application the measurement should be applied. In the mapping we define, we want to combine context and application model such that we can reason about the combination as an instance of the context model without having to reason about—or know of—the specific application's functionality. We can do so by having the context model *observe* the behaviour of the application model; that is the two models "run" in parallel and the context model's behaviour is additionally constrained by the application model and a mapping between state variables. We will represent the mapping as a relation $\phi^{Ctx}_{App} \subseteq \Sigma_{App} \times \Sigma_{Ctx}$, so that we can represent the application by defining

$$\Pi^{App}_{Ctx} \triangleq \Pi_{App} \wedge \Pi_{Ctx} \wedge \Box\left(\langle v_{App}, v_{Ctx}\rangle \in \phi^{Ctx}_{App}\right) \qquad (2)$$

where v_{App} refers to all flexible variables of the application state and, correspondingly, v_{Ctx} refers to the flexible variables of the context model.[5] Because $\Pi^{App}_{Ctx} \Rightarrow \Pi_{Ctx}$, we can, thus, reason about our application as though it was an instance of our context model.

Example 7 (Model mapping) Figure 5 shows an example of a simple model mapping. It maps the context model from Fig. 2 onto the `inc` operation call defined in the application model from Fig. 3. We have introduced a few hierarchical states to group application-model states that are mapped to the same context-model state.

[4] Based on the definition of a history-determined variable given in [3].

[5] Remember that Π_{App} is the property induced by the application model and Π_{Ctx} is the property induced by the context model. Π^{App}_{Ctx} is then the property induced by combining the two models.

المنارة للاستشارات
www.manaraa.com

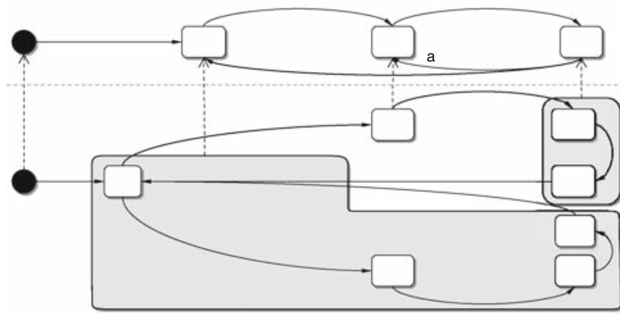

Fig. 5 An example model mapping. The *upper part* represents the context model from Fig. 2, the *lower part* is the application model from Fig. 3. The *dashed arrows* and *shaded areas* indicate which application-model states are mapped onto which context-model states

Three things should be noted about this example:

1. All application-model states that represent the invocation of the `getValue` operation have been mapped to the 'Idle' state in the context model. This means, that at the context-model level, we only observe calls to `inc`, because these cause state changes in the context model; all calls to `getValue` are ignored by this mapping. We will use this approach further down to associate measurements with individual operations, streams, or other parts of an application's behaviour.
2. In order for such a mapping to be possible, the application model's state machine must be fine-grained enough to allow the states required by the context model to be discerned. The context model then serves as an abstraction of the states of the application model. In particular, in our example, we had to model the mechanics of operation invocation in more detail than what would have been necessary purely for expressing the functionality of the Counter component. For example, instead of state 'ReceivedIncrement' for the Counter application it would have been sufficient to provide a transition directly between 'Idle' and 'StartingIncrement'. It is, however, conceivable that CASE tools providing a higher-level language for the expression of application models (e.g., a combination of UML and CQML$^+$such as reported in [64]) may use the context-model information to introduce the additional states necessary at the application-model level in a way that is transparent to the modeller.
3. The context-model transition labelled 'a' in Fig. 5 is effectively removed by this model mapping, as there is no corresponding transition in the application model. This is not a problem, however. We define model mappings so that the context model 'observes' the behaviour of the application model. Intuitively, the context model 'mimics' the steps taken by the application model. It is acceptable for the application model to reduce the space of behaviours represented by the context model. The following discussion will study this issue in more detail motivating why the same is not acceptable in the other direction.

Of course, not all mappings ϕ_{App}^{Ctx} are equally well suited. They represent how a certain measurement is being applied to a certain concrete application, and thus must maintain the semantics of the measurement and of the application. So, what are the conditions, ϕ_{App}^{Ctx} must fulfil? Most importantly, we want to retain the observational property of a measurement. We have used history-determined variables to define measurements so that adding a measurement to a specification will not change the set of behaviours described by this specification. To maintain this property also when mapping to concrete applications, we require that

$$\models \Pi_{App} \equiv \exists v_{Ctx} : \Pi_{Ctx}^{App} \tag{3}$$

where v_{Ctx} represents all flexible variables introduced by the context model. That is, for every behaviour satisfying the application model, we can find a sequence of values for the context model's variables so that both the model mapping and the context model are satisfied.

Of course, at some point, we want our non-functional specifications to restrict the set of possible behaviours of the system specified. We will discuss later in this section how constraining the possible *values* of measurements can constrain the set of possible behaviours of a system.

Equation (3) can be a little cumbersome to check. Therefore, we provide sufficient requirements on ϕ_{App}^{Ctx} directly:

(Φ1) Every initial state of the application model is mapped to at least one initial state of the context model:

$$\forall s_{App}^{f} \in F_{App} : \exists s_{Ctx}^{f} \in F_{Ctx} : \langle s_{App}^{f}, s_{Ctx}^{f} \rangle \in \phi_{App}^{Ctx}$$

Figure 6 shows a mapping respecting this condition. Initially, the application model starts out in `A0` and the context model starts in `C0`, both of which are initial states so that this is possible for both state machines. The application may eventually return to state `A0`, in which case the context model will go to state `C1`. This is not an initial state, but because the application has been running for some time already, this is not a problem. If `A0` had not been mapped to any initial state of the context model, conjoining the context model to the application model would effectively remove `A0` from the set of valid initial application states and so change the application's behaviour.

المنارة للاستشارات
www.manaraa.com

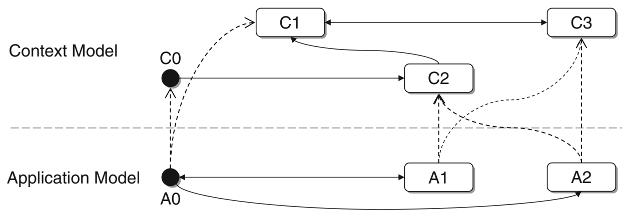

Fig. 6 Typical mappings for initial application states

Fig. 7 Example mapping violating the strict version of $\Phi 2$

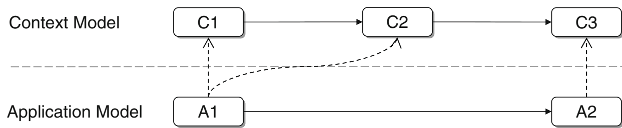

Fig. 8 Example mapping constrained by the less strict version of $\Phi 2$: the context model is allowed to observe an application model step in a sequence of steps

($\Phi 2$) Any legal state transition in the application model is mapped to at least one legal state transition or to a stuttering step in the context model:

$$\begin{aligned}\forall s^1_{App}, s^2_{App} \in \Sigma_{App} : \langle s^1_{App}, s^2_{App}\rangle \in N_{App} \\ \Rightarrow \forall s^1_{Ctx} \in \Sigma_{Ctx} \\ : \langle s^1_{App}, s^1_{Ctx}\rangle \in \phi^{Ctx}_{App} \\ \Rightarrow \exists s^2_{Ctx} \in \Sigma_{Ctx} \\ : \wedge \langle s^2_{App}, s^2_{Ctx}\rangle \in \phi^{Ctx}_{App} \\ \wedge \vee \langle s^1_{Ctx}, s^2_{Ctx}\rangle \in N_{Ctx} \\ \vee s^1_{Ctx} = s^2_{Ctx}\end{aligned}$$

Figure 7 shows a situation where this condition is violated. It can be seen that the set of behaviours allowed by the application model is restricted by the context model and the model mapping. The behaviour shown in the lower compartment, which had been allowed by the application model, is ruled out by the combination of application model and context model.

The rule above is overly strict, however. It also excludes the model mapping shown in Fig. 8, although the application model is at no time blocked by the context model. In many situations, it may be completely reasonable for the context model to perform multiple steps for one step of the application model. We, therefore, weaken the above condition as follows:

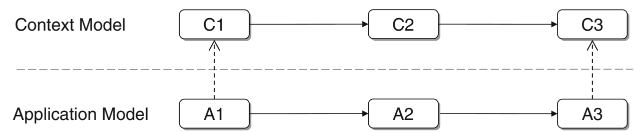

Fig. 9 An example mapping that violates rule $\Phi 3$

Any legal state transition in the application model is mapped to a sequence of legal state transitions or to stuttering steps in the context model:

$$\begin{aligned}&\forall s^1_{App}, s^2_{App} \in \Sigma_{App} : \langle s^1_{App}, s^2_{App}\rangle \in N_{App} \\ &\Rightarrow \forall s^1_{Ctx} \in \Sigma_{Ctx} : \langle s^1_{App}, s^1_{Ctx}\rangle \in \phi^{Ctx}_{App} \\ &\quad\Rightarrow \exists s^2_{Ctx} \in \Sigma_{Ctx} : \\ &\qquad \wedge \langle s^2_{App}, s^2_{Ctx}\rangle \in \phi^{Ctx}_{App} \\ &\qquad \wedge \vee \langle s^1_{Ctx}, s^2_{Ctx}\rangle \in N_{Ctx} \\ &\qquad\quad \vee \exists n \in \mathbb{N}, \widehat{s}^0_{Ctx}, \ldots, \widehat{s}^n_{Ctx} \in \Sigma_{Ctx} : \\ &\qquad\qquad \wedge \langle s^1_{Ctx}, \widehat{s}^0_{Ctx}\rangle \in N_{Ctx} \\ &\qquad\qquad \wedge \langle \widehat{s}^n_{Ctx}, s^2_{Ctx}\rangle \in N_{Ctx} \\ &\qquad\qquad \wedge \forall i = 0, \ldots, n-1 : \langle \widehat{s}^i_{Ctx}, \widehat{s}^{i+1}_{Ctx}\rangle \in N_{Ctx} \\ &\qquad\qquad \wedge \forall i = 0, \ldots, n : \langle s^1_{App}, \widehat{s}^i_{Ctx}\rangle \in \phi^{Ctx}_{App} \\ &\qquad\quad \vee s^1_{Ctx} = s^2_{Ctx}\end{aligned}$$

Note that we require that all states in the sequence of context-model states have been mapped to the first application-model state. This is exactly as we did in our example in Fig. 8, where `A1` is the first application-model state, and `A2` is the second one. If this condition is respected, the application model can always perform a series of stuttering steps while the context model moves along, preparing the application-model transition. Moreover, all situations where the context model needs to perform more steps than the application model can be represented in this way.

($\Phi 3$) The mapping is complete; that is, every state of the application model is mapped to at least one state in the context model:[6]

$$\forall s_{App} \in \Sigma_{App} : \exists s_{Ctx} \in \Sigma_{Ctx} : \langle s_{App}, s_{Ctx}\rangle \in \phi^{Ctx}_{App}$$

Figure 9 shows an example where this condition has been violated. It can be seen that the application model is blocked by this model mapping, because it is completely unclear what context-model state should be chosen to go with `A2`.

[6] This condition is a little over-exacting. It would be sufficient to demand that all *reachable* application states are mapped to some context-model state. However, the requirement that all states must be mapped can be fulfilled easily by providing a dummy mapping for unreachable states, and phrasing the condition so makes proofs a lot simpler.

المنارة للاستشارات
www.manaraa.com

Theorem 1 *(Conditions for model-mappings) Equations* $(\Phi 1)$–$(\Phi 3)$ *define sufficient conditions to produce a* ϕ_{App}^{Ctx} *fulfilling Eq.* (3).

The proof of this theorem would take up too much space in this article. It can be found in [81, Appendix C.2]. The theorem can be proved for both the strict and the weaker version of Eq. $(\Phi 2)$.

Equations $(\Phi 1)$–$(\Phi 3)$ look very similar to the conditions defined for refinement mappings in [2]. The major difference is that there is no externally visible state component which needs to remain identical. For very similar reasons, ϕ_{App}^{Ctx} only reminds one of the concept of observable simulation relation as defined, for example, in [74, p. 68]. A relation B between states is an *observable simulation relation* iff for each $(s_1, s_2) \in B$ and for each action A, $s_1 \overset{A}{\Rightarrow} s_3$ there exists a state s_4 so that $s_2 \overset{A}{\Rightarrow} s_4$ and $(s_3, s_4) \in B$.[7] Our conditions stipulate, rather, that there must exist some action C (different from A and which may also be the empty action ϵ) so that $s_2 \overset{C}{\Rightarrow} s_4$ and $(s_3, s_4) \in B$. This is a much weaker set of conditions.

The formalism most closely related to our notion of model mapping is the notion of a correct refinement for abstract state machines (ASMs) [11] as defined by Börger in [10]. Börger requires an equivalence relation $\equiv$ to be defined between states—this can be our relation ϕ_{App}^{Ctx}. An ASM M is then a *correct refinement* of another ASM M^* iff for every run R of M there exists a run R^* of M^* so that some subsequence of the states from R can be mapped to some subsequence of the states from R^* using $\equiv$. Thus, Börger allows both state machines to perform arbitrary steps between equivalent states. In contrast, we require every step of the application model to be matched by a step of the context model. Either step may be a stuttering step, but still, ϕ_{App}^{Ctx} must relate all states involved. Börger's approach works well for defining a notion of equivalence or refinement between specifications. However, we require a definition that allows usage of ϕ_{App}^{Ctx} as part of a specification, following Eq. (2). The realisation mappings between state machines proposed in [22] are also closely related to our model mappings. However, the conditions for well-defined realisation mappings are not discussed in [22] at all.

The lack of externally visible state or identically named actions, however, is also the biggest problem with Eqs. $(\Phi 1)$–$(\Phi 3)$, because the effects of the mapping are, thus, difficult to capture formally. This is similar to the situation with interface refinements as discussed in [44]. There, the point is made that such mappings actually add information to the specification. This is the reason why it is not possible to provide a complete definition of what a correct mapping looks like. All we can do, is to provide conditions that enable us to identify *incorrect* mappings. Equations $(\Phi 1)$–$(\Phi 3)$ are a good start on this way.

Example 8 (Bad model mapping) Consider the following mapping relation ξ_{App}^{Ctx}:

$$\begin{aligned} &\exists f_{Ctx} \in F_{Ctx} : \\ &\quad \wedge \forall s_{App} \in \Sigma_{App} : \langle s_{App}, f_{Ctx} \rangle \in \xi_{App}^{Ctx} \\ &\quad \wedge \forall s_{Ctx} \in \Sigma_{Ctx} : \vee\ f_{Ctx} = s_{Ctx} \\ &\qquad\qquad\qquad\qquad \vee \forall s_{App} \in \Sigma_{App} : \\ &\qquad\qquad\qquad\qquad\quad \langle s_{App}, s_{Ctx} \rangle \notin \xi_{App}^{Ctx} \end{aligned} \tag{4}$$

mapping every state of the application onto the same initial state of the context model. This is fully consistent with the conditions above, as well as with Eq. (3). However, the context model does not *observe* the application model at all, and any measurements defined based on this context model become meaningless. Yet, ξ_{App}^{Ctx} fulfils Eq. (3), which means that this condition is not sufficiently strict. We address this shortcoming in the next section.

3.2.4 A common basis for a domain—computational models

As we have seen, we were not able to give sufficient conditions to exclude any unsensible model mappings. The issue behind this is located not at the formal level, but rather it is a fundamental issue about the relation between the formalism and reality, or about the *intended semantics* of a measurement definition. By this we mean, the structure or function in reality that the original specifier intended to denote by the formal specification. Because any conditions we place on ϕ_{App}^{Ctx} can only express constraints on the formal level, they are not appropriate to capture intended semantics. Because in a component market, different people will develop different parts of the final and complete application specification, we need a common formal basis among these people, which allows them to communicate correctly. Note that this does not solve the initial problem of the intended semantics, but it makes it accessible to negotiation between the stakeholders outside the formal system. Once all parties agree on a common formal basis, everything else can be treated at the formal level. We call this common conceptual basis a *computational model:*

Definition 3 (*Computational model*) A computational model S_{CM} is given by a state machine

$$S_{CM} = (\Sigma_{CM}, F_{CM}, N_{CM})$$

A computational model is similar to a context model (*cf.* Definition 1) in that it is defined at the meta-level, but it is not constructed with view to a specific measurement definition. In contrast, a computational model captures the terms, structures and behaviours commonly agreed between different stakeholders in a *domain.* It captures the concepts relevant

[7] $E \overset{A}{\Rightarrow} E'$ stipulates that state E can be evolved to E' through a number of hidden actions and the action A.

المنارة للاستشارات
www.manaraa.com

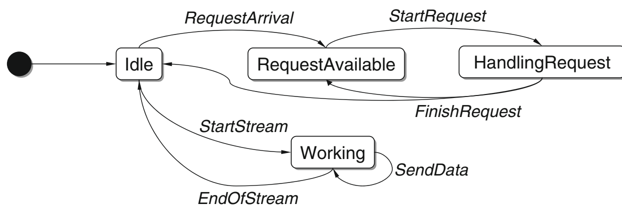

Fig. 10 An example for a computational model of services. In this domain, services can be either request-based (*upper part*) or stream-based (*lower part*)

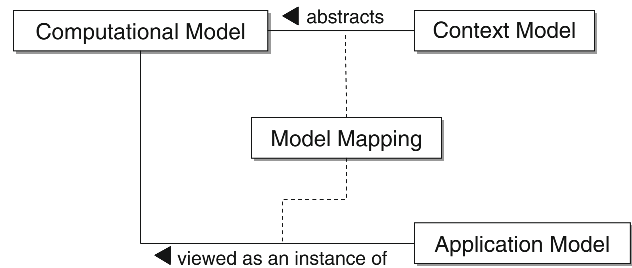

Fig. 11 Types of models defined and their interrelations

to this domain and thus limits and grounds the set of possible measurement definitions of this domain.

Example 9 (Computational model) Figure 10 shows an example computational model related to the example discussed so far. It contains all knowledge agreed on for a certain domain. It can be seen that in this domain services can be either request-based or stream-based services, with corresponding parts of the model reflecting each situation.

Once we have defined a computational model, all context models in a domain can be formally mapped onto this computational model. For example, the context model in Fig. 2 can be mapped to the computational model from Fig. 10 by mapping states `Idle` and `Working` to `Idle` in the context model. Thus, the computational model provides a commonly agreed basis for communication between different players in a domain.

It seems important to point out, that agreeing on a computational model is a consequential step in defining a domain, and that it *limits* the set of measurements that can be expressed in this domain. Because every measurement definition references variables already present in the context model, and because the context model will eventually be mapped onto the computational model, every variable to be referenced by a measurement specification must be representable in the computational model. A computational model which only considers operation calls will not allow measurements related to stream-based communication to be defined. This statement is in contrast to the—often implicit—notion in other works on measurement-based specification (most notably [1,29]) that these approaches can be used to specify any arbitrary measurement. While it remains true that they have the potential to do so, an important step toward making them useful is to agree on a computational model, and as soon as a computational model has been fixed, the expressiveness has been restricted. This shows up implicitly in Aagedal's thesis [1] when he presents characteristics without a definition of their semantics (i.e., without a `values`-clause).

Because context models and computational models are both defined at the meta-level, and because context models will eventually need to be mapped onto the computational model anyway, it can be argued that there is no need for context models at all. We have discussed in other publications [64] how multiple context models can be used to support the application designer's refinement decisions in incremental design. For the purposes of this article we will not consider this aspect and we will no further discuss the distinction between computational model and context models. For the remainder of this article we will use these terms interchangeably.

We have introduced now a number of different types of models, namely context, application, and computational model. To summarise and clarify their relationships, Fig. 11 provides an overview of these models and how they are connected. Computational models are abstractions from arbitrary applications, focussing only on those properties relevant for the definition of a number of measurements important in a certain domain. Context models are abstractions from computational models in that they contain only those concepts and properties relevant for the definition of a specific measurement. Model mappings, as defined in Sect. 3.2.3 are used to describe the relations between these models.

3.2.5 Non-functional properties

It is common in temporal logics to define *properties* as sets of behaviours—that is, infinite sequences of states. A property characterises a subset of the set of all behaviours. Recall that above we defined a measurement by using history-determined variables so that the set of valid behaviours was not affected by the addition of a measurement to the specification. Therefore, a measurement alone is *not* a non-functional property. A *non-functional property* is derived from a measurement, or a set of measurements, by specifying constraints over the measurement values.

Definition 4 (*Non-functional property*) Given a set $M = \{m_1, m_2, \ldots\}$ of measurements, a non-functional property over this set $\Pi_{Nf}(M)$ is given by any formula constraining the values of the measurements in M.

In this section, we focus on cases where M only contains one measurement m and discuss the fundamental definitions

www.manaraa.com

of our approach. Extensions to specifications with multiple measurements will be discussed in Sect. 3.4.

It should be noted that the above definition, together with the definition of a measurement, means that our approach cannot be used for properties that are not "measurable" as functions of the state of the product. For example, properties such as learnability or maintainability cannot be expressed as functions of the state of a running application. Rather, they depend on issues of user-interface design or code structure, among others. However, any other product property that can be so measured is covered by our approach.

As mentioned above, we distinguish four different types of non-functional properties: intrinsic, extrinsic, resource, and container specifications. In the following, we will discuss each of these four types in more detail.

As components are essentially subsystems, we would expect non-functional component specifications to be very similar to non-functional system specifications. However, there is one important difference between the two, which is related to the contextual knowledge of their producers and specifiers. While application designers will usually know about the specific system they use, in particular about the available resources and container strategies, component developers do not have such knowledge. Even worse, CBSE stipulates for components to be "[…] subject to composition by third parties" [75]. This implies that it is undesirable for component developers to make assumptions about the context of use of their components. Therefore, non-functional component specifications can only talk about properties intrinsic to the component (and thus independent of the component's context of use). Independence of the context of use can be achieved by modelling explicitly in the measurement definition the parts of the context which can affect the measurement value.

Definition 5 (*Intrinsic versus extrinsic specifications*) We distinguish two kinds of specifications, and—correspondingly—two kinds of non-functional measurements:

1. *Intrinsic specifications* and *intrinsic measurements:* Intrinsic properties apply to components and can be specified by component developers without further knowledge of the context of use of the components being specified. They are specified in the form of constraints over the relative values of intrinsic measurements, only. Intrinsic measurements are measurements whose value can be determined exclusively from the implementation of a component without consideration of context of use. The definition of an intrinsic measurement will typically include hooks describing explicitly the assumptions made about the context of use. We use S_{Cmp} to denote an intrinsic specification of non-functional properties of a component.

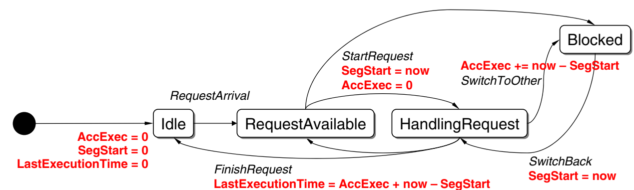

Fig. 12 State-machine representation of the execution-time measurement definition. Everything except the variable assignments is a representation of the context model of a component operation. Note the new state 'Blocked' modelling influences of the context of use

2. *Extrinsic specifications* and *extrinsic measurements:* Extrinsic properties apply to services and systems and can be derived by application designers using knowledge of the context of use. They can also be specified by system users to express their requirements on a system or service. Extrinsic specifications are based on extrinsic measurements, which assume the context of use to be known and, therefore, provide no hooks to explicitly describe assumptions about the context of use.

Example 10 (Extrinsic measurement) Response time as defined in Example 6 is an extrinsic measurement, as can be seen from the fact that it is based on the context model of a service operation. In contrast, Fig. 12 shows the definition of the intrinsic measurement execution time. It is based on the context model for a component operation call, including the additional state 'Blocked' to represent times when component execution is stopped by the environment to execute other components, container functionality, or because the component must wait for availability of some resource. The TLA$^+$ representation of the component context model and the execution time measurement can be found in [82].

3.2.6 Specifying system resources

Many applications require system resources, such as CPU, memory, hard disks, databases, and so on, to provide their functionality. The availability of these resources is an important factor determining the extrinsic non-functional properties of the system. Therefore, in order to determine the extrinsic properties of a system, we need to understand—and model—the resources available to it.

A resource specification consists of three layers, very similarly to intrinsic or extrinsic specifications:

1. The *resource-service layer:* This layer models the service provided by the resource. For example, a CPU provides execution slots to various tasks. This layer is very similar to the context models used with intrinsic or extrinsic properties in that it defines the terminology to be used in the other layers.

Springer
المنارة للاستشارات
www.manaraa.com

2. The *resource-measurement layer:* This layer uses history variables to describe the non-functional aspects of the resource. For our CPU example this would be the periods, worst-case execution times, and actual execution times per period for each task. This layer is very similar to the definition of measurements for intrinsic or extrinsic specifications.
3. The *resource-property layer:* This layer defines constraints over the history variables defined in the resource-measurement layer. For example, for an RMS-scheduled CPU, such a constraint would express that all tasks will meet their deadlines provided the schedulability criterion [Eq. (1)] is satisfied.

Definition 6 (*Resource specification*) A *resource specification* is a formula of the form

$$CapacityBoundsSpecification \stackrel{+}{\rhd} ResourceServiceSpecification$$

where

$CapacityBoundsSpecification$ is a predicate that is true if the current resource demand for the resource is below the capacity limit for the resource, and

$ResourceServiceSpecification$ specifies the service delivered by the resource.

Such a specification resides at the resource-property layer. The other two layers provide the terminology to be used in the final resource specification. They define the abstract resource, while the resource-property layer defines a concrete resource. So, both $CapacityBoundsSpecification$ and $ResourceServiceSpecification$ make use of concepts defined at the resource-measurement and the resource-service layer.

Example 11 (RMS-scheduled CPU) To continue with the CPU-example from above, $CapacityBoundsSpecification$ would use the measurements defined at the resource-measurement layer to express the RMS schedulability criterion, and $ResourceServiceSpecification$ would use those measurements to express the fact that all tasks scheduled will meet their respective deadlines.

Figure 13 shows the TLA$^+$ specification of a RMS-scheduled CPU. Notice that this is the resource-property layer only. The other layers are encoded within the specification `TimedCPUScheduler` imported on Line 37. The actual resource-property specification can then be found on Line 54. The `Schedulable` property defined on *lines 44–47* expresses the RMS schedulability criterion as also given on Page 6.

```
 1  ------------------------ MODULE RMSScheduler ------------------------
    A CPU Scheduler using RMS.
 5  EXTENDS Reals

    Parameters:

    TaskCount – the number of tasks to be scheduled on the CPU.
    Periods – the periods to be scheduled for these tasks. This is an array with one entry per task.
    Wcets   – the worst case execution times of the tasks to be scheduled. This is an array with one
              entry per task.
16  CONSTANT TaskCount
17  ASSUME (TaskCount ∈ Nat) ∧ (TaskCount > 0)

19  CONSTANT Periods
20  ASSUME Periods ∈ [{1 .. TaskCount} → Real]

22  CONSTANT Wcets
23  ASSUME Wcets ∈ [{1 .. TaskCount} → Real]

    Variables:

    MinExecTime – records for each task the minimum amount of execution time it has been allocated
                  over all periods so far.
    AssignedTo – holds the number of the task currently assigned the resource
    now        – the current time.
33  VARIABLE MinExecTime
34  VARIABLE AssignedTo
35  VARIABLE now

37  TimedCPUSched ≜ INSTANCE TimedCPUScheduler

39  ...
40  ----------------------------------------------------------------------
    Schedulable is TRUE if the given task load can be scheduled using RMS.
44  Schedulable ≜ LET usage ≜ [k ∈ {1 .. TaskCount}
45                             ↦ (Wcets[k]/Periods[k])]
46                IN
47                  Sum(usage) ≤ (TaskCount ∗ (sqrt(TaskCount, 2) − 1))

    The actual specification: A TimedCPUScheduler which will meet all deadlines provided the RMS
    schedulability is met by the tasks to be scheduled.
53  RMSScheduler ≜ ∧ TimedCPUSched!TimedCPUScheduler
54                 ∧ □Schedulable -+-▷ □TimedCPUSched!ExecutionTimesOk

56  ======================================================================
```

Fig. 13 Formal specification of an RMS-scheduled CPU

3.2.7 Binding together resources and components: containers and container strategies

To understand the extrinsic properties of a system, we need formal specifications of how available resources and components are used together inside the system. We use the concept of a container, and more specifically of a container strategy (a specific algorithm within a container), to represent this knowledge. We, therefore, need another type of formal specification: *container-strategy specifications.* Each such specification mentions the input of a container strategy; that is, the intrinsic properties and resources it uses. Furthermore, the specification describes the extrinsic properties the container strategy provides based on this input.

Definition 7 (*Container strategy specification*) A container is given by the specification of its container strategy. Each container strategy is specified by a *container-strategy specification* S_{CS}. Container strategy specifications are formulas of the form

$$\begin{aligned} &\wedge IntrinsicProperties \\ &\wedge ResourceRequirements \\ &\wedge EnvironmentConditions \\ &\wedge ComponentFunctionalityHook \\ &\stackrel{+}{\rhd} ExtrinsicSpecification \wedge \\ &\quad ServiceFunctionalityHook \end{aligned}$$

www.manaraa.com

where

IntrinsicProperties is a conjunction of constraints over intrinsic measurements expressing the requirements of the container strategy on the available components.

ResourceRequirements is a conjunction of constraints over resource measurements expressing the requirements of the container strategy regarding the available resources.

EnvironmentConditions represent the assumptions the resulting system makes about environment behaviour. For example, this may include assumptions about the frequency of incoming requests, or the accuracy of data entered.

ExtrinsicSpecification is a conjunction of constraints over extrinsic measurements, expressing the behaviour of the resulting system.

ComponentFunctionalityHook and *ServiceFunctionalityHook* represent the fact that each container strategy preserves the functionality provided by the components. $ComponentFunctionalityHook \triangleq CompFun \wedge CompMap \wedge (CompFun \Rightarrow ServFun)$ where $CompFun$ is a predicate parameter for the functionality provided by the component, $CompMap$ is a predicate parameter for the mapping between application model and context model, and $CompFun \Rightarrow ServFun$ is the actual statement of functionality preservation. $ServiceFunctionalityHook \triangleq ServFun \wedge ServMap$ with analogous meanings of the conjuncts.

This definition is limited to containers with only one container strategy. We will discuss containers with more than one strategy in Sect. 3.4.

Example 12 (Container strategy) Figure 14 shows an example specification of a container strategy. This strategy takes one component, whose execution time is known, and schedules a CPU task such that it can guarantee a certain maximum response time. Line 60 shows the actual container strategy specification. *Lines 14–37* show the specification of the container's expectations on the environment, and *lines 40–58* show the definition of what the container is to provide.

```
 1 ---------------- MODULE SimpleContainer ----------------
   A container specification for a very simple container. This container manages just one component
   instance and tries to achieve a certain response time with it.

   Below, we have cut out large parts of the specification. These used to import various additional
   specifications required, such as the specification of execution time and response time, and the
   specification of a CPU, where execution times and deadlines of tasks can be scheduled.
12 ...

14 ContainerPreCond ≜
15   ∧ ExecutionTime ≤ ResponseTime
16   ∧ The CPU must be able to schedule exactly one task with a period equal to the requested
       response time and a wcet equal to the specified execution time of the available component.
21     ∧ CPUCanSchedule(1, [n ∈ {1} ↦ ResponseTime],
22                         [n ∈ {1} ↦ ExecutionTime])
23   ∧ A component with the required execution time is available.
26     ∧ ComponentMaxExecTime(ExecutionTime)
27     ∧ CompFun
28     ∧ CompModelMapping
29   ∧ The component functionality implements the service functionality.
32     CompFun ⇒ ServFun
33   ∧ Requests arrive with a constant period, the length of which is somehow related to the
       period length requested from the CPU.
37     ∧ MinInterrequestTime(ResponseTime)

40 ContainerPostCond ≜
41   ∧ The promised response time can be guaranteed
44     ∧ ServiceResponseTime(ResponseTime)
45     ∧ ServFun
46     ∧ ServModelMapping
47   ∧ The container will allocate exactly one task for the component.
50     □ ∧ TaskCount = 1
51       ∧ Periods = [n ∈ {1} ↦ ResponseTime]
52       ∧ Wcets = [n ∈ {1} ↦ ExecutionTime]
53   ∧ State that the container will hand requests directly to the component, without buffering
       them in any way. If the container provides buffering, this would need to go here
58     □(CmpUnhandledRequest = EnvUnhandledRequest)

60 Container ≜ ContainerPreCond -+▷ ContainerPostCond
61 ---------------------------------------------------------
```

Fig. 14 Simple container strategy specification

3.2.8 System specifications and feasible systems

So far, we have discussed individual specifications for the individual elements of a system. Each of these pecifications can be written independently of all of the other specifications. Finally, we need to provide a specification of a complete system, built from various components, resources and a container.

Definition 8 (*System specification*) A *system specification* is given by the conjunction of intrinsic specifications of the available components, resource specifications of available resources, and a container strategy applied to the components and resources in the system:

$$S_{System} \triangleq \wedge \bigwedge_i S_{Cmp}{}^i \\ \wedge \bigwedge_i S_R{}^i \\ \wedge S_{CS}(S_{Cmp}{}^i, S_R{}^i)$$

where $S_{Cmp}{}^i$ and $S_R{}^i$ refer to the specifications of the ith component and resource in the system, respectively, and $S_{CS}(S_{Cmp}{}^i, S_R{}^i)$ expresses the application of a container strategy to these components and resources.

An example of a TLA$^+$ system specification based on the execution-time and response-time measurements we have discussed above can be found in [82].

We have now discussed the different types of specifications that comprise a complete system specification. It becomes important then to analyse whether "supply meets demand"; that is, whether the extrinsic properties provided by the system satisfy the requirements of the users.

Definition 9 (*Feasible system*) Given a system specification S_{System}, and a requirements specification $S_{Rqmts} \triangleq Environment \stackrel{+}{\rightharpoonup}\!\!\triangleright ExtrinsicProperty$ we call the system specified by S_{System} *feasible* with respect to S_{Rqmts}, denoted by $IsFeasible\left(S_{System}, S_{Rqmts}\right)$, iff the system specification is an implementation of the requirements specification:

$$IsFeasible\left(S_{System}, S_{Rqmts}\right) \triangleq S_{System} \Rightarrow S_{Rqmts}$$

www.manaraa.com

Here, $ExtrinsicProperty$ is an extrinsic specification as per Definition 5 and $Environment$ can be an arbitrary TLA$^+$-specification representing admissible behaviour of the system's environment.

Of course, we would like to be able to prove feasibility of a system specification with respect to certain requirements. Fortunately, Abadi/Lamport [4] have shown the *Composition Principle* which allows us to perform implementation proofs based on conjunctions of individual specifications, provided some conditions hold. We can limit ourselves to safety properties, because we are mainly interested in checking that a system's non-functional properties will hold *whenever it does something useful,* not necessarily in checking *that* it does something useful at all. Checking that the system does something useful at all is mostly concerned with the system's functional properties, and we will consider this a separate task to be performed prior to checking any non-functional properties. Therefore, we can use Abadi/Lamport's Composition Principle in a simplified form, ignoring liveness.[8] The original composition principle developed by Abadi/Lamport does support liveness under certain conditions, but at the cost of additional complexity of proofs. (In general, we might require liveness in the definition of our non-functional properties, however we have not yet encountered such a property.)

We first give the general composition theorem, then relate it to our specifications:

Theorem 2 (Composition principle) (*simplified from* [4, Theorem 3])

If, for $i = 1, \ldots, n$,

1. E, M, E_i, M_i *are safety properties,*
2. $\models E \wedge \bigwedge_{j=1}^{n} M_j \Rightarrow E_i$
3. (a) $\models E_{+v} \wedge \bigwedge_{j=1}^{n} M_j \Rightarrow M$ *where* v *a tuple of variables including all the free variables of* M
   (b) $\models E \wedge \bigwedge_{j=1}^{n} M_j \Rightarrow M$

then $\models \bigwedge_{j=1}^{n}(E_j \overset{+}{\rhd} M_j) \Rightarrow (E \overset{+}{\rhd} M)$.

Using this composition principle we can prove that the composition of component, resource and container strategy specifications (all of which are of the form $E_j \overset{+}{\rhd} M_j$[9]) implement the system requirements specification (which again is of the form $E \overset{+}{\rhd} M$). That is, the theorem provides the proof obligations for feasibility proofs. An example for a feasibility proof can be found in [81, Appendix C.3.3].

[8] Intuitively, safety properties specify that 'nothing bad happens', while liveness properties state that 'something good eventually happens'. A more formal definition can be found for example in [52].

[9] For example, in a resource specification as defined in Definition 6, E_j is the $CapacityBoundsSpecification$ and M_j is the $ResourceServiceSpecification$.

Notice, that the above definition essentially requires a certain amount of overlap between the individual specifications: The pre-conditions E_i of each specification must be fulfilled by some other specification M_i, or a combination of a number of such specifications. Applied to our framework we find such overlap particularly between container strategy specifications and the other specifications: Containers provide specifications of their expectations about components and resources—these are the E_i. Component and Resource specifications are the corresponding M_i. In our example TLA$^+$-specification, we have simply reused the same specifications in both places. This is the simplest form of overlap, but not a realistic one. If the specifications are written by different people, they will be different. All we require is that we can prove an implementation relationship between the specifications.

The discussions so far have covered the core concepts, but have been restricted to systems of one component and to only one intrinsic non-functional property. Real component-based systems consist of more than one component. Also, more than one intrinsic property are relevant for the construction and evaluation of such systems. Therefore, in the next two sections, we will extend these concepts to cover networks of components and more than one property.

3.3 Component networks

A component network provides its functionality through interactions between the constituting components. The structure of such a component network, also called its architecture, is typically modelled using an architecture description language (ADL) [51] (or an ADL-like language, such as some parts of the UML [36]). ADLs provide three basic concepts to model system architecture:

1. *Components* for modelling the actual loci of computation,
2. *Connectors* for modelling the interactions between components, and
3. *Configurations* for modelling component networks.

In a specification of non-functional properties of a component-based system, we need to provide information about non-functional properties for each of these. Therefore, there are three major subsections in this section:

1. *Component specification (Sect. 3.3.3):* The non-functional properties exhibited by a component C_0 depend on non-functional properties exhibited by other components C_i used by C_0. An important question is how these dependencies should be specified.
2. *Component interconnection (Sect. 3.3.2):* We need to specify the non-functional properties of connectors, as

المنارة للاستشارات
www.manaraa.com

they can have a major effect on non-functional properties of a component-based system. For example, data security provisions differ massively between systems where all components are co-located on the same machine (or even in the same process), and systems where all information exchange happens through a (possibly open and insecure) network.

3. *Configuration Specification (Sect. 3.3.1):* Container strategies must be extended to cope with cooperating components. Here, we need to balance expressive power in the container strategy specification against simplicity of the approach. An important sub-issue is the question of combining the resource demands for the individual components to describe the overall resource demand of the system.

3.3.1 Container strategies for component networks

There are essentially two ways in which we can extend the concept of a container strategy from above to support inter-operating networks of components:

1. *Global strategies:* A global strategy describes the way the container manages the complete network of components making up a service. In addition to the intrinsic and extrinsic properties for which it is applicable, a global container strategy also specifies certain architectural constraints describing the kinds of component networks for which it can be used.
2. *Local strategies:* A local strategy describes the way the container manages a single component. The complete container behaviour is then composed from individual container strategies, which have been selected for each component in the component network.

These two approaches represent the spectrum of available approaches. It is, of course, possible to mix them by having some of the components be managed as a group by one strategy and having other components be managed by other strategies. Figure 15 summarizes these three options, which we inspect in more detail in the following.

Global container strategies Global container strategies extend the container strategies from above so that they can handle *many* components and transform their interaction into one service. Hence, they allow the specification of global optimisations.

Example 13 (Global container strategies) In a radar tracking system the container may balance the amount of time spent in the actual sensor component against the amount of time spent in the ensuing analysis component, thus trading overall response time against precision of results. However, it can

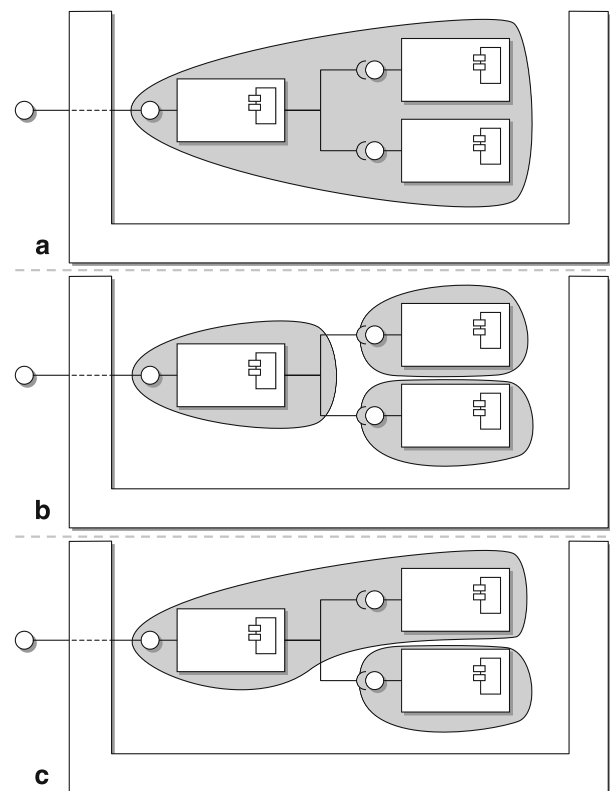

Fig. 15 Three approaches to specifying container strategies for component networks: **a** one global strategy for all components, **b** one local strategy for each component, and **c** mixed approach. The container is shown as a trough, each *shaded area* indicates the components managed by a separate strategy

only do so, if it is aware of the interaction of these two components and their relative contribution to the extrinsic properties of the system. As another example, knowing all components cooperating to provide the service, the container can decide where to place buffers and how to dimension these buffers, thus balancing response time against jitter. This example is discussed in further detail in [33].

The specific strategies typically depend on the architecture of the component network. For example, the contribution of the execution time of an individual component towards the response time of the complete system depends on how the components are interconnected, and how often they invoke each other. Because we want to specify container strategies independently of the concrete components they are going to manage, we need to make assumptions about the system's architecture and make them explicit in the specification of the container strategy. Thus, a *global container strategy* is the result of conjoining a container specification and a set of architectural constraints specifying the assumptions the container strategy makes about the system's architecture.

المنارة للاستشارات
www.manaraa.com

The specification of architectural constraints must fulfil at least the following requirements:

1. It needs to express the essential properties the architecture must possess to be managed by the container strategy.
2. It must be generic, meaning that it must not fit only one specific architecture, but as many concrete architectures as possible.
3. It must allow a clear identification of individual components, so that the components can be referenced in the actual container strategy specification.

Of course, the first two constraints must be balanced against each other depending on the specific container strategies. Some container strategies (e.g., the radar tracking example above) perform very application-specific optimisations; for these the architectural constraints need to pin the application rather specifically. Other strategies (e.g., the strategy trading response time for jitter) are only concerned with a certain set of non-functional properties; here the architectural constraints should be as generic as possible.

Local container strategies The container strategies we defined above took intrinsic properties of the component and available resources, and transformed them into a service complete with its extrinsic properties. The container strategy essentially worked as a wrapper around the component and the resources used by it, and also as an adapter producing a service from a component. This situation changes when we consider component networks: The extrinsic non-functional properties provided through a component A may now depend upon extrinsic properties of other components (namely the components used by A). Hence, assuming the container strategy has only knowledge of component A (the component it is local to), it cannot produce a complete service specification. Additionally, the system's resources may be shared between components in the same system. Again, the container strategy has no knowledge of the other components with which it needs to share resources. Thus, the application of a local container strategy to a component can only result in a conditional service specification, which still depends on the properties of other components and on global system resource sharing and availability. We call such a component wrapped by a local container strategy an *encapsulated component:*

Definition 10 (*Encapsulated component*) An *encapsulated component* is the result of composing (i.e., conjoining the specifications of) a component and a local container strategy. Its external view can be represented by a formula following the schema:

$$\begin{aligned} ECS \triangleq\ & \wedge\ ResourceRequirements \\ & \wedge\ RequiredQualitySpecification \\ & \wedge\ EnvironmentAssumption \\ & \stackrel{+}{\rhd} ExtrinsicProperties \end{aligned}$$

where
ResourceRequirements specifies the resource demand of the encapsulated component.
RequiredQualitySpecification specifies what non-functional properties the encapsulated component requires from the components it uses. These normally specify constraints over extrinsic properties, because encapsulated components are intended to be composed with other encapsulated components.
EnvironmentAssumption specifies assumptions the encapsulated component makes about its usage context.
ExtrinsicProperties specifies the extrinsic properties the encapsulated component provides.

$RequiredQualitySpecification$ and $EnvironmentAssumption$ are very similar in their intention. The difference is that $RequiredQualitySpecification$ specifies what the encapsulated component requires from components it uses, whereas $EnvironmentAssumption$ refers to how the encapsulated component itself is being used. For example, in the case of response time, $RequiredQualitySpecification$ would specify constraints on the response times of used components, whereas $EnvironmentAssumption$ could specify a constraint on the maximum invocation frequency.

We can use feasibility proofs quite similar to those introduced above to show that a system composed from a component C and a local container strategy LCS indeed implements the more abstract encapsulated component specification ECS from Definition 10. In a complete system specification, we can thus replace every component C_i by its corresponding encapsulated component ECS_i, conjoining them to a specification of the available system resources and component interconnections. We can then attempt to prove feasibility for the complete system.

Resource sharing requires some more preparations in our specifications. The container strategies discussed above assume that they are the only agent in the system requiring resources. Therefore, they use a pre-condition which essentially states "I need a resource which can handle *exactly* this demand." In contrast, in a component network, the local container strategy may need to share resources with other encapsulated components, but there is no way to know, when writing the container strategy specification, which components these are. Indeed this may be different from system to system in which the container strategy is to be employed. Therefore, we need a precondition which states

المنارة للاستشارات
www.manaraa.com

intuitively: "There needs to be a resource that handles my demand and that can still handle its complete load." Then, in the system specification, we can specify how the various resource requirements are combined and handed to the various available resources.

For different resources, combining resource demands means different things. For a simple resource such as main memory, resource demands are combined by adding their values (i.e., the kilobytes of memory required). For more complex resources, no single figure can appropriately express the combination of multiple resource demands. For example, in the case of a CPU, a set of tasks can often be boiled down to a single figure called *utilisation* (expressing the average percentage of time the CPU is occupied), but this figure is not always useful for determining schedulability of this set of tasks. Instead, sometimes we require more detailed knowledge of the tasks involved. In our specifications, we want to keep all knowledge of how resource demands are combined confined to resource specifications. Outside of resource specifications we only need to know how to express an individual resource demand.

Definition 11 (*Shared resource*) A shared resource R is a resource (*cf.* Definition 6) that defines a set D_R of potential resource demands, and accepts elements from $\wp(D_R)$ as descriptions of the total load.

Example 14 (Shared resources) The following list presents some resource demand models for a few typical shared resources:

CPU Individual resource demands are called tasks. Different scheduling models define different task models, for example:

- most standard scheduling models characterise a task by its ID, a worst-case execution time, and a period
- approaches based on imprecise scheduling [17] characterise a task by its ID, a distribution function for the execution time of a mandatory part, distribution functions for the execution times of one or more optional parts, a period and a quality figure indicating what percentage of the optional parts must be executed before the period end.

Memory Memory has the most simple resource demand model: Resource demand is simply expressed by a number of kilobytes required.

Network For network connections the simplest way to express resource demand is by giving a required bandwidth (be it constant or average rate) together with a source and a target address. More elaborate network management schemes require more information about resource demand—for example, a characterisation of the traffic to be transported.

With such a shared resource, combining resource demands becomes the union over sets. Testing whether a certain resource demand is contained in a certain set becomes testing for set inclusion. All knowledge of how resource demands are actually combined is hidden behind this interface. This approach is similar to that described in [53], where a central Quality-of-Service (QoS) Manager provides a resource-independent interface for specifying application resource demands based on XML. However, while that approach is an implementation solution for a real-time operating system, we provide an approach that allows for independent *specification* of resource demands and available resources.

For each encapsulated component we can now define two resource demand parameters per resource used:

1. the *global resource load* d_{global}—an input parameter of the encapsulated component, set in the system specification—and
2. the *local resource demand* of the encapsulated component d_{local}—an output parameter of the encapsulated component.

We use d_{local} to describe the resource demand of the encapsulated component, but use d_{global} to check resource availability. To make the underlying assumption explicit, we conjoin $d_{local} \subseteq d_{global}$ to the pre-condition of the encapsulated component. In the system specification we collect all the d^i_{local} and combine them using $d_{global} = \bigcup_i d^i_{local}$.

Comparison and summary We have discussed two polar approaches to extending container strategies to support interacting networks of components. *Global* container strategies manage *all* components forming an application. On the other hand, *local* container strategies manage *exactly one* component and, by wrapping it, transform it into an encapsulated component that provides a certain service under the condition that it receives certain resources and that other components it uses also provide certain extrinsic properties.

Both approaches have their advantages and disadvantages: For local container strategies the required extension is comparatively simple: all that is needed is to export the environment expectations (including resource demand and requirements on other components) and a mechanism specifying how resource demands are distributed on the actually available system resources. However, every container strategy can only influence the environment for one component in the network, and it can do so based essentially only on the properties of this component, and the components this component uses *directly*. On the other hand, global container strategies can query the properties of all components in the system, and, therefore, allow for global optimisations of the extrinsic properties of the complete system. Unfortunately, global container strategies are much more complex,

المنارة للاستشارات
www.manaraa.com

because they require a separate specification of architectural constraints describing the types of architectures the strategy supports, and they must balance the demands and properties of more than one component.

It is, therefore, sensible to combine the two approaches when constructing a system. To this end, we extend the definition of an encapsulated component as follows:

Definition 12 (*Composite encapsulated component*) A *composite encapsulated component* is an encapsulated component, which is internally composed of more than one component (encapsulated or simple). Its external view can be represented as for any other encapsulated component. Resource demand and requirements on other components may be derived from any of the internal components.

In Fig. 15 c, the upper shaded area represents a composite encapsulated component consisting of two subcomponents. The lower shaded area represents an atomic encapsulated component. Note that this definition encloses both polar approaches: Global container strategies turn the complete system into a single composite encapsulated component, while local container strategies use only atomic encapsulated components.

The concept of composite encapsulated components is very similar to the notion of hierarchical or nested components as used by many ADLs. However, for an encapsulated component, the behaviour of the complete component is always derived from the behaviours of the internal component *by a container strategy.* This introduces an additional layer not considered for hierarchical components in ADLs.

3.3.2 Component interconnection

Component connection can have a profound effect on the non-functional properties of a component-based system. We, therefore, need to attach specifications of the non-functional properties of the components' environment to the connectors that are part of the functional specification of a component-based application.

Example 15 For example, a connection over a network incurs a much greater performance penalty than a direct connection inside the same address space (for which the performance penalty will be close to zero in the ideal case). Figure 16 shows a measurement definition for the delay incurred by a connector transporting data packets between two components.

As another example, consider a measurement determining the security of some data in terms of who may become aware of the contents of this data.[10] Here, the measurement value is primarily influenced by the connection between the

[10] see [49] for a discussion of how to model such properties in TLA$^+$

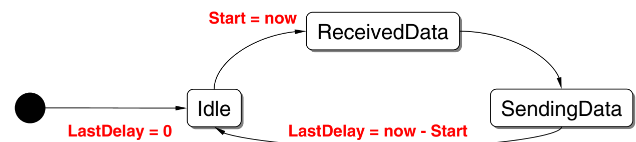

Fig. 16 A simple connector context model including the definition of a `delay` measurement

components: is it via an open network, via inter-process communication (IPC), or directly inside the same address space on the same machine?

Formally, connectors are not much different from components. We can define context models and measurements for connectors as for components and services, and use model mappings to constrain the behaviour of specific connectors. However, connectors are not managed by the container, and, therefore, do not distinguish between intrinsic and extrinsic properties.

3.3.3 Component specification

In component networks, the properties exhibited by an individual (encapsulated) component typically depend on properties of the components used by this component. The specification of these component dependencies can follow one of two polar approaches:

1. Extension of intrinsic specifications
2. Extension of container strategies

We discuss these polar approaches in more detail below:

Extension of intrinsic specifications We can extend intrinsic specifications to include a description of how the intrinsic properties of a component depend on properties exhibited by the components used by this component.

Example 16 (Extending intrinsic specifications) Consider a specification that intuitively asserts that the execution time of a `getData` operation in an object-relational-mapper (OR mapper) component is at most 100 ms plus the amount of time it takes the used database component to process a certain type of SQL query. It is important to realise that the amount of time used by the database is expressed as a constraint over database response time, i.e., an *extrinsic* property. This is so because the OR mapper always communicates with the database component through the container (even if both components are managed by the same global container strategy. Therefore, it views the database component as an encapsulated component. We do not care, in the specification of the OR mapper, how much of the database time is due to database execution time, and how much is due to some management work of the container.

المنارة للاستشارات
www.manaraa.com

Extending intrinsic specifications in this manner has the advantage of simplicity. We do not need new concepts or specification constructs. However, there is also a disadvantage: As we have seen in the example, such an extension causes intrinsic specifications to become a mixture of constraints over intrinsic and extrinsic measurements.

Extension of container strategies We can extend container strategies with a specification that uses the intrinsic specification, the functional specification of the component network, and the properties of other components when determining extrinsic properties of an encapsulated component. In many cases, the relation between properties of the components used and properties of the using component are quite independent of the computation performed by the component itself.

Example 17 In Example 16 above, the formula for the total time spent in the operation is always

$$TotalWaitTime \triangleq ExecutionTime + \Sigma_{co \in CalledOps} ResponseTime(co)$$

irrespective of whether the component under consideration is an OR-mapper or a hotel-management component.

In this example, the formula is specific to the set of measurements; the actual properties are only parameters to the formula. It is, therefore, sensible to move the specification of this relation from the intrinsic specification of the component into the container strategy specification dealing with this combination of properties. When we do this, the functional specification must provide explicit information on relevant properties of component interaction—namely the number of invocations of other operations in our example. Parametrised contracts—discussed by Reussner in his dissertation [60]—are an interesting approach for this, but sometimes we may be able to derive the required information directly from the functional specification.

Two aspects underlie the distinction between these two approaches:

1. *Separation of intrinsic and extrinsic properties*: The first approach of extending the intrinsic specification of a component forced us to mix constraints over intrinsic and extrinsic measurements in the component specification. Ideally, we should separate these concerns as much as possible, for two reasons: First, it mixes two levels of specification, making the specifications harder to understand. Secondly, mingling concerns keeps architectural information, such as the number of invocations made to a used component implicit.
2. *Generality of the dependency*: For some properties—for example, for execution time, as discussed above—the effect of component-interaction on the property is completely independent of the specific components. For other properties, however, the dependency of a component's property on the properties of used components is specific to this component. An example for such a property is accuracy: The error of the result of some computation obviously depends on the error of the input values, but the precise dependency hinges on the specific computation being performed. For such properties we cannot move the specification of the component dependency into the container strategy, but have to keep it in the intrinsic component specification.

It can be seen that of the two approaches we discussed each favours one of these aspects over the other: The extension of intrinsic specifications is useful for properties whose dependencies are very component-specific, but mixes intrinsic and extrinsic properties and keeps architectural information implicit. On the other hand, the extension for container strategies is good at separating intrinsic and extrinsic specifications and at making architectural information explicit, but fails to support properties with component-specific dependencies. Therefore, in general we should strive to keep property dependencies in the container strategy specifications, but for properties which have component-specific dependencies, we need to resort to using extended intrinsic specifications.

3.4 Specifying multiple interacting properties

In this section we extend our discussions to specifications of multiple, possibly interacting, non-functional properties. Three aspects of this problem need to be discussed:

1. *Support for multiple intrinsic non-functional properties in component specifications*: We need a possibility to specify more than one non-functional property of the same component. In addition, we need to be able to describe interactions between these properties—such as the increase in execution time caused by a certain increase in data quality.
2. *Support for multiple extrinsic non-functional properties in a service specification*: Similarly, we need a possibility to specify a combination of non-functional properties for a service provided by an application.
3. *Support for multiple non-functional properties in container specifications*: Ideally, we want to be able to combine individual container strategies for individual non-functional properties to form a container specification managing all of these properties.

www.manaraa.com

We will see that the first two points are very similar, and can be solved straightforwardly using the specification techniques we have already introduced. The main focus of this section will, therefore, be a discussion of problems and some solution ideas for combining individual container strategies to specify the behaviour of a container when faced with a set of intrinsic properties to be transformed into a set of extrinsic properties. The most prominent issue in this area is the problem of *feature interaction*—that is in our case the effect the application of one container strategy may have on the results of another one. This issue forms a research area of its own and leads us out of the scope of our work, so that we will only give some initial comments and possible research directions.

3.4.1 Relations between measurements

When a component exhibits multiple non-functional properties, there often exists a relationship between them. In a complete specification of the component, such relations must be formally expressed. For example, a component may offer different levels of accuracy of the result, but at the cost of increased execution time. Thus, the component must specify (in addition to specifying the individual properties themselves) how these properties are interrelated for this component. As another example, the execution time of an operation depends on whether parameters and result value must be de- and encrypted before and after the actual execution of the operation. Thus, if a component offers a non-functional property describing whether de-/ encryption happens, and an execution time specification, it also needs to specify how the de-/ encryption affects execution time.

In the two examples above, we have not indicated one specific component. However, the precise (quantitative) relationship between intrinsic properties typically depends on the specific component. Of course, examples like the ones above are commonplace, and we always talk about them without mentioning specific components. All such discussions reflect the general trend only, however, and as soon as we want to be more concrete, it seems that we always need to talk about specific components. To the best of our knowledge, no intrinsic properties exist about whose relationships meaningful statements can be made without reference to concrete components. We leave the search for such properties open as a research topic.

Modelling multiple intrinsic properties of the same component is actually quite easy. Each property can be modelled as though it existed in isolation. That is, for each intrinsic measurement we provide a model mapping from the corresponding context model to the underlying application model of the concrete component, and we specify any constraints required for expressing the property we are interested in.

```
 1 ---------------- MODULE CompleteCounter ----------------
   A counter component with constraints on accuracy and execution time.
 5 EXTENDS Reals, CounterInterface, RealTime

 7 --------------------------------------------------------
 8 VARIABLES MyCompExec, MyCompInState, MyLastAccuracy
 9 VARIABLES MyInternalCounter, MyDoHandle

11  Worst-case execution time measurement
12 _ExecTimeSpec ≜ INSTANCE CounterAppExecTime
13                  WITH ExecutionTime ← MyCompExec,
14                       inState ← MyCompInState,
15                       internalCounter ← MyInternalCounter,
16                       doHandle ← MyDoHandle,
17                       ldots

19  Accuracy measurement
20 _AccuracySpec ≜ INSTANCE AccuracyLimitedCounter
21                  WITH LastAccuracy ← MyLastAccuracy,
22                       inState ← MyCompInState,
23                       internalCounter ← MyInternalCounter,
24                       doHandle ← MyDoHandle,
25                       ldots

27  The actual (combined) component specification.
28 Counter ≜ ∧ _ExecTimeSpec!CounterComponent
29           ∧ _AccuracySpec!CompleteSpec
30           ∧ □(MyCompExec ≥ 4 + (2 − MyLastAccuracy))

32 ========================================================
```

Fig. 17 Sample component specification showing a relation between accuracy and execution time. We have removed some variable renamings to focus on the relevant parts

Example 18 (Accuracy and execution time) In Fig. 17, we see a sample specification expressing the relation between the accuracy of a Counter component and its execution time. This specification reuses the specifications of execution time and the Counter component. The formal specification of the accuracy measurement can be found in [81].

Notice how the two different measurements are applied to the same component (lines 11–25) and how these two specifications are then conjoined together with a constraint over both execution time and accuracy to form the complete specification of the component (lines 27–30). A specification like this can be used wherever a component specification can be used. In particular, it can be used to model a system and can then be formally evaluated in a feasibility proof.

The extrinsic properties of a service are the result of a transformation of the intrinsic properties of components by the container. Therefore, the corresponding constraints are typically known individually for each extrinsic measurement rather than in the form of relations between measurements. In any case, the specification technique is the same as for the specification of intrinsic properties of components, of course using service-related context models instead of component-oriented ones.

3.4.2 Extending the container specification to combine multiple properties

There is nothing in Definition 7 that prevents us from mapping one or more intrinsic properties to one or more extrinsic properties. However, in this case, container designers would have to predict and pre-specify every combination of intrinsic

المنارة للاستشارات
www.manaraa.com

non-functional properties that could occur. This is undesirable for at least three reasons:

1. *Combinatorial explosion*: even for a comparatively small number of intrinsic properties, the number of potential combinations becomes huge,
2. *Redundancy through a lack in modularity of the container strategies*: dealing with one property is often done the same way irrespective of the other properties supported in parallel, but may have to be reimplemented for every combination, and
3. *Unpredictability:* the container designer cannot predict the combinations of non-functional properties for which the container will be used; therefore, he must provide all combinations.

All of these issues could be resolved, if we follow a more orthogonal approach, in which container strategies only deal with individual non-functional properties and are combined to deal with multiple properties for a specific application. Thus, each non-functional property with its corresponding container strategy is essentially treated as an aspect in the sense of aspect-oriented programming (AOP) [25,41]. There have been a few approaches towards building such containers in the research community; for examples see [5,77,78,80].

Before we can discuss such a more orthogonal approach, we need to revisit our definition of a container specification. So far, we have unified the specification of the container and the container strategy, in effect only considering containers with exactly one container strategy. However, if we want to support more than one non-functional property in an orthogonal manner, containers must support more than one strategy. We, therefore, refine our definition of a container specification:

Definition 13 (*Container specification*) (*Multiple container strategies*) A container specification S_C is a set of container strategy specifications:

$$S_C = \bigcup_{i=1}^{n} \left\{ S_{CS}{}^i \right\}, \quad \text{for some } n \in \mathbb{N}$$

In a specification supporting such a modularised container we need to solve the following issues:

1. Selection of appropriate strategies.
2. Interactions between container strategies.

We will discuss them in more detail in the following:

Selection of appropriate strategies We need to select the container strategies to be applied in the context of a specific system. This is comparatively easy in the case where S_C contains only one strategy for each intrinsic property relevant to the system. However, a container may support more than one container strategy dealing with the same intrinsic property, possibly in different ways. In each case, it must eventually be clear which strategies to apply.

Example 19 (Container strategy selection) A container may have two strategies determining CPU demand for component based on its execution time: One strategy that guarantees an upper bound on response time and one that minimizes jitter of the result stream based on parameters of the request stream. Depending on the requirements of a certain application one of these strategies—or both—must be selected.

We have two basic possibilities for realising container strategy selection:

1. We can specify the container strategies to be used explicitly in the system specification.
2. The system can select an optimal set of container strategies, based on the available strategies, components and resources, and the requirements and preferences of the user (*cf.* e.g. [45,59]).

The second possibility is still an open field for research. Here, we restrict ourselves to the first possibility.

Interactions between container strategies Different strategies (even for different properties) may have an influence on each other. For example, consider again the two strategies from above. The strategy dealing with jitter will likely make a decision about CPU allocation which differs from the decision made by the response-time strategy. A conflict arises when both strategies need to affect the same component.

This issue can be split into two sub-problems:

1. *Description of possible interactions*: We need to specify where and how interactions between container strategies can occur.
2. *Resolution of conflicts created by strategy interactions*: We need to specify what to do when an interaction occurs.

These sub-problems depend on the way container strategies are selected: When explicitly specifying the container strategies to be used, we only need to check for conflicts; this is much simplified by a formal specification of the container strategy as proposed in this article. When container strategies are selected by the system based on some optimisation criterion, these sub-problems become much more difficult.

The whole area of feature interaction is still not well understood in the research community. For this reason, a complete treatment of the interactions between container strategies is

Springer
www.manaraa.com

well beyond the scope of this article and must be left open for future research.

3.5 Summary

In this section, we have presented the core concepts of our semantic framework for the specification of non-functional properties of component-based systems. We have seen, how context models and their use in the definition of measurements enables us to define a terminology for such specifications independently of the specific components or services whose properties are to be specified. We have further seen how we can use a mapping relation between transition systems to apply these abstract measurement definitions to concrete application models. Furthermore, we have seen that, because of the way knowledge about non-functional properties is distributed between the different players in a component market, we need to distinguish *intrinsic* and *extrinsic* non-functional measurements and properties. That is, we need to distinguish properties that solely depend on how a component is *implemented* from those properties that depend on how the component is *used.* After this presentation of our semantic framework, in the next section, we are going to discuss how this formalism can be applied.

4 Application of the framework

A formal specification framework is only as useful as its applications. For this reason, in this section, we discuss two applications of our semantic framework. We begin by showing how the framework can form the basis of a new specification language for non-functional properties of component-based systems. In the second sub-section, we show how the framework can be used for formalising requirements of analysis techniques.

4.1 A new specification language for non-functional properties

The TLA$^+$ specifications we have discussed so far, are pretty lengthy and complicated already for rather simple specifications. A lot of this complexity is accidental. It is caused more by technicalities of TLA$^+$ and the level of detail that such specifications require, rather than by the inherent complexity of the specifications themselves. It would be nice to be able to hide this complexity from the authors and readers of specifications. This can be done by introducing a more abstract specification language that provides customised concepts for non-functional specifications of component-based systems. The semantics of these concepts can be given by a context-free translation into equivalent TLA$^+$ specifications. In this section, we are going to present a simple example of such a

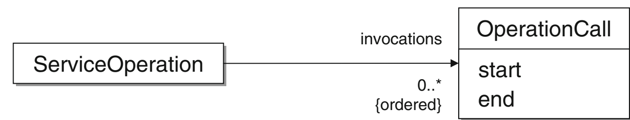

Fig. 18 A UML class diagram presenting the static part of the context model used in defining the response time measurement

specification language. This language is intended as a proof of concept. More research, especially into the integration with languages like UML, is required to design a practical specification language.

Comparing the semantic concepts available in existing specification languages for non-functional properties (e.g., [1,29,63]) to the concepts defined above, we see that many important concepts are missing from these languages. In particular, none of these languages provides support for the kind of partial specifications that we have discussed here and that are essential in the context of component markets; containers and resource specifications are not supported, intrinsic and extrinsic properties are not distinguished. Hence, we propose a new language, which takes inspiration from the languages cited above, but provides all the concepts identified in our work. We call this language *quality-modelling language for component-based systems* or *QML/CS* for short.

In this section, we first present the syntax of QML/CS and then discuss how its semantics can be defined based on our semantic framework.

4.1.1 Syntax of QML/CS

A QML/CS specification consists of a number of declarations. Each declaration provides either the definition of a measurement or an actual non-functional specification in the sense of Sect. 3.2. The context model of a measurement definition as well as the application model for any other specification is given through a UML model consisting of a class diagram and a state-chart description of the behaviour.

A measurement is defined using the following syntax:

```
in context <URIContextModel>
declare measurement
  <Type> <Name> (<ParameterList>) {
  spec <OCLExpression> ;
}
```

where <URIContextModel> is the address of a context model definition, <Type> and <Name> indicate type and name of the new measurement, resp., <ParameterList> is a list of parameters to the measurement, the types of these parameters come from the context model, and <OCLExpression> is an Object-Constraint Language (OCL) [56] expression defining the semantics of the new measurement.

المنارة للاستشارات
www.manaraa.com

Example 20 (Response time specified in QML/CS) Figure 18 shows a UML diagram that gives the static part of the context model of a service's operations. The important information here is that for every operation, we can access a list of invocations. In addition to this static diagram, there also exists a dynamic specification, indicating that a new `OperationCall` object is created whenever the operation is invoked. `OperationCall.start` and `OperationCall.end` store the moment in time when the operation was invoked and when the invocation was finished.

Based on this context model (with a URI of `RTContext`), we can proceed to define `response_time`:

```
in context RTContext
declare measurement real response_time
              (ServiceOperation op) {
  spec op.invocations->last.end -
       op.invocations->last.start ;
}
```

We have chosen to explicitly represent measurements as functions from a current system state to some value. This makes it difficult to directly express those measurements that depend on a history of states (such as `response_time` above). For this reason, measurement definitions need to be performed in a two-step manner: First, we define *probes* in the context model. These are defined as formal measurements as per Definition 2. In Example 20, we have defined the `end` and `start` probes measuring end and start time of the last operation invocation, respectively. These probes are then used in a second step in the definition of a measurement in QML/CS to express what values are combined, and in what manner, to produce the current measurement value. Therefore, OCL is sufficient to express the semantics of a measurement declared in QML/CS, even though it does not support any temporal-logic operators.

Measurements so defined can be used for expressing non-functional properties of components or services. Component or service specifications are based on a UML model of the component or service. Therefore, we need to repeat only those elements that are of relevance to our specification:

```
application <URIApplicationModel>;

declare component <Name> {
  <UsesOrProvidesStatements>

  <NonFunctionalPropertySpecification>
}
```

`<URIApplicationModel>` represents a location where the application model can be found. `<UsesOrProvidesStatements>` is a list of operation signatures, stream interfaces, etc. each preceded by the key word `uses` or `provides`. The elements of this list indicate the part of the component's interface that is relevant for the specification of non-functional properties. Only elements that have been defined in the application model can be listed here. `<NonFunctionalProperty Specification>` is a temporal-logic specification of the non-functional property to be expressed. It may contain all standard logical connectives written in the same manner they are expressed in OCL (for example, `and` or `implies`) and, additionally, the temporal-logic connectives `always` and `sometimes`. The possible atomic expressions are comparisons between a measurement parametrised with elements from `<UsesOrProvides Statements>` and a value or an expression, possibly using additional measurements. A service specification is syntactically quite similar, the only difference being that we use the word `service` instead of `component`.

Example 21 (Counter response time) Assuming we have a model `Counter` that represents the application model of our Counter component from before, we can provide the following specification to indicate a constraint on the response time of the `getData()` operation:

```
application Counter;

declare service Counter {
  provides int getData();

  always response_time (getData) < 60;
}
```

This constraint states that the response time of `getData()` is always less than 60 units of time.

Resources are specified in two stages: First, we declare an *abstract resource,* defining the interface that can be used to describe resource demands and check the resource's capacity:

```
in context <URIResourceModel>
declare abstract resource <Name> {
  demand <Type>;

  service (Set(<Type>) demand)
    = <ServiceTemporalExpression> ;

  always (capacityLimit(demand)
          => service (demand));
}
```

defines an abstract resource of name `<Name>`. An abstract resource definition is always accompanied by a resource model (located at `<URIResourceModel>`) that gives an abstract description of the overall behaviour of the resource, defines a type for expressing individual resource demands (named `<Type>`), and provides the vocabulary required to

المنارة للاستشارات www.manaraa.com

talk about the resource. This vocabulary is then used in `<ServiceTemporalExpression>` to specify the service the resource can provide. The last line of the abstract resource specification makes explicit the principle behind resource specifications; namely, that a resource provides its service as long as its capacity limit is respected. Notice, though, that no concrete capacity limit has been specified yet. This happens in the next step, where we define a concrete resource.

```
declare resource <Name>
        of <AbstractName> {
  capacityLimit (Set(<Type>) demand))
      = <OCLExpression> ;
}
```

is then used to define a concrete resource `<Name>` as a specialisation of an abstract resource `<AbstractName>`. The important thing for a concrete resource is the definition of the capacity condition for this resource. This is done using an `<OCLExpression>` in the declaration of the resource.

Example 22 (Resource specification for CPU) Assume that `CPUModel` references a state-machine model of a process continually assigning processing slots to any one task in a set of tasks. `CPUModel` provides a function `timeAlloted` that returns the amount of time allocated to a given task in the last period and a collection `scheduledTasks` of (id, demand) tuples representing the tasks executing on the CPU. Further, type `Task` defines the structure required to describe a single task demand. It has fields for the period, worst-case execution time and relative deadline.

Based on this model, we can define resource CPU as follows:

```
in context CPUModel
declare abstract resource CPU {
  demand Task;

  service (Set(Task) demand)
    = always (
        -- All tasks in demand
        -- are scheduled
        scheduledTasks
          ->collect (t | t.demand)
          ->includesAll(demand) and
        -- Only demand is scheduled
        scheduledTasks->size()
          = demand->size() and
        -- All scheduled tasks meet
        -- their deadline
        scheduledTasks
          ->forAll (t |
            timeAlloted (t.id)
              >= t.demand.wcet
          )
      );

  always (capacityLimit(demand)
         => service (demand));
}
```

This service specification stipulates that all tasks in the demand set are running on the CPU and that all these tasks meet their deadlines.

Based on this definition, we can now define a concrete, RMS-scheduled CPU as follows:

```
declare resource RmsCpu of CPU {
  capacityLimit (Set(Task) demand))
      = demand->iterate (t: Task;
                         acc: Real |
          acc + t.wcet / t.deadline
        ) <=
        demand->size() *
        (2.sqrt(demand->size()) - 1);
}
```

To define this concrete CPU, we have to define its capacity limit. This is done through the expression above, which is an OCL rendering of the standard RMS schedulabitily criterion also given on Page 6.

Container specifications bind together resources and components to provide services:

```
declare container <Name>
    (<OptionalParameterList>) {
  <HelperVariables>
  requires
    <ComponentPatterns>
    <ResourceConstraints>

  provides
    service
      implemented by <ComponentName> {
      <NonFunctionalPropertySpecification>
    }
}
```

The `requires` part of this specification indicates the components and resources the container uses. A `<ComponentPattern>` looks much like a component declaration without the `declare` key word and with abstract names for used or provided operations etc. A `<Resource Constraint>` describes a capacity requirement for some abstract resource:

```
resource <AbstractName>
  .canHandle (<ResourceDemand>);
```

where `<AbstractName>` is the name of an abstract resource and `<ResourceDemand>` is a list of concrete

المنارة للاستشارات
www.manaraa.com

values for the demand parameters of this abstract resource. Multiple demand items are separated by a semicolon.

The `provides` part of the container specification indicates the service the container provides. While a service can be implemented by a network of components, there will typically be a top-level component that provides the direct interface to the service implementation. This component can be indicated by name after the `implemented by` key words. The name to be used must have occurred as the name of a component pattern in the `requires` section of the container specification.

Example 23 (Container specification) Based on the specifications from previous examples, we can now define a simple container strategy as follows:

```
declare container SimpleContainer
              (ResponseTime: Real) {
  ExecutionTime: Real;

  requires
    component C {
      provides op1();

      always execution_time (op1) <
             ExecutionTime;
    };
    resource CPU.canHandle (
        Set{Task(
            period = ResponseTime,
            deadline = ResponseTime,
            wcet = ExecutionTime)});

  provides
    service implemented by C {
      ExecutionTime < ResponseTime =>
           always response_time (op1) <
               ResponseTime
    }
}
```

After declaring a helper `ExecutionTime` to implicitly represent the execution time of any component made available to the container, we continue to enumerate the requirements of this strategy: The container requires a component for which `ExecutionTime` is an upper execution-time bound and a CPU that can handle one task for executing the component's code. In return, the container provides a service with the same functionality as the component (expressed by `service implemented by C` with a response time bounded by the specification parameter `ResponseTime`.

Finally, we need to bind all partial specifications together into a system specification:

```
system <SystemName> {
  <InstancesList>

  container
    uses <ComponentsAndResources>;
  container
    provides
      <ServiceSpecificationName> <Name>;
}
```

is the syntax for such a system specification. `<SystemName>` is a name used to identify the system. `<InstancesList>` is a list of entries of the form:

```
instance
  <ComponentResourceOrContainerName>
    <Name>;
```

defining a set of components and resources to be used by the system, as well as the (single) container to connect components and resources and provide a service. The second part of a system specification is responsible for wiring these instances. This happens by allocating instances to the parameters of the container using `container uses`. `<ComponentsAndResources>` represents a comma-separated list of instance names as defined in `<Instances List>`. The order of elements in this list corresponds to the order of elements in the `<requires>` part of the container. Notice that specifications could be slightly more compact without the instance list (instances could be directly named in the uses and provides parts), but we keep this form for clarity and explicitness.

Example 24 (System specification) The following system specification binds together the specifications from the examples before. Notice, that the name for the single container instance cannot be freely chosen, but must always be `container`.

```
system CompleteSystem {
  instance CounterComp c;
  instance RmsCpu cpu;
  instance
      SimpleContainer(60) container;

  container
    uses c, cpu;
  container
    provides
      Counter cServ;
}
```

Springer
المنارة للاستشارات
www.manaraa.com

4.1.2 Semantics of QML/CS

The semantics of QML/CS can be given by mapping QML/CS expressions into TLA$^+$ specifications according to the framework introduced in Sect. 3.

Notice that every QML/CS specification is implicitly given relative to a context model defining the types of potential parameters to measurements etc. Such context models for QML/CS will typically be expressed using mechanisms like UML class and state diagrams; as has, for example, been done in [64]. A mapping from the UML-based context-model representation to a TLA$^+$-based expression of the context model is required. Because context models and QML/CS specifications are so closely linked, we will also have to take this mapping into consideration when defining the semantics of QML/CS. For the discussions below, we will assume a very simple computational model and a very simple mapping function. The computational model essentially provides the notions of operations of services and of components. These are then mapped to the TLA$^+$ context models from Sect. 3.2.

TLA$^+$ can be quite unwieldy at times. In order to make the following definitions easier to understand, we introduce a template-based mechanism for the creation of TLA$^+$ specifications from QML/CS: We define a function $\iota\,(T, n, (p))$ taking a TLA$^+$ template T, a name n, and a list of parameters p and producing a TLA$^+$ specification. Such a template is very similar to a specification: It starts with a comment indicating the formal parameters of the template—these will be mapped to the actual parameters in p by the call to $\iota\,(T, n, (p))$. The rest of the template is essentially a normal TLA$^+$ module. The name of the module is replaced with n by $\iota\,(T, n, (p))$. In the body of the module definition, $\iota\,(T, n, (p))$ will replace any occurrence of a formal parameter with the corresponding (by order) actual parameter from p. Additionally, we use the construction $\hat{\forall} x \in y : t$ in the template to produce an instantiation of the TLA$^+$ text t for every x in y, where y typically is a formal parameter of the template. Examples of such templates will be given in the figures to follow.

QML/CS is structured analogously to the concepts in our semantic framework. For this reason, we can discuss the mapping of each construct of QML/CS individually without cross-referencing mappings for other constructs. We begin by mapping measurement definitions (cf. Page 185). For each individual QML/CS measurement, we define a corresponding TLA$^+$ measurement, using the computational model as the context model. The type(s) of the formal parameters of the QML/CS measurement definition determine(s) which part of the computational model will be used as the context model for the measurement and when measurement values will be determined. For example, because in Example 20 we have used a parameter of type `ServiceOperation`, we know to use the context model for services and the corresponding TLA$^+$ templates in the translation. Our example computational model only considers operation invocation and we have decided to update any operation-related measurements always at the end of each operation invocation. This has been encoded in probes in the computational model, which themselves are already measurements following the definitions of our semantic framework. Additionally, for this example, we only consider QML/CS measurements with at most one parameter.

```
   \* (n, fp, d, p)
 2 ┌──────────────── MODULE ServOp ────────────────┐
 4 EXTENDS RealTime
 6  Importing the context model
 8 VARIABLES ServUnhandledRequest, ServInState
 9 VARIABLES ServOpStart, ServOpEnd, ServInCall
11 fp.n ≜ INSTANCE muCQMLContext
13 ├────────────────────────────────────────────────┤
15 VARIABLE n  The actual measurement
17 ├────────────────────────────────────────────────┤
19 Init ≜ ∧ n ∈ d
21 OnFinishRequest ≜ fp.n!Service!FinishRequest ⇒ ∧ n' = ⟦p⟧_s(fp)
23 Spec ≜ ∧ fp.n!ServSpec
24        ∧ Init
25        ∧ □[OnFinishRequest]_⟨n⟩
26 └────────────────────────────────────────────────┘
```

Fig. 19 TLA$^+$ measurement definition template for service operations. *Bold font* indicates place holders to be replaced with information from the QML/CS specification. The variable declarations on *lines 8–9* are a technical necessity of TLA$^+$ when instantiating the context model

The semantics of a QML/CS measurement specification is given by:

$$
\begin{aligned}
&[\![(d, n, fp, p)]\!] = \\
&\quad \textbf{if } |fp| = 1 \wedge fp.t = SERVOP \\
&\qquad \textbf{then } \iota\,(ServOp, Char_n, (n, fp, d, p)) \\
&\qquad \textbf{else } \textbf{ if } |fp| = 1 \wedge fp.t = COMPOP \\
&\qquad\qquad \textbf{then } \iota\,(CompOp, Char_n, (n, fp, d, p)) \\
&\qquad\qquad \textbf{else } \perp
\end{aligned}
$$

where (d, n, fp, p) is an abbreviation for a QML/CS measurement

```
declare measurement d n (fp) {
  spec p ;
}
```

The $\perp$ symbol indicates that no semantics can be given for the measurement specification. We use $|x|$ to indicate the cardinality of set x. $fp.t$ refers to the type part of an arbitrary element of fp; because of $|fp| = 1$, this element is unambiguously defined. $ServOp$ and $CompOp$ refer to the TLA$^+$ templates shown in Figs. 19 and 20.

Particularly, note how the QML/CS measurement's definition p is inserted into the template on Line 21 of Figs. 19 and 20: Again, a semantic mapping function is applied to it, but this time it maps OCL expressions to TLA$^+$. This mapping is a parameter to the semantic mapping of QML/CS,

www.manaraa.com

```
   \* (n, fp, d, p)
 2 ------------------ MODULE CompOp ------------------
 4 EXTENDS RealTime
 6  Importing the context model
 8 VARIABLES CmpUnhandledRequest, CmpInState
 9 VARIABLES CmpOpStart, CmpOpEnd, CmpInCall
11 fp.n ≜ INSTANCE muCQMLContext
13 ----------------------------------------------------
15 VARIABLE n  The actual measurement
17 ----------------------------------------------------
19 Init ≜ ∧ n ∈ d
21 OnFinishRequest ≜ fp.n!Component!FinishRequest ⇒ ∧ n' = [[p]]_c(fp)
23 Spec ≜ ∧ fp.n!CmpSpec
24        ∧ Init
25        ∧ □[OnFinishRequest]_⟨n⟩
26 ----------------------------------------------------
```

Fig. 20 TLA$^+$ measurement definition template for component operations. *Bold font* indicates place holders to be replaced with information from the QML/CS specification. The variable declarations on *lines 8–9* are a technical necessity of TLA$^+$ when instantiating the context model

completely dependent on the mapping of the computational model to TLA$^+$. Note that this translation may be different depending on the type of fp. The decision whether to use the component or the service variant has been made by using two slightly different mapping functions: $[\![.]\!]_c$ is the mapping for the component version and $[\![.]\!]_s$ is the one for the service version.

We have only considered QML/CS measurements with one formal parameter above. In theory, a measurement can have any number of parameters. So far, we have not found an example for this yet. In any case, the above schema can easily be extended to more than one parameter by providing one mapping template per relevant combination of parameter types.

Example 25 (Semantics of `response_time`) Figure 21 shows the TLA$^+$ specification corresponding to the QML/CS definition of `response_time`. It has been created quite straight-forwardly by instantiating the $ServOp$ template in Fig. 19.

The most interesting part is probably the result of $[\![p]\!]_s(fp)$, which can be seen on lines 20–21. In the computational model, $ServOpStart$ and $ServOpEnd$ are used to represent `op.invocations->last().startTime` and `op.invocations->last().endTime` for a service operation `op`, respectively. Note that this mapping depends only on the UML representation of the computational model and on how this is mapped to its TLA$^+$ representation.

QML/CS component and service specifications (cf. Page 186) combine a number of concepts from our semantic framework: The `<NonFunctionalProperty Specification>` corresponds to our notion of a property specification, either for a component or for a service. However, it is not expressed in terms of abstract measurements,

```
 1 -------------- MODULE char_response_time --------------
 3 EXTENDS RealTime
 5 VARIABLES ServUnhandledRequest, ServInState
 6 VARIABLES ServOpStart, ServOpEnd, ServInCall
 8 op ≜ INSTANCE muCQMLContext
10 ----------------------------------------------------
12 VARIABLE response_time
14 ----------------------------------------------------
16 Init ≜ ∧ response_time ∈ R
18 OnFinishRequest ≜ op!Service!FinishRequest ⇒ ∧ response_time' = ServOpEnd −
19                                                              ServOpStart
21 Spec ≜ ∧ op!ServSpec
22        ∧ Init
23        ∧ □[OnFinishRequest]_⟨response_time⟩
25 ----------------------------------------------------
```

Fig. 21 The TLA$^+$ specification for `response_time`

```
   \* (n, ops, p)
 2 ---------------- MODULE CompOpConstr ----------------
 4 EXTENDS S_App, RealTime
 6 ∀̂op ∈ ops :
 7   ∀m ∈ Measures(op, p) : {
 9 VARIABLES ⟨m, op.n⟩ServUnhandledRequest, ⟨m, op.n⟩ServInState
10 VARIABLES ⟨m, op.n⟩ServOpStart, ⟨m, op.n⟩ServOpEnd, ⟨m, op.n⟩ServInCall
11 VARIABLES ⟨m, op.n⟩CmpUnhandledRequest, ⟨m, op.n⟩CmpInState
12 VARIABLES ⟨m, op.n⟩CmpOpStart, ⟨m, op.n⟩CmpOpEnd, ⟨m, op.n⟩CmpInCall
14 ⟨m, op.n⟩Spec ≜ INSTANCE Char_m
15 ⟨m, op.n⟩ModelMapping ≜ □[[m, op]]_MM
17 }
19 Spec ≜ ∧ ∀̂op ∈ ops :
20            ∀m ∈ Measures(op, p) : {
21              ∧ ⟨m, op.n⟩ModelMapping
22              ∧ ⟨m, op.n⟩Spec!Spec
23            }
24        ∧ [[p]]_P
25 ----------------------------------------------------
```

Fig. 22 TLA$^+$ template for component specifications. *Bold font* indicates place holders to be replaced with information from the QML/CS specification

but is already related to specific features of a specific component or service. Correspondingly, the semantic mapping must include a model mapping from the context models of any measurements used to the application model referenced. Figure 22 shows the TLA$^+$ specification template used in translating component specifications:

$$[\![(n, ops, p)]\!] = \iota\,(CompOpConstr, Constr_n, (n, ops, p))$$

where (n, ops, p) is a short notation for

```
declare component n {
  ops;

  p;
}
```

i.e., ops is a list of operations, and p is a temporal-logic expression formulating constraints over certain measurements.

Figure 22 is a bit more complicated. It uses some additional functions to extract information from the template

المنارة للاستشارات
www.manaraa.com

parameters. In particular, it uses a function $Measures(op, p)$ (cf. lines 7 and 20) that analyses p and returns a set of measurements which are applied in p to op. Furthermore, the template employs two more semantic mapping functions: (1) $[\![.]\!]_P$ (Line 24) maps p into its TLA$^+$ representation, and (2) $[\![.]\!]_{MM}$ takes a measurement and an operation and produces a model mapping specification to reflect the fact that the measurement has been applied to the operation. Service specifications are translated quite similarly, the only difference being that the service context model is used instead of the component context model.

Mapping resources, container specifications, and system specifications happens analogously. For reasons of space, we omit a detailed discussion in this article. The semantics of QML/CS can be explained completely by mapping QML/CS onto TLA$^+$ specifications following our semantic framework.

4.2 Specifying the Interface of analysis techniques

Besides giving the semantics of QML/CS, another application of our semantic framework is the precise specification of analysis methods for non-functional properties. Wherever such analysis methods (for example, queueing-network analysis for performance properties, or security-risk analysis, etc.) are presented in the literature, their pre-conditions and result values are characterised in natural language only. This leaves room for interpretation. Recently, in the context of model-driven architecture (MDA) [42,57], a new concept has been proposed to provide for a more formal representation of model transformation and analysis expertise: MDA tool components (MDATCs) [9,58]. An MDATC is a collection of related models and meta-models representing knowledge about a specific model transformation. MDATCs can be executed in so-called MDA containers integrated into CASE tools. Thus, modelling knowledge can be packaged in interoperable units that can be reused and traded across business boundaries. A formal description of analysis techniques—in particular of their interfaces—is obviously a key pre-requisite for reusing them as an MDA tool component.

In our view, an analysis method for non-functional properties is nothing but an operation which takes a non-functional specification obeying some additional rules and produces a constraint over some measurement regarding that specification. The additional rules mentioned could constrain which non-functional constraints are contained in the specification, or what type of application it describes. We can, therefore, formalise the interface of an analysis method using pre- and post-conditions as for any other operation. The most important difference versus normal operation specification is that the parameters of the operations themselves represent specifications again. Providing such a formal specification of the pre-requisites for an analysis method would even allow formal proofs of the applicability of the analysis method; but more importantly, it provides an unambiguous description of what input is needed to perform the analysis.

Example 26 (Performance analysis parameters) Figure 23 shows a formal representation of the pre- and post-conditions for performance analysis for a sequence of operation calls; that is for a performance-critical scenario. This specification assumes that this analysis can be represented as an operation of the general signature

```
perf_analysis (in Nat NumOps,
               in Scenario Services,
               in Real []  ResponseTimes,
               out Real GlobalTimeConstr)
```

i.e., it takes a scenario of a number (`NumOps`) of `Services` being invoked, for each of which the maximum response time (`ResponseTimes`) is given, and produces a real value for the global time constraint.

The specification imposes constraints on these parameters (in the following, line numbers refer to the lines in Fig. 23):

- `Services` describes a set of service operations. lines 40–75 specify that the service operations are invoked one after the other.
- `ResponseTimes` is an array containing the maximum response time for each service operation in the order in which they are described in `Services` (*cf.* Line 45).
- `GlobalTimeConstr` represents an upper bound for the time between the invocation of the first operation in `Services` and the return of the last such operation (*cf.* lines 84–101).

The most interesting aspect of this specification is how operation invocations are specified: We have lifted the service operation context model from Fig. 2 to a sequence of operations by collecting the corresponding state variables in a function `Services` from the operation number to a tuple containing all the state variables. This function is then used to define an operator $Operation(n)$ (lines 40*ff.*) that represents an individual operation by its context model. An operation call is then described by including the state machine from that context model (*cf.* lines 59 and 65). This way, we can reuse the concept of a model mapping (see Sect. 3.2) to define the application of the analysis method to a specific application model and a specific sequence of operation calls. The performance expert needs to present such a mapping to validate that he has chosen the right parameter values for the performance analysis.

4.3 Summary

In this section we have discussed applications of our semantic framework defined in Sect. 3. We have, first, defined a new

المنارة للاستشارات
www.manaraa.com

```
1  ──────────────── MODULE SPEAnalysis ────────────────
3  EXTENDS RealTime

   Parameters:

   Services         An array containing the state variables for the response time model. This es-
                    sentially describes the sequence of calls. Even though parameters are normally
                    specified as constants, this must be specified as a variable as the individual
                    fields will change as the services are executed. The service state machine
                    is modelled slightly differently from Fig. 2: each state is represented by ele-
                    ments inState and unhandledRequest. State 'RequestAvailable' from Fig. 2 is
                    identical to ("Idle", TRUE).
   ResponseTimes    An array containing the response time constraints for each operation in Ser-
                    vices.
   NumOps           The number of operations in the sequence of calls.
   GlobalTimeConstr The analysis result value.

24 CONSTANT NumOps

26 VARIABLE Services
27 ASSUME Services ∈ [{1 .. NumOps} → [LastResponseTime : Real,
28                                    Start : Real,
29                                    inState : {"Idle", "HandlingRequest"},
30                                    unhandledRequest : BOOLEAN ]]
31 CONSTANT ResponseTimes
32 ASSUME ResponseTimes ∈ [{1 .. NumOps} → Real]
33 CONSTANT GlobalTimeConstr
34 ASSUME (GlobalTimeConstr ∈ Real) ∧ (GlobalTimeConstr ≥ 0)

   Import the response time definition for each service.
40 Operation(n) ≜ INSTANCE ResponseTimeConstrainedService
41                 WITH LastResponseTime ← Services[n].LastResponseTime,
42                      Start ← Services[n].Start,
43                      inState ← Services[n].inState,
44                      unhandledRequest ← Services[n].unhandledRequest,
45                      ResponseTime ← ResponseTimes[n]

47 ─────────────────────────────────────────────────────
   Define the sequence in which the service are executed. Here, we simply express a sequential
   execution from 1 to NumOps.
52 VARIABLES CurOp, InOp

54 Init ≜ ∧ CurOp = 1
55        ∧ InOp = FALSE

57 DoOp(n) ≜ ∧ CurOp = n
58           ∧ InOp = FALSE
59           ∧ Operation(n)!Serv!StartRequest
60           ∧ InOp' = TRUE
61           ∧ ∀ m ∈ {1 .. NumOps} : m ≠ n ⇒ UNCHANGED Services[m]

63 FinishOp(n) ≜ ∧ CurOp = n
64               ∧ InOp = TRUE
65               ∧ Operation(n)!Serv!FinishRequest
66               ∧ InOp' = FALSE
67               ∧ CurOp' = n + 1
68               ∧ ∀ m ∈ {1 .. NumOps} : m ≠ n ⇒ UNCHANGED Services[m]

70 Next ≜ ∃ n ∈ {1 .. NumOps} : ∨ DoOp(n)
71                             ∨ FinishOp(n)

73  This encodes the expectations of the analysis method
74 Precond ≜ ∧ Init
75           ∧ □[Next]_⟨CurOp, InOp⟩

77 ─────────────────────────────────────────────────────
   Based on the sequence defined above, define how we measure. DoStart measures when we begin
   executing the first service. DoEnd is triggered at the end of the last service.
84 VARIABLES Start, GlobalTime

86 InitGM ≜ ∧ Start = 0
87          ∧ GlobalTime = 0

89 DoStart ≜ DoOp(1) ⇒ ∧ Start' = now
90                     ∧ UNCHANGED GlobalTime

92 DoEnd ≜ FinishOp(NumOps) ⇒ ∧ GlobalTime' = now
93                            ∧ UNCHANGED Start

95 NextGM ≜ DoStart ∧ DoEnd

97  This encodes the result of the analysis method
98 PostCond ≜ ∧ Precond
99            ∧ ∧ InitGM
100             ∧ □[NextGM]_⟨CurOp, InOp, Start, GlobalTime⟩
101           ∧ □(GlobalTime ≤ GlobalTimeConstr)

103 ────────────────────────────────────────────────────
```

Fig. 23 Formal specification of Use-Case-based analysis

specification language—QML/CS—for the specification of non-functional properties of component-based systems and shown how its semantics can be defined based on our framework. Second, we have demonstrated how our formal concepts for non-functional properties can be used to clarify the interface of analysis techniques for non-functional properties. Both applications form both an evaluation of our formal framework and a contribution in their own right.

This section concludes the main part of this article. In the next section, we give an extensive discussion of related work.

5 Related work

The work here presented merges two movements in research: Component-based software engineering (CBSE) and specification of non-functional properties. Consequently, we will review the literature from these two directions and show how the work we presented fits into the picture.

5.1 Application structuring techniques

In the research community, the question 'What is a component?' has led to much discussion and controversy [12,14,20,23,31,39,75]. Nowadays, the definition given by Szyperski in [75] has become more or less accepted:

> "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties." [75]

Thus, software components are elements of software which need to be composed with other components to form an application. The different components in an application may be developed by different component developers. They can interact because there exists a standardised component runtime environment offering a space to live in to the components. This definition was first formulated at the 1996 European Conference on object-oriented programming (ECOOP) as an outcome of the Workshop on component-oriented programming, and has since become the definition of a component most accepted in the community.

Szyperski and others even envision a market of commercial-off-the-shelf (COTS) components where companies no longer produce complete applications, but highly reusable software components which are sold to third parties.This part of the definition of a software component is, however, controversial. There are those who, like Szyperski, hold that the main benefit in CBSE lies in the ability to reuse pieces of code produced by third parties. On the other hand there are those—for example, Cheesman and Daniels [14]—who

Springer

www.manaraa.com

claim that the main benefit lies in a gain of flexibility through a strong modularisation and clean separation of concerns in the application structure. The concepts presented in this article provide benefits to both sides in this discussion. On the side of Cheesman and Daniels, we show concepts which allow a modular consideration of non-functional properties in addition to the functional properties. On the other side, the concepts allow non-functional specifications of component implementations to be written without knowledge of the contexts in which the component will be deployed. This is an important precondition for a component market.

Cheesman and Daniels [14] also introduce another important notion into the world of CBSE, namely the idea of *component forms*. They argue that the concept of a component varies depending on where in the project life cycle one is. Cheesman and Daniels distinguish four major component forms: component specification, component implementation, installed component, and component object. We consider non-functional properties associated with component implementations. In particular, the concept of *intrinsic properties* introduced in this article applies to component implementations, capturing all those properties determined by the specific algorithms chosen to implement a component specification as opposed to the properties determined by how the component is used. However, it should be noted that there are some non-functional properties (in addition to the extrinsic properties) that can only be determined at the level of installed component, or even component object. In our response-time example we assumed that we could determine the execution time for our component. However, execution time of a component heavily depends on the underlying hardware and runtime environment [15]—for example, the time it takes the specific CPU to execute certain machine code commands.

An important issue in the context of CBSE is *compositionality;* that is the ability to derive properties of a composed system directly from properties of its constituting elements (*viz* the components) without the need to analyse the inner structure of these elements. Werner and Richling [79] identify five types of compositionality: invariant quality, bound quality, disappearing quality, emerging quality, and transferred quality. The common usage of the term *compositional* seems to imply invariant quality; that is cases where the composed system has the same property as its constituents. In the context of this work we are especially interested in compositionality of properties concerning the behaviour of systems. Writing component specifications in rely–guarantee style—first introduced by Cliff B. Jones [40]—has proven very useful for compositional specifications. Abadi and Lamport have developed a *composition theorem* [4] which allows to compose temporal-logic rely–guarantee specifications expressed in extended TLA$^+$ [44]. Both TLA$^+$ and the composition principle are an important basis for this article. Hu and Marcus [37] have studied compositionality from a graph-theoretic viewpoint. Their work is related to [79], but it is more formal than that, presenting four different types of compositionality. They distinguish between properties of components, properties of the composition structure (called the *fusion* in their terminology) and properties of the composed system. The relations between these properties determine the different types of compositionality.

In recent years, a new approach called service-based software engineering has gained increasing importance. We use the concept of a service to distinguish between the user's view on a system and the internal view of the same system. Over and above this simple concept of a service, we do not use the more advanced concepts of service-based software engineering defining services as interaction patterns [43] or as partial component specifications [66]. We will, therefore, refrain from further reviewing the literature in this field.

5.2 Non-functional properties

The research in the field can be classified into two major categories: work concerning non-functional (a) requirements, and (b) properties of actual software artefacts. While our work is in the latter area of non-functional properties, we will review some work from the former field, because the insights gained there are also applicable to the theory of non-functional properties. The area of non-functional properties can be further divided into three sub-areas:[11]

1. *Basic contract concepts*: The work in this area is concerned with general observations of what is required to specify non-functional properties in a contractual manner. No choice is made about concrete specification languages or styles. Research rather attempts to explain how the concept of design by contract (and variants thereof) can be extended to non-functional properties. Most of the work already specifically addresses component-based software.
2. *Characteristic-specific approaches*: These approaches introduce new, or extend existing, formal description techniques to deal with specific non-functional properties or classes of such properties. Because our research belongs to the strand of measurement-based approaches we will not review this area in this article.
3. *Measurement-based approaches*: Work in this area makes non-functional measurements (often called *characteristics*) first-class citizens of specifications, and thus allows any kind of non-functional property to be expressed as long as the underlying measurement can

[11] A more extensive and occasionally updated review of literature in the field can be found at http://www.steffen-zschaler.de/bibliographies/nfp.php.

المنارة للاستشارات
www.manaraa.com

be formalised in the language. This is also the approach chosen in this article, because it is the approach with the greatest flexibility. At the same time, the high degree of generality makes it harder to make such specifications usable in property-specific analysis techniques.

The work of Hissam et al. [35] at the Software Engineering Institute at Carnegie Mellon University does not quite fit into this classification. The authors describe prediction-enabled component technology (PECT), which is a generic concept for combining a component technology with one or more analysis models for non-functional properties. The name PECT stands both for the generic concept and for an individual instantiation of the concept with a concrete component technology (e.g., EJB) and a concrete analysis model (e.g., Software Performance Engineering (SPE) [71]). Their approach shows some similarity to the approach proposed in this article, but the authors do not strive for a formal description of the general notions. Instead, they focus on explaining the application of specific analysis techniques to specific component models using their framework. Also, whether their approach is measurement-based or characteristic-specific seems to depend largely on the concrete analysis technique used.

5.2.1 Non-functional requirements

Chung et al. [18] present a framework for reasoning about design decisions which leads from non-functional requirements to the actual design and eventually the implementation. They use the notion of a *softgoal* to represent non-functional requirements, which may be imprecise, or subjective. Softgoals are related to each other as well as to *operationalisations* (representing possible realisations for a softgoal) to drive the software development process. Rationale for design decisions is explicitly recorded in the form of *claims.* In contrast to this approach, which is mainly concerned with transforming non-functional requirements into a running system, the approach presented in this article focuses on formally specifying non-functional properties of components and applications. The two approaches can be seen as complementary to each other in that our approach can be used to formally describe the properties of components which can be used when building applications following the process described in [18].

Various authors have given classifications of non-functional requirements (e.g., [38,50,72] some more are also reviewed in [18]), which can equally well be applied to non-functional properties. We will not review all the different classifications and their differences, but rather concentrate on some common characteristics, which seem to be recurrent. We base our explanation on the classification given by Sommerville [72, pp. 130 *ff.*], which to us is the most comprehensive one. A graphical representation of this classification can be found in Fig. 24. Sommerville identifies three main classes of non-functional requirements:

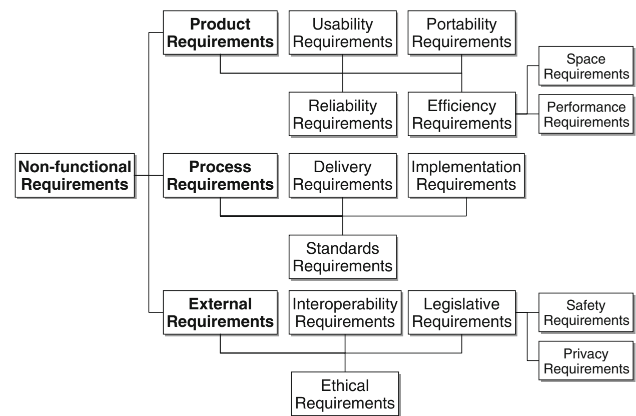

Fig. 24 Classification of non-functional requirements defined by Sommerville (from [72, p. 131])

1. *Product requirements*: These are requirements directly concerning the software system to be built. They include requirements relevant to the customer—such as usability, efficiency, and reliability requirements—but also portability requirements which are more relevant to the organisation developing the software.
2. *Process requirements*: Sometimes also called *organisational requirements,* these requirements "[…] are a consequence of organisational policies and procedures." They include requirements concerning programming language, design methodology, and similar requirements defined by the developing organisation.
3. *External requirements*: These requirements come neither from the customer nor from the organisation developing the software. They include, for example, requirements derived from legislation relevant to the field for which the software is being produced.

This classification is relevant in the context of this article because it allows us to express what kinds of non-functional properties our approach supports. It is clear that any classification of non-functional requirements that is not based on how these requirements can be elicited, can also be used as a classification of non-functional properties. Thus, simply replacing "requirements" by "properties" in Fig. 24 gives a classification of non-functional properties. We can classify the properties our approach can model as product properties.

5.2.2 Basic contract concepts

Beugnard et al. [8] propose to distinguish four levels of contracts, each level depending on all the lower levels: syntactic,

www.manaraa.com

behavioural, synchronisation, and QoS level. A contract can be negotiated the more flexibly the higher its level. It is obvious that syntactic contracts are as good as cast in stone once an interface has been defined. They form the basis for communication between the components so negotiating about them is all but impossible. On the other hand it is not unreasonable to expect components to be able to provide their services in a range of qualities so that clients can select between them and potentially even perform actual negotiations with bids and counter bids being exchanged between component and client. This article does not include concepts specifically made for contract negotiation. However, by defining a formal framework, we provide the basis on which negotiation partners can communicate precisely and unambiguously so that negotiations can be performed successfully.

Röttger and Aigner [62], and Selic [69] point out the importance of specifying the required resources in contracts for real-time properties. Both papers enhance the structure of component contracts adding an explicit description of the resources a component requires from its environment. This leads to a layered system where components on one layer are connected through their used and provided properties and the layers are connected by resource associations between components on different layers. In this article, we show that this approach is not directly feasible, because the resource demand of a component depends largely on how the component is used.

Reussner [60] proposed the concept of parametrised contracts, a more formal representation of dependencies between interfaces provided or required by a component. Parametrised contracts capture the dependencies inside a component as opposed to dependencies between components, which are expressed in more conventional contracts. The concept of parametrised contracts has originally been developed for functional specifications, but Reussner et al. [61] have extended this work to also include non-functional properties. Their parametrised contracts explicitly acknowledge the intra-component dependencies between provided and required properties. In our article, these dependencies become visible in the expression of intrinsic properties of components. Moreover, Reussner et al. show for the specific case of reliability that it is important to distinguish between properties inherent to a component implementation and properties which emerge from using the component. This supports the distinction of intrinsic and extrinsic properties proposed in this article.

Chimaris and Papadopoulos [16] present a quite different notion of *contracts*. In their view, a contract consists of both a specification of a non-functional property and an aspect (in the sense of aspect-oriented programming (AOP) [25,41]) that can be used to guarantee that property at runtime. It is thus a combination of our notions of measurement, non-functional property, and container strategy.

5.2.3 Measurement-based approaches

The approaches collected under this heading all make characteristics first-class citizens of a specification; that is, they allow characteristics to be defined as part of a specification. We call them measurement-based, because characteristics as defined by these approaches are essentially measurements in the sense of this article. The basic terms employed in this strand of research have been standardised by ISO and ITU [38]. The most important terms are:

QoS characteristic "A quantifiable aspect of QoS, which is defined independently of the means by which it is represented or controlled." For the reasons stated above, we also use the term *measurement* to mean characteristic.

QoS category "A group of user requirements that leads to the selection of a set of QoS requirements." Although it is tempting to view QoS categories as a representation of the classes discussed in Sect. 5.2.1, the definition is actually intended to classify applications into groups with commensurable types of non-functional requirements.

QoS management "Any set of activities performed by a system or communications service to support QoS monitoring, control and admission."

QoS mechanism "A specific mechanism that may use protocol elements, QoS parameters or QoS context, possibly in conjunction with other QoS mechanisms, in order to support establishment, monitoring, maintenance, control, or enquiry of QoS."

QoS policy "A set of rules that determine the QoS characteristics and QoS management functions to be used."

The approach presented here fits well into the framework provided by these definitions. We, essentially, give formal semantics to QoS characteristics (namely as intrinsic and extrinsic measurements), and to QoS Management, QoS Mechanism, and QoS Policy (namely the container specification). In addition to presenting a formal representation of these concepts, we also give a more detailed analysis and identify more fine-grained distinctions within these definitions. The source for these distinctions is our requirement to provide a semantics specifically targeted to component-based software systems. QoS categories are little more than a convenience grouping mechanism, which has no semantic significance, so we do not support this in our approach.

Measurement-based approaches can be categorised into two groups:

1. *Predicate-based approaches*: These approaches use measurements to formulate constraints on the system behaviour. A system either fulfils these constraints or it does not fulfil them, so the underlying semantics is very similar to that of functional specifications: For each system

www.manaraa.com

we can decide whether it is a correct implementation of the specification, but over and above that we cannot compare different implementations.

2. *Optimisation-based approaches*: These approaches deviate from predicate-based approaches in viewing the achievement of non-functional properties (typically called *quality* in this context) as an optimisation problem. For each system we can still analyse whether it is a correct implementation of a specification, but in addition, we can compare two systems A and B, and, for example, state that A *is a better implementation than* B. Such statements are of course only valid in relation to some *objective function*. Objective functions are typically given as *utility functions* (or *value functions*, *cf.* [26] for an overview) representing users' or clients' preferences on different quality combinations.

The work presented here falls into the first category, although we believe it is general enough to be extended to support optimisation-based approaches, too.

Another interesting distinction is based on the degree of formality with which the measurements can be defined in the various approaches. We can distinguish two major cases: A first group of approaches defines measurements as functions of some domain without providing a semantic framework relative to which the meaning of each measurement could be formally defined. We say that these approaches have a *weak semantics,* because measurements are barely more than names for values. The second group of approaches provides a semantic framework—albeit the degree of formality may vary between approaches—and, thus, allows specifiers to define the meaning of measurements formally and precisely. We say that these approaches have a *strong semantics.* It is one of the aims of our approach to provide a framework for the formal definition of measurements, thus, it is an approach with strong semantics.

We will now review some predicate- and some optimisation-based approaches. For each approach we will also indicate whether it has a strong or weak semantics.

Predicate-based approaches One of the earliest works that proposes a measurement-based specification language for non-functional properties of component-based systems has been written by Xavier Franch [28]. It proposes a language called *NoFun*. The main concept of this language is the *non-functional attribute.* Franch distinguishes basic attributes and derived attributes. While derived attributes are formally specified in terms of other (basic or derived) attributes, basic attributes are not formally specified. They remain names for values, their semantics can only be expressed outside NoFun. Franch's approach, therefore, is an approach with weak semantics. Nonetheless it already contains many of the concepts found in modern predicate-based approaches.

Abadi and Lamport [3] present an approach which integrates time as a flexible variable into temporal-logic specifications. Although this approach is limited to the expression of timeliness properties, we classify it as a measurement-based approach, because—for example in contrast to [46] discussed above—the individual measurements are explicitly modelled as part of the specification (using normal flexible variables of the specification language), and are thus first-class citizens. Also, the approach can be extended to arbitrary measurements, which is what we have done in this article. Abadi and Lamport use standard temporal logic as their formal framework in which they also define their measurements. We can, therefore, classify them as an approach with a strong semantics.

In his thesis, Aagedal defines component quality modelling language (CQML) [1], a specification language for non-functional properties of component-based systems. The definition remains largely at the syntactic level, semantic concepts are mainly explained in plain English without formal foundations. The language is based on the definitions given in [38]. Arbitrary measurements can be defined as `quality_characteristics`, which have a `domain` and a semantics given in a `values` clause. The approach has a strong semantics by our definition of the term, even though the degree of formality of the semantic framework is comparatively low. We have proposed a more explicit representation of the semantic framework in previous work [63,64] which eventually led to the concept of context models presented in this article.

The UML has developed into a well-accepted language for specifying software systems. Consequently, several researchers have investigated using UML to model measurements and non-functional properties of software. Most important among these approaches is probably the UML SPT profile [55], which is based on ideas previously presented by Selic [69]. This standard profile defines a meta-model for the specification of performance- and scheduling-related parameters in UML models. Although it is comparatively flexible, and not specific to one characteristic, it does not consider issues related to CBSE, such as independent development of components and applications, or runtime management of resource allocation and component usage by component runtime environments. Another interesting approach has been chosen by Skene et al. [70]. They present SLAng a language for precisely specifying service-level agreements (SLAs). Their work is based on the precise UML (pUML) definition of the semantics of UML [21]. There, UML-like (meta-)models are used to specify both the syntax and the semantics of a modelling language. SLAng leverages the flexibility inherent to such a meta-modelling approach to allow specifiers to define measurements of their own, complete with a tailor-made semantic domain and semantic mapping. Because it uses UML as its semantic framework, it has

المنارة للاستشارات
www.manaraa.com

a strong semantics. Because the semantics of UML itself is not formally defined, the degree of formality of SLAng definitions remains very low.

Optimisation-based approaches Liu et al. [48] present a task-based model to describe QoS properties of applications. The tasks are considered to be so-called *flexible tasks* that "[…] can trade the amounts of time and resources [they] require to produce [their] results for the quality of the results [they] produce." Each task is described by a *reward profile,* which relates the quality of incoming data, the quality of data produced, and the amount of resources used while processing. Resource demand is considered only where it can be adjusted during execution. The model is completely oriented towards adaptation, admission control is not considered. In contrast, we defined the notion of a *feasible system* which captures admission control. If tasks are composed to form applications, they interact in a producer–consumer pattern. Consumers formulate their expectations on quality of incoming data using *value functions*—that is, objective functions over relevant quality measurements. A QoS management system then strives to allocate resources to tasks such that the value functions of corresponding consumers are maximised. [48] uses a weak semantics of measurements.

Sabata et al. [65] also present a task-based model. System specifications are composed from *metrics* and *policies,* and are written from three perspectives:

1. *Application perspective*: In this perspective one specifies the properties of one application without considering other applications, which might contend for the same resources. The specification uses *metrics,* which are essentially measurement definitions, and *benefit functions*—objective functions used to formulate constraints over *metrics.*
2. *Resource perspective*: This perspective serves to determine the total resource demand for each individual resource.
3. *System perspective*: In this perspective one specifies how resource conflicts between different applications can be resolved.

Again, the approach uses a weak semantics. However, the authors provide a classification of different types of *metrics,* so that some additional information about the semantics of a measurement can be derived from its placement in this classification.

In his dissertation [45], Lee presents another approach to modelling non-functional properties of applications and systems as an optimisation problem. In contrast to the two approaches described before, this approach does not consider the internal structure of applications, but is only concerned with balancing the resource allocation to applications contending for shared resources. The approach also features a weak semantics, defining measurements (called *quality dimensions*) as name–value pairs. For each measurement, the author defines an ordering relationship over the value domain. Resource demand and resource allocations are also simplified to name–value pairs. For each application Lee defines a *resource profile* as a relationship between allocated resources and delivered quality. The quality specification of an application is given by a *task profile,* the main part of which is a *utility function* representing the desired quality to be produced by this application. These utility functions are then combined in a weighted sum to form the *system utility.* The system utility is the global objective function to be maximised by allocating resources to applications. Lee has developed several algorithms to solve such optimisation problems efficiently and with sufficient accuracy. The notions of task and system utility are very close to our notion of extrinsic specifications, which makes Lee's approach an interesting candidate for integration with our approach. This would allow our approach to be extended to support optimisation-based techniques, as well as providing a strong semantics to Lee's approach.

5.3 Related projects

In this section, we review selected research projects working in related areas.

Quite a few projects consider the realisation of component infrastructure supporting non-functional properties. The Quality of Service-Aware Component Architecture (QuA) project [6] at SIMULA in Oslo aims to build a component runtime environment supporting *platform-managed QoS.* In their view, components have a functional specification (called the *QuA type*), an implementation, and a non-functional specification using the error functions introduced in [73]. Clients request a service by specifying a QuA type and some non-functional requirements. Based on this information the runtime environment selects components and QoS-management algorithms to instantiate an application providing the requested service. Our approach is more focused on specification and could be useful as a complimentary technique to the platform realisation in QuA.

In the context of the ADAPTIVE Communication Environment (ACE) project, the distributed object computing group at Washington University, St. Louis (Missouri, USA) have built The ACE ORB (TAO) [68], an efficient Object Request Broker (ORB) with support for real-time guarantees implementing Real-Time CORBA [54,67]. In another project based on this work—Component Integrated ACE ORB (CIAO) [19]—an implementation of CCM providing support for guaranteeing QoS and real-time properties has been built.

The components with quantitative properties and adaptation (COMQUAD) [34] project aimed at developing speci-

المنارة للاستشارات
www.manaraa.com

fication techniques, runtime support, and development technology for component-based applications with special focus on non-functional properties. The project developed a container architecture, CQML$^+$ an extension of Jan Aagedal's CQML [1] specification language, an integration of this language into UML, and a first component-oriented development process.

The Metropolis project [7, and references therein] defines an integrated development approach for computer systems based on formal methods. The project uses a meta-model based on the concepts *process, port,* and *medium.* So-called *quantities* and *quantity managers* are used to represent non-functional properties of a system. System descriptions can be structured into layers—called platforms in Metropolis. Each layer consists of a network of interacting processes and media, and can make use of services provided by the underlying platform. Many of the concepts are related to work presented in this article. However, Metropolis does not directly support CBSE. Also, there is only one meta-model which must be used by all specifications following the Metropolis approach. This means that only those properties which can be represented using this meta-model (mainly performance properties) can be used. In contrast, in this article we allow designers to freely define context models.

5.4 Summary

In this section, we have reviewed literature from two main research areas: CBSE and the theory of non-functional properties of software systems. We demonstrated that while research in the area of CBSE has advanced quite far to this day, the theory of non-functional properties still leaves many open questions. In particular, in the area of measurement-based approaches to the specification of non-functional properties of software systems (Sect. 5.2.3), a commonly accepted approach to the semantics of such specifications is missing. With our work, we aim to close this gap by providing a semantic framework for the specification of non-functional properties of component-based software.

6 Conclusions and outlook

In this article, we have discussed the specification of non-functional properties of components and component-based systems. We have provided a number of contributions to the field:

1. We have presented a formal semantic framework for the specification of non-functional properties of component-based software. The approach enables partial specifications of components, resources and containers to be independently developed. Later, these specifications can be composed to a system specification, which, then, allows reasoning about properties of the overall system. In particular, we can prove *feasibility* of a system implementation; that is, we can show that the components available, the resources provided and the container used combine into a system that fulfils certain non-functional requirements as given by a separate system specification. The semantic framework has been encoded in TLA$^+$, allowing for a large number of product properties that can be expressed as functions from a system state to some value domain to be used in specifications. For example, we can model performance properties, such as execution time, response time, or throughput, but also data-quality properties such as accuracy or precision, and security properties, such as propagation of confidential knowledge. The approach is generic; that is, it does not come with a pre-defined set of properties that can be handled, but allows properties to be defined as required.
2. On top of this semantic framework, we have developed and presented QML/CS, a new specification language for non-functional properties of component-based systems. The semantics of QML/CS has been defined based on our semantic framework, as shown in Sect. 4.1.2. To the best of our knowledge this is the first generic and formally founded specification language for non-functional properties of components and component-based systems.
3. We have shown how our semantic framework can be used to formalise the interface of analysis techniques for non-functional properties. This can be useful especially in the context of MDA tool components. Analysis techniques can, thus, be packaged as separate modules and reused in software development tools.

A major driver in the development of our semantic framework has been the separation of the concerns of the various parties playing roles in a component market. Hence, we separated so-called intrinsic properties, that a component developer controls completely, from so-called extrinsic properties, that depend on the context of use of a component. This separation will need to be driven even further. We need to distinguish two types of context of use:

1. *The specific hardware on which the component will be executed*: This includes processor architecture, memory protocol, floating point unit precision, and similar issues.
2. *The platform which will execute the component and the usage profile*: This includes the container strategies and resource availability, but also the frequency of requests, risk level of the machine (is it inside a firewall, not connected to the Internet, freely accessible from anywhere on the Internet, and so on).

المنارة للاستشارات
www.manaraa.com

In this article, we have only discussed type 2. We assume that component developers know the machine for which they have developed their components and can, therefore, include machine-specific information in the intrinsic specification. This is not necessarily true, however. For example, the Intel family of personal computer processors share a common instruction set—which means that most components developed for execution on an Intel processor can be executed on any specific processor type—but have fundamentally different architectures, leading to fundamentally different performance. Where such effects become important, we need to introduce another distinction, which we call *machine-independent* versus *machine-dependent* specification. There has been some discussion of this topic in the literature, but much work still remains to be done.

Another important distinction concerns the data to be processed by a component or a system. Some non-functional properties (again, most notably performance) are heavily dependent on the data to be processed. We, therefore, need to distinguish: (a) *data-independent* specifications, which, similarly to intrinsic specifications, can be given by component developers and provide the information only depending on the component itself, with some hooks where information about the data to be processed should be placed, and (b) *data specifications* which describe relevant properties of the data to be processed and can be combined with the data-independent specification to give a complete system specification. Again, this issue remains for future work.

The approach we have presented allows specifying and reasoning about arbitrary product properties. However, it does not provide support for determining what properties a component developer should specify and how she can predict what properties will be required by component users. Decision support in this area forms another large and interesting area for future research.

Non-functional properties are different from functional ones in that they typically support a notion of *"satisfaction to degrees";* that is, a system can be said to fulfil a certain non-functional property *better than* some other system. This aspect of non-functional properties has not been captured by the approach presented in this article. In Sect. 5.2.3 we discussed so-called optimisation-based approaches supporting this aspect. It would be interesting to attempt to improve our approach by combining it with some optimisation-based approach.

Finally, for our specification techniques to be practically used, we need to provide tooling to support specification and reasoning about specifications. Such tooling can be based on existing tooling for reasoning about TLA$^+$ specifications. Tooling should include support for QML/CS (or a similar, further evolved language), CASE-tool support for hiding the distinction between application, context and computational models as briefly discussed in Sect. 3.2, and support for reasoning based on the specifications thus created.

Acknowledgments This article is a summary of work done for my dissertation. I am grateful to Prof. Heinrich Hußmann, who in his role as my supervisor spent many hours of discussion and gave many useful pointers and comments throughout the dissertation project. I wish to thank Simone Röttger, Katja Siegemund, and Florian Heidenreich for their helpful comments on an earlier version of the article. Last but not least I must thank the anonymous reviewers who did a really outstanding job investing lots of effort to push me into making this a readable article. Of course, any lengths and awkward formulations remain completely my own fault. This research has been partially funded by the German Research Council in the research project COMQUAD.

References

1. Aagedal, J.Ø.: Quality of Service Support in Development of Distributed Systems. PhD thesis, University of Oslo (2001)
2. Abadi, M., Lamport, L.: The existence of refinement mappings. Theor. Comput. Sci. **82**(2), 253–284 (1991)
3. Abadi, M., Lamport, L.: An old-fashioned recipe for real time. ACM ToPLaS **16**(5), 1543–1571 (1994)
4. Abadi, M., Lamport, L.: Conjoining specifications. ACM ToPLaS **17**(3), 507–534 (1995)
5. Aigner, R., Pohl, C., Pohlack, M., Zschaler, S.: Tailor-made containers: Modeling non-functional middleware service. In: Bruel, J.-M., Georg, G., Hussmann, H., Ober, I., Pohl, C., Whittle, J., Zschaler, S. (eds.) Workshop on Models for Non-functional Aspects of Component-Based Software (NfC'04) at UML conference 2004, September 2004. Technical Report TUD-FI04-12 Sept.2004 at Technische Universität Dresden
6. Amundsen, S., Lund, K., Eliassen, F., Staehli, R.: QuA: platform-managed QoS for component architectures. In: Proceedings of the Norwegian Informatics Conference (NIK), Stavanger, Norway. Tapir Akademisk Forlag, Trondheim, Norway, November (2004)
7. Balarin, F., Watanabe, Y., Hsieh, H., Lavagno, L., Passerone, C., Sangiovanni-Vincentelli, A.: Metropolis: an integrated electronic system design environment. IEEE Comput. **36**, 45–52 (2003)
8. Beugnard, A., Jézéquel, J.-M., Plouzeau, N., Watkins, D.: Making components contract aware. IEEE Comput. **32**(7), 38–45 (1999)
9. Bézivin, J., Gérard S., Muller P.-A., Rioux, L.: MDA components: Challenges and opportunities. In: Evans, A., Sammut, P., Willans, J.S. (eds.) Proceedings of First International Workshop Metamodelling for MDA, pp. 23–41, York (2003)
10. Börger, E.: The ASM refinement method. Formal Aspects Comput. **15**(2–3), 237–257 (2003)
11. Börger, E., Stärk, R.: Abstract State Machines–A Method for High-Level System Design and Analysis. Springer, Berlin (2003)
12. Broy, M., Deimel, A., Henn, J., Koskimies, K., Plášil, F., Pomberger, G., Pree, W., Stal, M., Szyperski, C.: What characterizes a (software) component?. Softw. Concepts Tools **19**(1), 49–56 (1998)
13. Bruel, J.-M. (ed.): Proceedings of First International Workshop on Quality of Service in Component-Based Software Engineering, Toulouse, France. Cépaduès-Éditions, June (2003)
14. Cheesman, J., Daniels, J.: UML Components: A Simple Process for Specifying Component-Based Software. Addison-Wesley, Reading (2001)
15. Chen, S., Gorton, I., Liu, A., Liu, Y.: Performance prediction of COTS component-based enterprise applications. In: Crnkovic, I., Schmidt, H., Stafford, J., Wallnau, K. (eds.) Proceedings of

المنارة للاستشارات
www.manaraa.com

Fifth ICSE Workshop on Component-Based Software Engineering (CBSE'2002): Benchmarks for Predictable Assembly, May (2002)
16. Chimaris, A., Papadopoulos, G.A.: Implementing QoS aware component-based applications. In: Meersman, R., Tari, Z. (eds.) On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, LNCS, vol. 3291, pp. 1173–1189. Springer, Agia Napa (2004)
17. Chung, J.-Y., Liu, J.W.S., Lin, K.-J.: Scheduling periodic jobs that allow imprecise results. IEEE Trans. Comput. **39**(9), 1156–1174 (1990)
18. Chung, L., Nixon, B.A., Yu, E., Mylopoulos, J.: Non-Functional Requirements in Software Engineering. The Kluwer international series in software engineering. Kluwer, Dordrecht (1999)
19. Ciao website: http://www.cs.wustl.edu/~schmidt/CIAO.html
20. Ciupke, O., Schmidt, R.: Components as context-independent units of software. In: WCOP 96, Special Issues in Object-Oriented Programming, Workshop Reader of the Tenth European Conference on Object-Oriented Programming ECOOP96, pp. 139–143. d.punkt.verlag, Heidelberg (1996)
21. Clark, T., Evans, A., Kent, S.: Engineering modelling languages: A precise meta-modelling approach. In: Kutsche, R.-D., Weber, H. (eds.) Proceedings of Fifth International Conference on Fundamental Approaches to Software Engineering (FASE 2002), LNCS, vol. 2306, pp. 159–173. Springer, Grenoble (2002)
22. Cottenier, T., van den Berg, A., Elrad, T.: Joinpoint inference from behavioral specification to implementation. In: Ernst, E. (ed.) 21st European Conference on Object-Oriented Programming (ECOOP'07), LNCS, vol. 4609, pp. 476–500. Springer, Berlin (2007)
23. Councill W.T., Heinemann, G.T.: Definition of a software component and its elements. In: Heinemann, G.T., Councill, W.T. (eds.) Component-Based Software Engineering—Putting the Pieces Together, pp. 5–20. Addison-Wesley, Reading (2001)
24. Crnkovic, I., Larsson, M., Preiss, O.: Concerning predictability in dependable component-based systems: Classification of quality attributes. In: de Lemos, R., et al. (eds.), Architecting Dependable Systems III, LNCS, vol. 3549, pp. 257–278. Springer, Berlin (2005)
25. Filman, R.E., Elrad, T., Clarke, S., Akşit, M.: Aspect-Oriented Software Development. Addison-Wesley, Reading (2004)
26. Fishburn, P.: Preference structures and their numerical representations. Theor. Comput. Sci. **217**(2), 359–383 (1999)
27. Ford, G.: Measurement theory for software engineers. In: Lecture Notes on Engineering Measurement for Software Engineers. Carnegie Mellon University. CMU/SEI report CMU/SEI-93-EM-9 (1993)
28. Franch, X.: Systematic formulation of non-functional characteristics of software. In: Proceedings of Third International Conference on Requirements Engineering, IEEE Computer Society, pp. 174–181 (1998)
29. Frølund, S., Koistinen, J.: Qml: A language for quality of service specification. Technical report, Hewlett-Packard Software Technology Laboratory. Technical Report No. HPL-98-10 (1998)
30. Gościński, A.: Distributed Operating Systems: The logical design. Addison-Wesley, Reading (1991)
31. Griffel, F.: Componentware. dpunkt.verlag, Heidelberg (1998)
32. Hamann, C.-J.: On the quantitative specification of jitter constrained periodic streams. In: Proceedings of Fifth International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'97). IEEE Computer Society (1997)
33. Hamann, C.-J., Zschaler, S.: Scheduling real-time components using jitter-constrained streams. In: Proceedings of Workshop on Advances in Quality of Service Management (AQuSerM'06) (2006)
34. Härtig, H., Zschaler, S., Pohlack, M., Aigner, R., Göbel, S., Pohl, C., Röttger, S.: Enforceable component-based realtime contracts—supporting realtime properties from software development to execution. Springer Real Time Syst. J. **35**(1), January (2007)
35. Hissam, S.A., Moreno, G.A., Stafford, J.A., Wallnau, K.C.: Packaging predictable assembly. In: Bishop, J. (ed.) Proceedings of IFIP/ACM Working Conference on Component Deployment (CD 2002), LNCS, vol. 2370, pp. 108–126. Springer, Berlin (2002)
36. Hofmeister, C., Nord, R.L., Soni, D.: Describing software architecture with UML. In: Donohoe, P. (ed.) Software Architecture. Proceedings of First Working IFIP Conference on Software Architecture (WICSA1), pp. 145–159 (1999)
37. Hu, T., Marcus, L.: Semantic foundations of an adaptive security infrastructure: Delegation. Unpublished (2005)
38. Information technology–quality of service: Framework. ISO/IEC 13236:1998, ITU-T X.641 (1998)
39. Jacobson, I., Griss, M., Jonsson, P.: Software Reuse–Architecture, Process and Organization for Business Success. ACM Press, London (1997)
40. Jones, C.B.: Specification and design of (parallel) programs. In: Manson, R.E.A. (ed.) Proceedings of IFIP '83, IFIP, pp. 321–332. North-Holland, Amsterdam (1983)
41. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (eds.) 11th European Conference on Object-Oriented Programming (ECOOP'97), LNCS, vol. 1241, pp. 220–242. Springer, Berlin (1997)
42. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley, Reading (2003)
43. Krüger, I.H.: Service specification with MSCs and roles. In: Proceedings of IASTED International Conference on Software Engineering (IASTED SE'04), IASTED, ACTA Press, Innsbruck (2004)
44. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, Reading (2002)
45. Lee, C.: On Quality of Service Management. PhD thesis, Carnegie Mellon University, August (1999)
46. Leue, S.: QoS specification based on SDL/MSC and temporal logic. In: Bochmann, G.v., de Meer, J., Vogel, A. (eds.) Workshop on Multimedia Applications and Quality of Service Verification, Montreal (1994)
47. Liu, J.W.S.: Real-Time Systems. Prentice-Hall, New Jersey (2000)
48. Liu, J.W.S., Nahrstedt, K., Hull, D., Chen, S., Li, B.: EPIQ QoS characterization. ARPA Report, Quorum Meeting, July (1997)
49. Lund, M.S., den Braber, F., Stølen, K.: A component-oriented approach to security risk analysis. In: Bruel [13], pp. 99–110
50. Malan, R., Bredemeyer, D.: Defining non-functional requirements. Bredemeyer Consulting, White Paper. http://www.bredemeyer.com/papers.htm, 2001
51. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Trans. Softw. Eng. **26**(1), 70–93 (2000)
52. Naumovich, G., Clarke, L.A.: Classifying properties: An alternative to the safety–liveness classification. In: Proceedings of the eighth ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 159–168. ACM Press, New York (2000)
53. Nothnagel, J.: Ressourcenverwaltung in drops. Diplomarbeit, Technische Universität Dresden, July 2002. In German
54. Object Management Group. Real-time CORBA joint revised submission. OMG Document, March 1999. URL http://cgi.omg.org/cgi-bin/doc?orbos/99-02-12 or http://cgi.omg.org/cgi-bin/doc?orbos/99-03-29

المنارة للاستشارات
www.manaraa.com

55. Object Management Group. UML profile for schedulability, performance, and time specification. OMG Document, March (2002). URL http://www.omg.org/cgi-bin/doc?ptc/02-03-02
56. Object Management Group. UML 2.0 OCL specification. OMG Document, October (2003). URL http://www.omg.org/cgi-bin/doc?ptc/03-10-14
57. Object Management Group. MDA guide version 1.0.1. OMG Document, June (2003). URL http://www.omg.org/cgi-bin/doc?omg/03-06-01
58. Object Management Group. Request for proposals: MDA tool component. OMG Document, July (2006) URL http://www.omg.org/cgi-bin/doc?ad/2006-06-09
59. Rajkumar, R., Lee, C., Lehoczky, J., Siewiorek, D.: Practical solutions for QoS-based resource allocation problems. In: Proceedings of IEEE Real-Time Systems Symposium, December (1998)
60. Reussner, R.H.: Parametrisierte Verträge zur Protokolladaption bei Software-Komponenten. Logos Verlag, Berlin (2001) (In German)
61. Reussner, R.H., Poernomo, I.H., Schmidt, H.W.: Reasoning about software architectures with contractually specified components. In: Cechich, A., Piattini, M., Vallecillo, A. (eds.) Component-Based Software Quality: Methods and Techniques, LNCS, vol. **2693**, pp. 287–325. Springer, Berlin (2003)
62. Röttger, S., Aigner, R.: Modeling of non-functional contracts in component-based systems using a layered architecture. In: Component Based Software Engineering and Modeling Non-functional Aspects (SIVOES-MONA), Workshop at UML, October 2002
63. Röttger, S., Zschaler, S.: CQML$^+$: Enhancements to CQML. In: Bruel [13], pp. 43–56
64. Röttger, S., Zschaler, S.: Tool support for refinement of non-functional specifications. Softw. Syst. Model. J. (SoSyM), **6**(2), June (2007)
65. Sabata, B., Chatterjee, S., Davis, M., Sydir, J.J., Lawrence, T.F.: Taxonomy for QoS specifications. In: Proceedings of Third International Workshop on Object-oriented Real-Time Dependable Systems (WORDS'97). Newport Beach, California (1997)
66. Salzmann, C., Schätz, B.: Service-based software specification. In: Proceedings of International Workshop on Test and Analysis of Component-Based Systems (TACOS) ETAPS 2003, Electronic Notes in Theoretical Computer Science. Elsevier, Warsaw (2003)
67. Schmidt, D.C., Kuhns, F.: An overview of the real-time CORBA specification. IEEE Comput. 56–63 (2000)
68. Schmidt, D.C., Levine, D.L., Mungee, S.: The design of the TAO real-time object request broker. Comput. Commun. **21**(4), April (1998)
69. Selic, B.: A generic framework for modeling resources with UML. IEEE Comput. **33**(6), 64–69 (2000)
70. Skene, J., Lamanna, D.D., Emmerich, W.: Precise service level agreements. In: Proceedings of 26th International Conference on Software Engineering (ICSE'04), pp. 179–188. IEEE Computer Society, Edinburgh (2004)
71. Smith, C.U., Williams, L.G.: Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. Object-Technology Series. Addison-Wesley, Reading (2002)
72. Sommerville, I.: Software Engineering. Addison-Wesley, Reading (1996)
73. Staehli, R., Eliassen, F., Aagedal, J.Ø., Blair, G.: Quality of service semantics for component-based systems. In: Middleware 2003 Companion, Second International Workshop on Reflective and Adaptive Middleware Systems (2003)
74. Stirling, C.: Modal and Temporal Properties of Processes. Texts in Computer Science. Springer, Berlin (2001)
75. Szyperski, C.: Component Software: Beyond Object-Oriented Programming, 2nd edn. Component Software Series. Addison-Wesley, Reading (2002)
76. Tanenbaum, A.S.: Modern Operating Systems, 2nd edn. Prentice-Hall, Englewood Cliffs (2002)
77. Völter, M.: A generative component infrastructure for embedded systems. In: Voelter, M., Kircher, M., Schwanninger, C., Zdun, U., Schmid, A. (eds.) Proceedings of Workshop on Reuse in Constrained Environments at OOPSLA'03, October (2003)
78. Völter, M.: Model-driven development of component infrastructures for embedded systems. In: Klein, T., Rumpe, B., Schätz, B. (eds.) Proceedings of Dagstuhl-Workshop Modellbasierte Entwicklung eingebetteter Systeme (MBEES 2005), Dagstuhl, Germany, 2005. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl. http://drops.dagstuhl.de/opus/volltexte/2005/31
79. Werner, M., Richling, J.: Komponierbarkeit nichtfunktionaler Eigenschaften–Versuch einer Definition. In: Fachtagung, G.I. Betriebssysteme 2002 Gesellschaft für Informatik, Berlin (2002) (In German)
80. Zhang, C., Jacobsen, H.-A.: Resolving feature convolution in middleware systems. In: Vlissides, J.M., Schmidt, D.C. (eds.) Proceedings of 19th Annual ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004), pp. 188–205. ACM, Vancouver
81. Zschaler, S.: A Semantic Framework for Non-functional Specifications of Component-Based Systems. PhD thesis, Technische Universität Dresden, Germany, April (2007)
82. Zschaler, S.: Example specifications of non-functional properties of a simple counter application. Technical Report COMP-006-2008, Computing Department, Lancaster University (2008)

Author Biography

Dr. Steffen Zschaler received his Diploma in Computer Science from Technische Universität Dresden in Germany in 2002 and his Doctorate in 2007. He is currently a Senior Research Associate with Lancaster University, UK. His interests are in Software Engineering, esp. in modularisation, modelling, and non-functional properties.

المنارة للاستشارات
www.manaraa.com

Copyright of Software & Systems Modeling is the property of Springer Science & Business Media B.V. and its content may not be copied or emailed to multiple sites or posted to a listserv without the copyright holder's express written permission. However, users may print, download, or email articles for individual use.

للاستشارات المنارة

www.manaraa.com