Resource-aware life cycle models for service-oriented applications managed by a component framework

Technische Universiteit Eindhoven
Department of Mathematics and Computer Science

Resource-aware Life Cycle Models for Service-oriented Applications managed by a Component Framework

R.H. Mak

13/07

ISSN 0926-4515

Computer Science Reports 13-07
Eindhoven, July 2013

Resource-aware Life Cycle Models for Service-oriented Applications managed by a Component Framework

Rudolf H. Mak

Department of Mathematics and Computer Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
`E-mail R.H.Mak@tue.nl`

Abstract

In this report we present a series of formal models that describe dynamically reconfigurable applications at various stages of their life cycle. It is our intention that these models capture the essential concepts of such applications and the platforms on which they are deployed, and that they indicate the essential activities required to accomplish an application's transition from one stage of its life cycle to the next. These models aim to support a life cycle in which applications are designed as a combination of services and realized by predefined components that are deployed in a framework specially tailored to the resource management needs of these applications.

1 Introduction

Many current day large scale applications exhibit the following characteristics. They are *interdisciplinary*: Their design relies on knowledge from a wide range of domains of expertise, that is difficult to cover by a single design team. They are *intrinsically distributed*: Control of the application occurs at a location that is geographically distant from sensors and/or actuators. Typical examples are surveillance systems for public areas such as shopping malls, airports, or even parts of a city, or control systems for chemical and nuclear plants. Similarly, processing of data may occur at locations remote from where the data is acquired and/or stored. Typical examples are large scale scientific computations that require computing power to an extent not available to a single user. Also, the variety of resources needed by the system may not be present at a single location or machine. They require *intricate resource management*: On the one hand, intelligent reaction to sensor input or data content may give rise to large fluctuations in resource demands of the application itself. On the other hand, the application may be operating in an environment where it has to compete for resources with other applications that execute simultaneously, and hence may be confronted with fluctuating resource availability.

These characteristics have consequences for the design and deployment of such applications. First of all, the interdisciplinary nature favors a design style, where applications are built from predefined components developed by experts in the various disciplines. Not only do these components encapsulate specialist domain knowledge, they also facilitate reuse and are mechanisms for protection and commercial exploitation of intellectual property.

For reuse of independently developed components to be successful, it must be complemented with reuse of architectural principles that facilitate interoperability between such components. For this a variety of component frameworks has been developed both in industry (EJB [8], COM [4], CCM [15]) and in academia (Koala [28], SOFA [7], Kobra [1], PECOS [14], PECT [16], Fractal [6]). Most of these frameworks, however, are not resource-aware and do not accommodate resource-dependent dynamic reconfiguration. Indeed, development of frameworks that exhibit the latter capability has been identified as a major research challenge in service-oriented computing [23].

In the literature many definitions of what a framework, or more specific a component framework, encompasses can be found [26, 13, 17]. In all cases a framework is defined as a software entity that realizes the goal stated above, i.e., a software platform that embodies the reuse of architectural principles for system composition within an application domain. The specification of precisely how the software achieves its goal is not perceived as being part of the framework, although tool support, in particular for component wrapping may reveal some of this information.

Therefore, we prefer a more extended notion of framework [20], that supplements the description of its software with additional concepts that are not only important for the proper deployment of the framework in applications, but also for the design of the framework itself. We refer to this model as the *MRTV framework model*, since it defines a framework as a set of Models and methods, a Run-time system, a set of Tools, and a set of Views. In this definition the run-time environment plays the role of the software platform of the previous definitions. Although each framework defines its own particular set of models, this set, in general, includes life cycle models — both for components and applications —, programming models, data models, process models, etc. Tooling, both for development of components and applications and platform management is considered an intrinsic part of the framework. Finally, and perhaps most importantly, the framework contains a set of views which explain relationships between its models, methods, tools and run-time system. Each view highlights a different aspect of the framework usage. Together, these views build a coherent picture of the framework that allows not only development of components suited for the framework, and the execution of applications built from such components on the its runtime platform, but also the development of its tools and its runtime platform. Again each framework defines its own particular set of views, but to get some idea of what is expected here one can think of views as defined by Kruchten [19].

Our research, is aimed at the design of a resource-aware component framework that facilitates third-party runtime composition of applications with a service-oriented architecture and with predictable resource usage. An initial discussion of that framework has been presented in [10, 22]. A similar framework targeting the domain of video surveillance applications can be found in [5].

This report presents a variety of conceptual models of a mathematical nature, both for the framework entities that constitute the fabric of our applications — interfaces, services, and components — and for the framework entities that constitute the platform on which they are deployed — hosts and networks. We target dynamically reconfigurable applications, so we model these entities at various phases of their life cycle, which gives rise to a hierarchy of models for each entity. Furthermore, we use the models to identify and explain the responsibilities of the framework entities that manage the activities that take place in the various phases of an application's life cycle. Thus, the models also constitute abstract versions of the data structures maintained by these entities to perform their management tasks.

The rest of this report is organized as follows. Section 2 is devoted to the application life cycle. It discusses in some detail the three major phases of that life cycle. Section 3 is devoted to the component-based framework we envision. It introduces its principle entities and outlines its organization. Sections 5 up to 8 are devoted to models. These models are developed according to four views, an initial one that emphasizes resource management, followed by three more views, each of which is associated with a phase from the application life cycle. These sections are preceded by Section 4 which contains some general modeling principles and introduces some mathematical notation. For the less mathematically inclined readers, a software engineering oriented version of the models, in the form of UML diagrams, is given in the appendices. Finally, in Sections 9 and 10, we discuss related work, summarize results and indicate directions for further research.

2 Application life cycle

In this section we describe a life cycle for dynamically reconfigurable applications. This life cycle is displayed in Figure 1 and distinguishes the three main phases in which an application can be engaged: (re)design, (re)deployment and operation.

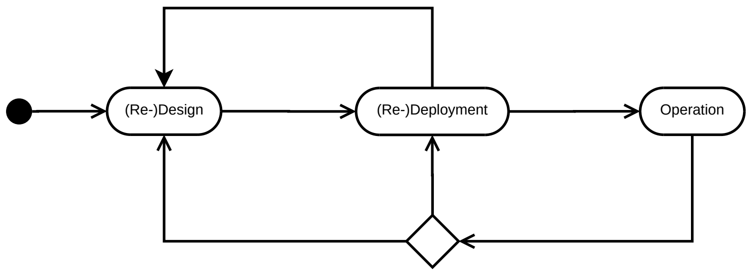


Figure 1: Application life cycle

Before we explain the activities that take place in the individual phases in more detail, we make some general observations. The most important phase of the life cycle is the operation phase. In this phase the application performs the services for which it has been designed. The other two phases represent deviations from this "normal" behavior during which the application is reconfigured. Reconfiguration is triggered by events that can be both external and internal to the application and that occur for many reasons. In this report, we concentrate on reconfiguration needed to resolve resource conflicts. We consider five reconfiguration mechanisms, three of which are so far-reaching that they require an application to temporarily leave its operation phase and enter a dedicated reconfiguration phase, viz., a redeployment or redesign phase.

To understand why there are two distinct phases associated with reconfiguration, we first need to explain our approach to application design, which is commonly known as *third-party composition.* We are concerned with service-oriented applications, and consider their design as a form of architectural design which consists of composition of predefined units, called services, of well-defined functionality to be delivered at a specified level of quality. How individual services are designed, however, is of no concern to us. Instead, we simply assume the existence of predefined executable software units, called components[1], that deliver the desired services upon deployment. Note that this is a deviation from the classical definition of a component given by Szyperski [26], where a component is both the unit of composition (as its name suggests) and the unit of deployment. We retain the latter, but consider services as the more fundamental unit of composition. By ignoring the service design aspect and concentrating on service composition, we obtain an application life cycle in which application design becomes a recurring activity, rather than an activity that occurs once at the start of the life cycle, (with the possible exception of some redesign as part of maintenance). Of course, any application must be designed and deployed before it comes into operation for the first time. Apart from the amount of effort involved, such initial phases are not different from the ones needed for reconfiguration of an already operational application. Hence, the single arrow indicating the entry point of the life cycle in Figure 1 suffices to capture initial design and deployment.

The two reconfiguration phases differ in that redeployment does not alter the structure of an application, but merely changes the location where components are deployed, whereas redesign also modifies the structure of the application, by changing either the service composition or the selection of components that provide the services.

Design phase

The main activities of the design phase are recorded in the UML activity diagram depicted in Figure 2. We distinguish: application architecture by service composition, allocation of components to services, definition of reconfiguration strategies and definition of deployment constraints.

[1]Actually component types, but in this section we will not yet make that distinction

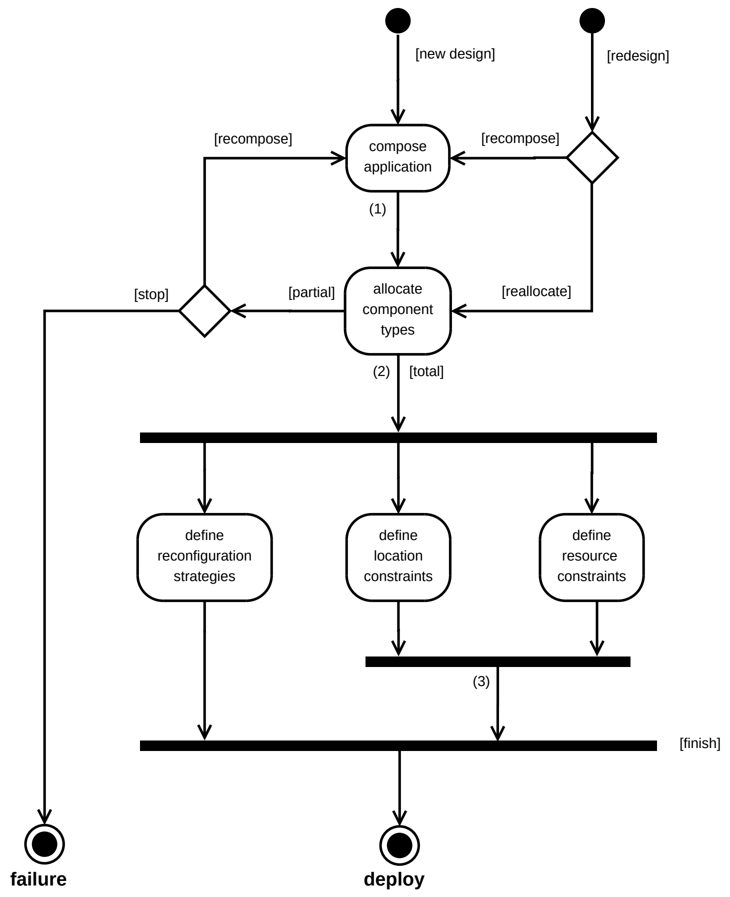


Figure 2: Design phase activities. The arrows labeled with a number have an associated model in Section 6. These models define the outcome of their immediately preceding activity and are given by: $\mathfrak{M}^{design}_{sb\_app\_design}$ for (1), $\mathfrak{M}^{design}_{cb\_app\_design}$ for (2), and $\mathfrak{M}^{design}_{constraint}$ for (3).

As explained above, design is architectural design and the architecture of an application consists of two structures, a service-based structure and a component-based structure. These structures need not be the same. Finding a component-based structure that is compatible with the service-based structure is a non-trivial activity and is, therefore, a separate activity of the design phase.

Deployment constraints come in two kinds. The first kind is concerned with the location of components. In some applications, e.g. surveillance applications, services may be required at specific geographic locations, for instance, the recording of images by a particular camera. Other constraints may not be location specific, but may require that a certain combination of services needs to be present on a single node (co-allocation constraints).

The second kind of deployment constraints is concerned with Quality of Service (QoS), which is specified as part of a service's contract. Services, however, are provided by components and the latter are assumed to be deployable in only a fixed number of modes. For each of these modes, it should, in principle, be possible to determine the QoS-level of every service offered by the component, when operating in that mode. Vice-versa, it should therefore also be possible to determine the set of modes at which a component can be operated such that all its services meet their contract. Whenever this set contains two or more modes the following situation may arise. Mode $M1$ provides service $S1$ with barely sufficient quality, but provides service $S2$ at excellent quality, whereas mode $M2$ does precisely opposite. In this case, it is conceivable, that although both modes meet the requirements stated in the service contracts, the application's end-user has a strong preference for one of them. In Figure 2, the stage of the life cycle design phase in which this preference is stated is indicated by the "define resource constraints"-activity, since each mode of deployment of component has a well-defined associated resource demand.

There is one more design phase activity, which is the definition of reconfiguration strategies. Although important, because they supply the criteria used in a number of decision boxes ("$\diamondsuit$"-boxes) in the activity diagrams, reconfiguration strategies are not part of any application model. Therefore, modeling this activity is outside the scope of this report.

Deployment phase

Given the outcome of the design phase, i.e., a component-based application design and a set of deployment constraints, the goal of the deployment phase is to create a situation in which this design is deployed on a component-based platform. This means that on a subset of the platform nodes, called hosts, a collection of processes must be created, each of which is a component of the designed application. Of course, hosts must be allocated to these components in accordance with the location constraints. Moreover, each component must be deployed in a resource mode in accordance with the resource constraints, and sufficient resources must be reserved on the hosts to allow components to execute in their assigned resource modes. Finally, network connections must be established to accommodate data traffic between components, as indicated by interface bindings.

To achieve this goal the deployment phase is divided in three activities that are depicted in the activity diagram in Figure 3.

The first activity is the determination of a feasible allocation, which consists of three parts, viz., an allocation of hosts to components, an assignment of resource modes to components, and, for each component, a reservation of a resource budget on its allocated host, in agreement with its resource mode. In general, many feasible allocations will exist, from which one has to be selected. Nevertheless, it is also possible that the deployment constraints are so tight that no feasible allocation exists. In that case, redesign is the only option. Although we have not investigated the computational complexity of this combinatorial problem, we expect it to be NP-complete.

Once a feasible allocation has been obtained, the application has to be instantiated accordingly. For each component executable code, if not already present, has to be installed on its allocated host, and a process has to be

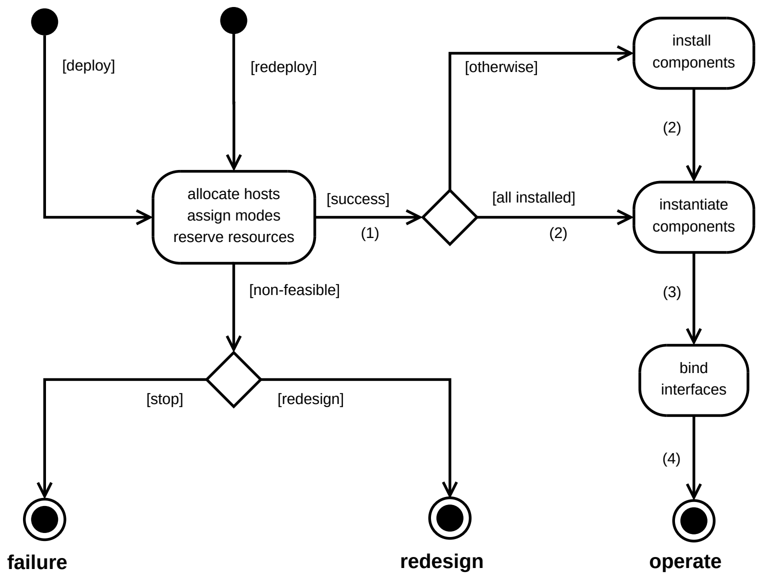


Figure 3: Deployment phase activities. At the arrows (1), (2), (3) and (4) the application complies with the models $\mathfrak{M}^{deployment}_{feasible\_scheme}$, $\mathfrak{M}^{deployment}_{installed\_app}$, $\mathfrak{M}^{deployment}_{instantiated\_app}$ and $\mathfrak{M}^{deployment}_{bonded\_app}$ respectively. These models are defined in Section 7.

created that runs that code in the required resource mode. Moreover, the resources budgets specified in the reservations have to be made available to those processes. This is the second activity of the deployment phase.

In the third and final activity of the deployment phase, network connections for the interface bindings have to be established. The reason why this activity is separate from the previous one is that they are performed by different parties. Component installment and instantiation is done solely by framework management entities (see Section 3), whereas the instantiated components must assist in setting up the network connections necessary to bind their interfaces.

Operation phase

The main activities of the operation phase are depicted in Figure 4. The first activity of the operation phase is to activate the application. This involves bringing all components in a state in which they are ready to execute. Amongst other things, this means allocating to each component the resources it needs. This activity is guaranteed to succeed, since these resources have been reserved in advance during the deployment phase.

After activation, an application enters a stage in which it is executing as specified until an error occurs in one of its components. During execution, the behavior of an application can be influenced by external commands issued by an operator. For instance, in a surveillance application an operator might

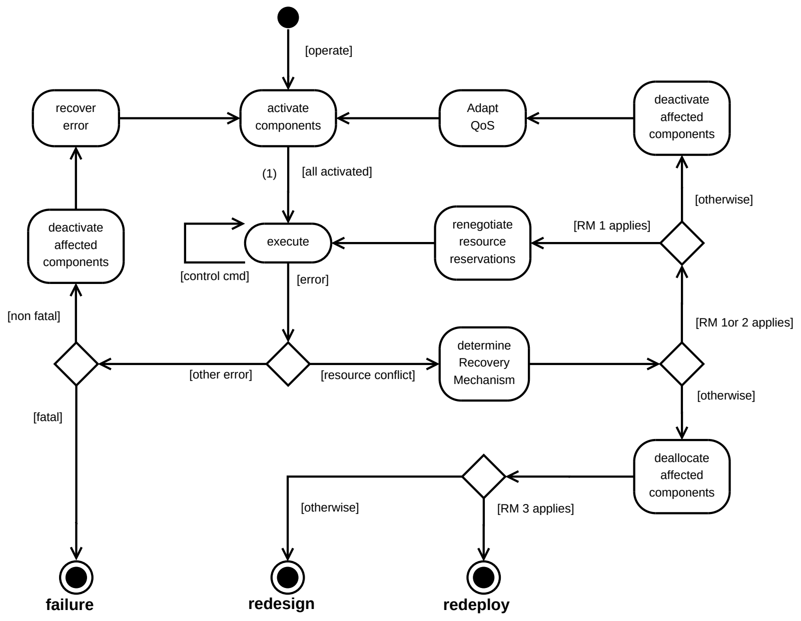


Figure 4: Operation phase activities. At arrow (1) the application complies with model $\mathfrak{M}^{operation}_{runtime}$ described in Section 8. See the text for an explanation of the recovery mechanism (RM) labels on outgoing edges of some of the decision boxes.

issue PTZ[2]-commands to a camera [5]. In general, any application containing actuators can be expected to require control commands. Similarly, commands that influence the data processing of an application can be issued. Again taking a surveillance application as an example, an operator might issue a request to track a person appearing on one of the camera images.

For any application, a large variety of errors may occur, some of which are application specific, whereas others are more general. Handling of application specific errors can not be part of any generic life cycle and is therefore outside the scope of this report. Of the more general errors, this report only addresses the handling of resource errors. Non-resource errors, whether general or application specific, are classified as either fatal or non-fatal. How this classification is made and how non-fatal errors are resolved is left unspecified. However, it is assumed that components affected by non-fatal errors must be deactivated — if this did not already occur as a side-effect of the error— before resolution can take place. The associated activities are depicted in the leftmost column of Figure 4.

A resource error is the manifestation of a *resource conflict*, i.e., a situation in which there is a mismatch between the resources used by a component and the resources reserved for it on its host. A resource error can be either a signal that this situation is currently present, or a prediction that such a situation will arise in the near future. Resource conflicts come in two flavors. When an application's resource usage exceeds its resource reservation, we speak of

[2]Pan Tilt Zoom

resource shortage. When, on the other hand, an application's resource reservation exceeds its resource usage, we speak of *reservation surplus.* In case of resource shortage an application is in trouble, but in case of reservation surplus the application is in good shape. Nevertheless, resolving reservation surplus is useful, because it can be beneficial for other applications that run on the same platform.

Resource conflicts are analyzed to determine which *recovery mechanisms* are applicable. Listed in order of increasing impact, we distinguish five recovery mechanisms:

RM 1 renegotiation of local resource reservations,

RM 2 QoS adaption,

RM 3 reallocation of components,

RM 4 reallocation of services,

RM 5 recomposition.

In case of resource shortage, renegotiating local resource reservations is an option, provided there are still free resources available on the host where the conflict occurs. In case of reservation surplus, renegotiation is always possible. Especially, when the resource error is not based on the current situation, but is a prognosis of a future situation, renegotiation is an attractive option because it may be possible to complete the renegotiation process, and have the new reservations in place, before the actual conflict arises. Thus, deactivation of components can be avoided. In fact, since the application is not involved in the resolution mechanism, it will remain oblivious to the occurrence of the conflict.

Another possibility to resolve a resource conflict without interrupting the operation phase is by QoS adaptation. This is achieved by changing the resource modes in which components operate. This mechanism, however, requires deactivation of the components that are in conflict. Moreover, adapting the quality of a service may affect components other than the ones in conflict and located on different nodes.

Another mechanism is to resolve resource conflicts is to relocate components that are in conflict to platform nodes where a better match can be obtained. In fact, in order to resolve a conflict in one application it may even be necessary to relocate components belonging to an other application that by itself suffers no conflict. Since allocation of components is part of the deployment phase, this mechanism interrupts the operation phase.

Finally, it may even be necessary to reallocate services or alter the logical structure of an application to resolve a resource conflict. In the former case, one or more components are replaced by others that provide the same services but under more suitable resource demands. Typical examples of the latter case can be found in patterns that create concurrency, such as replication of services to divide the workload between the copies, or replacing a service by a sequential composition of services and arranging the associated components in a pipeline to increase throughput. The activities associated with the handling of resource conflicts are displayed in the rightmost two columns of Figure 4.

Since, as described above, a resource conflict can be resolved by several resolution mechanisms, we need criteria to decide which mechanism to apply. For each application, these criteria have been defined in its design phase in the form of reconfiguration strategies. They are applied in the decision boxes that occur in Figure 2 and in the rightmost two columns of Figure 4.

3 Component framework

In this section we describe the structure of a resource-aware component-based framework for the design and deployment and operation of service-oriented applications. The main entities of the framework are depicted in Figure 5.

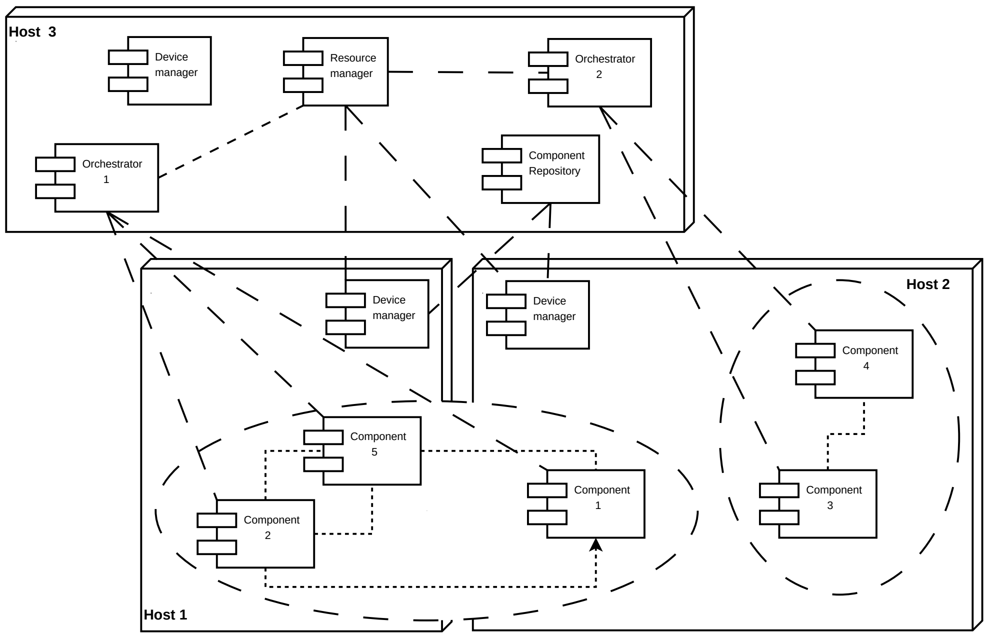


Figure 5: Framework entities

We distinguish three categories of entities. The first category consists of the entities that constitute the fabric of our applications. These are the components and interface bindings. Although the framework supports service-oriented applications, services do not appear as framework entities. They are considered logical concepts that describe functionality and performance aspects of an application, but it are the components, being containers of services, that realize these aspects. Indeed, we assume that components are the only means to obtain a service. In the diagram in Figure 5, applications are indicated by dashed ovals, and interface bindings between the components of an application are indicated by dotted zigzag lines.

The second category consists of the entities that provide the platform on which applications are deployed. These are the hosts and the networks. The diagram shows a very small platform consisting of three hosts, only two of which carry applications. The networks are left implicit in the diagram. Their existence, can be derived from the lines between framework entities, such as the interface bindings, which imply exchange of data. Although, in most practical case there will be a single network, our models will accommodate the situation in which several networks coexist.

The third category consists of management and storage entities. For each application there exists a dedicated framework entity, a so-called *orchestrator*, whose responsibility it is to manage that application. For each host there exists a dedicated framework entity, called a *device manager*, whose responsibility it is to manage all framework entities that reside on its host. Furthermore, there is a framework unique entity, called the *resource manager*, whose responsibility it is to manage the collection of all resources supplied by the entire platform. In the next sections, the responsibilities of the various management entities will be further detailed. Finally, there is a framework unique entity, called the *repository*, that contains all component descriptions. Every component of any application must be derived from a component description in the repository. So, the repository effectively defines the universe of components from which all applications are constructed. In the diagram all of these entities, with the notable exception of device managers, are located on a separate host that does not host any user application. The latter, however, is merely done for clarity of explanation, but is not a fundamental requirement of the framework.

As an aside, we remark that the framework entities of the third category can themselves be seen as components, and their combined management effort as yet another application. In that case, an additional orchestrator, called the *framework manager*, should be added to the diagram.

4 Modeling preliminaries

In this section we present some general principles about the models presented in this report and explain some mathematical notations. These models describe only application entities and platform entities. The management and storage entities of the framework are not modeled. In fact, the management entities of the framework are the ones that maintain models of the other entities. In particular, an orchestrator maintains application related models and the resource manager and device managers maintain platform related models.

Most models in this report, fall in one of two categories: *type models* and *instance models*. E.g, the component descriptions that can be found in the framework repository are component types, albeit very elaborate ones. Component descriptions should contain at least an executable program (or the sources for obtaining one), but they can be far more elaborate than that [21]. Of course, the component type models presented in this report are a huge abstraction of such descriptions. The components themselves, on the other hand, are processes that execute these programs and their models classify as instance models. Whereas type models are static, instance models are dynamic. They are snapshots that describe a framework entity at a certain moment in time and are therefore subject to updates during the life time of such an entity.

Furthermore, each model describes an entity from a particular point of view. These views correspond to the phases of the application life cycle described in Section 2. However, not every framework entity has a model for every phase. Since the models closely follow the life cycle, they can be organized in a hierarchy, where models of a later phase are extensions of those of an earlier phase. Besides the views that correspond to a phase of the life cycle, there is one additional view, the resource view. In line with our emphasis on resource-awareness, this is the most basic view. It defines models that lie at the bottom of the hierarchies.

Finally, for precision's sake, all models are expressed using formal mathematics. For the less mathematical oriented reader, some of the notational conventions that are used are explained below, as are notations specific to this report. In addition, Appendix A contains a collection of UML diagrams which present the same models from a software engineering perspective.

Notation

From a mathematical point of view all framework entities are tuples. These tuples are denoted as a list of attributes (fields) enclosed by angular brackets "$\langle$" and "$\rangle$". To refer to an attribute of a named tuple, we use the infix selection operator ".". For instance, we write $T.b$ to refer to the second attribute of quadruple $T = \langle a, b, c, d\rangle$[3]. Besides attribute selection we also use tuple slicing, i.e., we write $T.\langle a, d\rangle$ to refer to the slice that consists of the first and last attribute of T. Hierarchical models give rise to nested tuples. So, several applications of the selection operator may be needed to select an attribute of a nested tuple. For instance, if the first attribute of T is itself a nested tuple, i.e., $a = \langle e, f, g\rangle$ and $f = \langle k, l, m\rangle$, then we must write $((T.a).f).m$ to refer to the third attribute of the second attribute of the first attribute of T. When the level of nesting increases this notation quickly becomes difficult to digest. Therefore, we ensure that all attribute names of nested tuples are distinct, and adopt the convention that attributes that require expansion steps are selected using the "./"-operator. So we write $T./f$ instead of $(T.a).f$ and $T./m$ instead of $((T.a).f).m$. Occasionally, we also make use of anonymous attributes. For instance, we could have defined the tuple T above by $T = \langle\langle e, f, g\rangle, b, c, d\rangle$. Note that T is still a quadruple, but its first attribute has become an anonymous triple. In this case, we indicate the second attribute of the first attribute of T simply by $T.f$. Avoiding the "./"-operator, however, is not the reason for using anonymous attributes, but merely a side-effect. In principle, we use anonymous attributes to express that a newly defined tuple is an extension of its anonymous part. This is illustrated clearly in the UML diagrams, where we model anonymous attributes by specialization and named attributes by aggregation or composition.

Frequently, attributes of our models are functions of a special nature. In

[3]Strictly, this is an abuse of notation that blurs the distinction between attribute name and attribute value, but we will ensure that the resulting ambiguity can always be resolved from the context.

some cases, they are elements of a Cartesian product $\prod_{i\in I} V_i$, i.e, they are functions f from some index set I to a union $\bigcup_{i\in I} V_i$ that satisfy $f(i) \in V_i$. On other occasions, attributes are partial functions. To indicate a partial function f from A to B, we write $f : A \to B_\perp$. So, rather than making the subset A' of the function domain A whose elements have an image under f explicit, we extend the function range B with a special element $\perp$, which, by definition, is distinct from all elements of B. The extended range is denoted by $B_\perp$ and the set A' is implicitly given by $\{a \in A \mid f(a) \neq \perp\}$. Occasionally, we also find use for unordered pairs. Given a set S, we denote the set $\{\{s_1, s_2\} \mid s_1, s_2 \in S\}$ of all unordered pairs consisting of elements from S by $\binom{S}{2}$.

Finally, we use the following alphabet and font conventions. Instances are denoted by lower case roman symbols (c, n, ...), sets of instances are denoted by upper case roman symbols (C, N, ...), types are denoted by upper case sans serif symbols (C, N, ...), and sets of types by calligraphic symbols ($\mathcal{C}$, $\mathcal{N}$, ...). Universes, both of types and instances, are denoted by Euler Fraktur symbols. For instance, $\mathfrak{N}$ denotes the universe of names that are used to identify framework entities of various types. Other universes will be introduced as the need arises.

5 Resource view

In this section, we present a set of models that describe framework entities from a resource management perspective. These models are the most fundamental ones we consider. In later sections these models will be extended to incorporate other aspects of framework entities relevant for activities occurring in specific phases in their life cycle.

In practice, many kinds of resources can be distinguished, each of which has its own characteristics. In this report, however, we will not consider all those characteristics in detail, but describe a resource by means of a type, such as “processor”, “memory”, “energy”, or “network”, that is a rough indication of its *capabilities*, and a value which quantifies its *capacity*, i.e., the extent to which the resource possesses those capabilities.

So, we simply postulate a universe $\mathfrak{R}$ of *resource types* of which neither the number nor the nature of its individual elements is specified, and assume for each resource type $\mathsf{R} \in \mathfrak{R}$ the existence of a universe of values $\mathfrak{V}_\mathsf{R}$ whose elements are used to quantify and compare resources of type R.

We do not detail the sets $\mathfrak{V}_\mathsf{R}$, but note that in view of the above their values may be compounds. For instance, the value of a memory-type resource could state not only its storage capacity, but also its word size and the time it takes to store or retrieve a word. Also, when comparing resource values, it is important that these values are expressed using the same unit. In practice this need not be the case, but we assume that for each quantifiable capability there exists a standard unit, and that, given any context, it always possible to convert a non-standard resource value to a value in that standard. By definition, values from any universe $\mathfrak{V}_R$ are expressed in the corresponding standard unit, which, again, will be left anonymous.

Under the conventions above, a resource type instance, or *resource* for short, is a pair $\langle \mathsf{R}, v\rangle$ where $v \in \mathfrak{V}_{\mathsf{R}}$. Resources can be grouped into budgets, which give rise to our first model.

Model 5.1 ($\mathfrak{M}^{resource}_{budget}$)
Let $\mathcal{R} \subseteq \mathfrak{R}$ be a set of resource types. A *resource budget* for $\mathcal{R}$ is an element of $\prod_{\mathsf{R}\in\mathcal{R}} \mathfrak{V}_{\mathsf{R}}$, i.e., a tuple of resource capacities, one for each resource type $\mathsf{R} \in \mathcal{R}$. □

Resource budgets are used to model various resource management concepts. At any moment in time, a single resource supplier will supply its resources to a number of resource consumers. These consumers will specify their resource needs by means of a number of budgets, called *resource demands.* A consumer must specify at least two demands that specify the minimum respectively maximum resource usage over its entire life span. Of course, the *actual resource usage* at any particular moment in time is also expressed as a budget. Moreover, to assist resource management decisions, consumers may specify additional resource demands, like *typical usage.* As part of resource management, a resource supplier will set aside a resource budget for each consumer to which it supplies its resources. In general, these *resource allocations* have bounded duration and depend on the resource demands specified by the consumer. To avoid deadlock due to competition for resources, budget allocations have to be reserved beforehand. The total amount of *resource reservations* that a supplier can accommodate is called its *resource capacity* and is also expressed by a budget. Finally, given its resource capacity and outstanding allocations, a supplier has a *free budget* of remaining resources that is available for reservation.

The framework entity that supplies applications with resources is the platform. It consists of nodes that host the components of an application. These nodes are interconnected by networks that realize the data traffic associated with the interface bindings of an application. The resources supplied by hosts and networks are distinct. Henceforth, we therefore assume that the set of resource types $\mathfrak{R}$ is partitioned in two disjoint sets $\mathfrak{R}_{host}$ and $\mathfrak{R}_{net}$.

Model 5.2 ($\mathfrak{M}^{resource}_{host\_type}$)
In the resource view a *host type* is a pair $\langle \mathcal{R}, \mathtt{cap}\,\rangle$, where

1. $\mathcal{R} \subseteq \mathfrak{R}_{host}$ is a set of resource types,
2. $\mathtt{cap} \in \prod_{\mathsf{R}\in\mathcal{R}} \mathfrak{V}_{\mathsf{R}}$ is a budget indicating the resource capacities.

□

By definition, a host is a host-type instance. Since all information about the resources offered by a host is already captured by its set of resource types, the only additional information is its identity.

Model 5.3 ($\mathfrak{M}^{resource}_{host}$)
In the resource view a *host* is a pair $\langle \mathsf{H}, hn\rangle$, where

1. $\mathsf{H} \in \mathfrak{M}^{resource}_{host\_type}$ is a host type,

2. $hn \in \mathfrak{N}$ is a framework unique name.

□

Note that, in contrast to hosts, resources have no identity. Instead, they are identified through the host to which they belong. As a consequence, every host has only one resource of any single type. Obviously, this is not always conform reality. For instance, processors may have several cores, main memory may consist of several identical memory chips organized in banks, or secondary memory may be organized as a RAID. So, in our models, we implicitly assume that a host which in reality has two or more resources of the same type, with respective capacities κ_1 and κ_2, can always be modeled as if it had a single resource of capacity $f(\kappa_1, \kappa_2)$, with f a known function, e.g., addition. This assumption is not essential, but merely simplifies our models. Finally, one could argue that resource capacity is a property of a host and not of a host type. However, by modeling in this way, replacing a host by one of the same type will neither hamper nor benefit deployment of components.

In general, applications will consist of several components that are distributed over a number of hosts. The reasons for distribution can be various, but the one relevant from a resource perspective is that no single node has sufficient resources to execute the application in question at its required service level. In other words, there exists at least one resource for which the total resource demand of all components combined exceeds the capacity of that resource on any single host. Given this distributed nature of applications, facilities for data exchange between its components are needed; a service provided by platform entities known as networks. To be precise, a network is a framework entity that accepts data at one location, the source, and delivers that data at one or more other locations, the destinations. We assume that any location of a network that can serve as a source can also serve as a destination. These locations are called the external locations or *access points* of the network. In addition to external locations, a network may have internal locations or *routing points*, which are neither source nor sink, and merely serve to forward data on its journey from source to destination. In Section 7, where we model networks from a deployment perspective, we will return to these routing points and the physical links that connect them to the access points and to each other, but here we ignore them. Instead, we adopt the view that a network is an entity that supplies its users with a set of resources, called connections, that can be used to transfer data between access points.

Model 5.4 ($\mathfrak{M}^{resource}_{network\_type}$)
In the resource view a *network type* is a pair $\langle \mathsf{R}, A \rangle$, where

1. $\mathsf{R} \in \mathfrak{R}_{net}$ is the resource type of all network connections,

2. $A \subseteq \mathfrak{N}$ is a set of framework unique names that are addresses of the network's access points.

□

The network type resource model differs from the host type resource model in that it mentions only a single resource type. Thus, it models a homogeneous network, i.e, a network that offers the same communication facilities for each connection, irrespective of the pair of access points that are connected. Also a network type resource model contains no notion of capacity. Although physical links between individual network locations possess properties like bandwidth and propagation delay which constitute a form of capacity, such properties do not readily translate to similar ones for the logical connections between source-destination pairs offered by the network. Instead, network capacity — or more precisely the capacities of all existing logical connections between source-destination pairs — is a dynamic property of the network that depends both on network configuration and traffic load. For some networks, such as ATM networks, it is possible to configure the network in such a way that connections with fixed duration and guaranteed capacity arise; for others, such as the internet, connections can be established but capacities can not be guaranteed. So, a network description in terms of individual links, for which it is possible to specify a notion of capacity, contains too much detail for the type of high-level resource model we are interested in, whereas the instantaneous capacities of connections that depend on current network configuration and traffic load belong to the deployment view.

Thus what is left in the resource view is a network type model that only specifies a resource type R for connections and a set of access points that can be interconnected, but no actual connections, let alone the capabilities of such connections.

Similar to a host, a network is an instance consisting of a type and a name.

Model 5.5 ($\mathfrak{M}^{resource}_{network}$)
In the resource view a *network* is a pair $\langle \mathsf{N}, nn \rangle$, where

1. $\mathsf{N} \in \mathfrak{M}^{resource}_{network\_type}$ is a network type,
2. $nn \in \mathfrak{N}$ is a framework unique name.

□

Having defined the host and network model, we are in a position to define the platform as an aggregation of hosts and networks, with additional information to specify where hosts are attached to networks.

Model 5.6 ($\mathfrak{M}^{resource}_{platform}$)
In the resource view a *platform* is a triple $\langle H, N, \texttt{attm} \rangle$, where

1. $H \subseteq \mathfrak{M}^{resource}_{host}$ is a set of hosts,
2. $N \subseteq \mathfrak{M}^{resource}_{network}$ is a set of networks such that
   - $\forall_{n_1,n_2 \in N} \quad n_1./A \cap n_2./A = \emptyset$,
3. $\texttt{attm} : \prod_{n \in N} (n./A \to H_\perp)$ is a mapping, where $\texttt{attm}(n, a)$ indicates the host, if any, that is attached to network n at access point a.

In addition, the derived attribute

4. $\texttt{haps} : H \to \bigcup_{n \in N} n./A$ is a function that maps hosts to access points in agreement with $\texttt{attm}$, i.e.,
   - $\texttt{haps}(h) = \bigcup_{n \in N}\{a \in n./A \mid \texttt{attm}(n,a) = h\}$.

□

Since the platform is by definition framework unique, the platform model does not contain a name-attribute. Furthermore, note that it is possible for a platform to have several networks and that a single host can be attached to multiple access points, either belonging to a single network or to multiple networks.

It is the responsibility of the resource manager to maintain the platform model. Because the first attribute is a set of hosts, rather than a set of host names, this implies that the resource manager not only must be aware of the identity of the platform hosts, but also must have a local copy of the resource model of each of these hosts. The same holds for the network models of the platform.

From a resource management perspective, a component is both a consumer and a supplier. Its responsibility is to supply the services from which applications are composed and to do so it consumes platform resources. Its role as service supplier will be modeled in the next section. Here, we model its reliance on platform resources. We shall assume that components can operate in a number of *modes*, each of which is characterized by a pair of budgets specifying its minimum and maximum resource demand.

Model 5.7 ($\mathfrak{M}^{resource}_{component\_type}$)
In the resource view a *component type* is a quadruple $\langle \mathcal{R}, M, \texttt{dmd}_{min}, \texttt{dmd}_{max} \rangle$, where

1. $\mathcal{R} \subseteq \mathfrak{R}_{host}$ is a set of resource types,
2. $M \subseteq \mathbb{N}$ is a set of resource modes,
3. $\texttt{dmd}_{min} : M \to \prod_{\mathsf{R} \in \mathcal{R}} \mathfrak{V}_{\mathsf{R}}$ is a mapping from resource modes to demand budgets,
4. $\texttt{dmd}_{max} : M \to \prod_{\mathsf{R} \in \mathcal{R}} \mathfrak{V}_{\mathsf{R}}$ is a mapping from resource modes to demand budgets.

□

By definition, a *component* is a component type instance. Therefore, it has an identity expressed by its name. In practice, this name may have structure. For instance, it may consist of a host name and a process identifier. Such structure, however, can only be part of a deployment model. So, in the resource view, component names have no structure.

Model 5.8 ($\mathfrak{M}^{resource}_{component}$)
In the resource view a *component* is a pair $\langle \mathsf{C}, cn \rangle$, where

1. $\mathsf{C} \in \mathfrak{M}^{resource}_{component\_type}$ is a component type,

2. $cn \in \mathfrak{N}$ is a framework unique name.

$\square$

A component type model is part of a component description which must be stored in the framework repository. The framework entities responsible for keeping track of components will be discussed in Section 7, where we discuss the deployment view.

6 Design view

In this section we present two models that characterize an application at various stages of its design phase (see Figure 2), a model that prescribes deployment constraints, and a number of auxiliary models.

From a design perspective, an application is a composition of services. A service is characterized by a set of *interfaces* by means of which it interacts with other services, and a *contract* that specifies both functional and extra-functional aspects of the service. Since our focus is on resource management, we do not use application design models to establish whether an application built from services provides the desired functionality. We are, however, interested in the interaction patterns of an application, because they define the communication resources needed for its execution, i.e, the binding pattern of its services. Hence, in this report, we use a simple service model that considers interfaces but ignores contracts. When, on rare occasions, we do need to refer to service contracts, we do so in an informal manner.

Each interface is characterized by a type, which specifies the number, names and signatures of its operations. Based on the role they play in establishing its functionality, the interfaces used by a service are divided into two groups, viz. the *provided interfaces* and the *required interfaces.* Interaction between a pair of services along an interface can only take place when the interface assumes opposite roles in each member of that pair. In our models, we implicitly assume that services interact along interfaces of the same type. In practice, a notion of subtypes can be defined. In that case, the required interface needs to be only a subtype of the provided interface. For the same reason that we ignore service contracts, we also ignore the operations of an interface. Hence an interface type is modeled as an atomic entity without any structure, i.e., as an element from a universe $\mathfrak{I}$ of interface types. As a consequence, the only operation that can be performed on interface types is testing whether two types are equal, which is sufficient for our purpose.

Model 6.1 ($\mathfrak{M}^{design}_{service\_type}$)
A *service type* is a pair $\langle \mathcal{I}_{req}, \mathcal{I}_{prv} \rangle$, where

1. $\mathcal{I}_{req} \subseteq \mathfrak{I}$ is a set of interface types,

2. $\mathcal{I}_{prv} \subseteq \mathfrak{I}$ is a set of interface types

such that

- $\mathcal{I}_{req} \cap \mathcal{I}_{prv} = \emptyset$.

Since we do not always need to distinguish between interface types of a service based on their role, we introduce the derived attribute

3. $\mathcal{I} \subseteq \mathfrak{I}$ is the set of interfaces defined by
   - $\mathcal{I} = \mathcal{I}_{req} \cup \mathcal{I}_{prv}$.

□

In this model, services have at most one interface of any type, because from a logical perspective there is no need to provide the same functionality through several interfaces[4]. As a consequence, the model identifies the interfaces of a service by their unique type, instead of introducing separate identifiers.

A service, as is customary for an instance entity, is nothing but a service type and a name.

Model 6.2 ($\mathfrak{M}^{design}_{service}$)
A *service* is a pair $\langle \mathsf{S}, sn \rangle$, where

1. $\mathsf{S} \in \mathfrak{M}^{design}_{service\_type}$ is a service type,
2. $sn \in \mathfrak{N}$ is a framework unique name.

□

Although a service for which $\mathsf{S}.\mathcal{I}_{prv} = \emptyset$ is completely useless in practice, we do not impose this constraint in our simple service model.

The first design model of an application presented in this section is based on this simple service model. It defines the outcome of the "compose application" activity of the design phase (see Figure 2), in which an application is defined as a collection of services whose communication topology is given by a collection of *bonds* between their interfaces.

Model 6.3 ($\mathfrak{M}^{design}_{bond}$)
A *bond* is a pair $\langle \mathsf{I}, bn \rangle$, where

1. $\mathsf{I} \in \mathfrak{I}$ is an interface type,
2. $bn \in \mathfrak{N}$ is a framework unique name.

□

Binding interfaces of services is not arbitrary, but must obey two rules[5]. The first and most important binding rule states that each bond links a pair

[4]This does not preclude that, in the deployment phase, code and resources can be replicated to provide the service's functionality at a required QoS level.

[5]Note that we make a distinction between the composition activity, i.e. binding, and the entities, i.e., the bonds by which this is achieved.

of interfaces of the same type and opposite roles: one required interface and one provided interface. The second rule states that there can be only one bond per interface type between any pair of services. The latter rule is not very restrictive. In particular, it hardly constrains the number of bonds in which an interface participates. As displayed in Figure 6, of the 8 possible binding patterns for any pair of bonds allowed by the first rule, only one is excluded by the second rule.

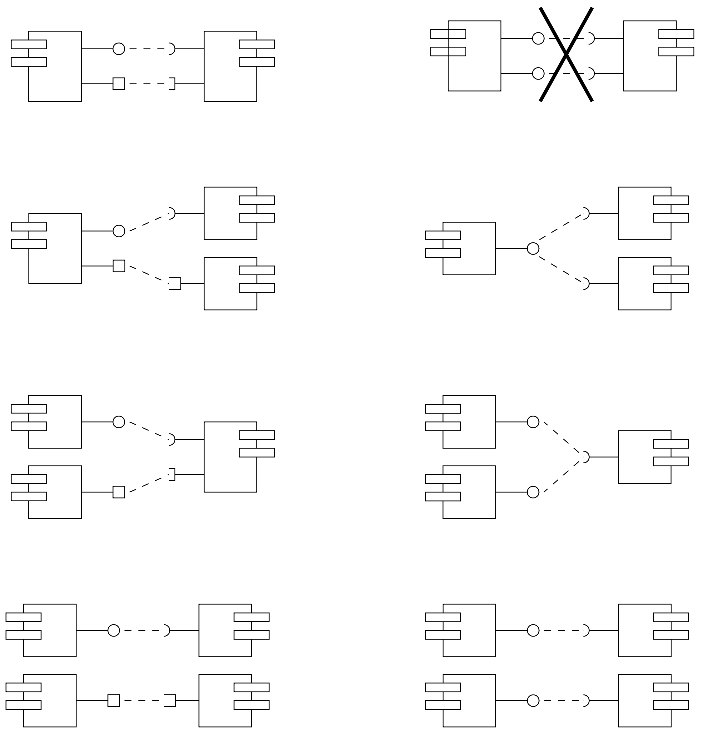


Figure 6: Component binding patterns consisting of two bonds. Components on the left hand side of a pattern expose provided interfaces, components on the right hand side expose required interfaces. Two component types (circular and angular) are distinguished. Of the eight possible patterns only one is forbidden.

Furthermore, notice that a service-based application design is a static model that does not provide any indication of the life span of the connections that realize bonds at run-time. For any interface participating in two or more bonds, corresponding run-time connections may either exist simultaneously or one at a time or in any combination thereof.

Model 6.4 ($\mathfrak{M}^{design}_{sb\_app\_design}$)
A *service-based application design* is a quadruple $\langle S, B, \mathtt{req}, \mathtt{prv} \rangle$, where

1. $S \subseteq \mathfrak{M}^{design}_{service}$ is a set of services,
2. $B \subseteq \mathfrak{M}^{design}_{bond}$ is a set of bonds,

3. $\texttt{req} : B \to S$ is a function indicating the service that exposes the required interface of the bond interface type and that satisfies
   - $\forall_{b \in B} \quad b.\mathsf{I} \in (\texttt{req}\,(b)).\!/\mathcal{I}_{req}$,
4. $\texttt{prv} : B \to S$ is a function indicating the service that exposes the provided interface of the bond interface type and that satisfies
   - $\forall_{b \in B} \quad b.\mathsf{I} \in (\texttt{prv}\,(b)).\!/\mathcal{I}_{prv}$,

such that

- $\forall_{b_1, b_2 \in B}\ (b_1.\mathsf{I} = b_2.\mathsf{I}) \ \Rightarrow\ (\texttt{req}\,(b_1) \neq \texttt{req}\,(b_2) \vee \texttt{prv}\,(b_1) \neq \texttt{prv}\,(b_2))$.

□

Recall that components are the units of deployment in our framework, and that they are the only means by which services can be realized. So given a service-based application the next design step is to identify a set of component(type)s that provides its services. For this purpose, we extend the component type resource model from Section 5 with the interface types of the services it provides.

Model 6.5 ($\mathfrak{M}^{design}_{component\_type}$)
In the design view, a *component type* is a quadruple $\langle\langle \mathsf{C}\rangle, \mathcal{I}_{req}, \mathcal{I}_{prv}, \texttt{dep}\,\rangle$, where

1. $\mathsf{C} \in \mathfrak{M}^{resource}_{component\_type}$ is a component type in the resource view,
2. $\mathcal{I}_{req} \subseteq \mathfrak{I}$ is a set of interface types, viz., the ones required by a component of this type,
3. $\mathcal{I}_{prv} \subseteq \mathfrak{I}$ is a non-empty set of interface types, viz., the ones provided by a component of this type such that
   - $\mathcal{I}_{req} \cap \mathcal{I}_{prv} = \emptyset$,
4. $\texttt{dep} : \mathcal{I}_{prv} \to \mathbb{P}(\mathcal{I}_{req})$ is a mapping, where $\texttt{dep}\,(\mathsf{I})$ indicates the set of required interfaces invoked by a component of this type to realize the functionality provided through interface I.

Besides these attributes we also introduce two derived attributes.

5. $\mathcal{I} \subseteq \mathfrak{I}$ is the set of interface types defined by
   - $\mathcal{I} = \mathcal{I}_{req} \cup \mathcal{I}_{prv}$.

This attribute is introduced, because we do not always need to distinguish between interface types of a component.

6. $\mathcal{S}_{impl} \subseteq \mathfrak{M}^{design}_{service\_type}$ is the set of service types defined by
   - $\mathcal{S}_{impl} = \{\langle \mathcal{I}_r, \mathcal{I}_p\rangle \mid \emptyset \subset \mathcal{I}_p \subseteq \mathcal{I}_{prv} \wedge \bigcup_{\mathsf{I} \in \mathcal{I}_p} \texttt{dep}\,(\mathsf{I}) \subseteq \mathcal{I}_r \subseteq \mathcal{I}_{req}\}$.

Services with a type from $\mathcal{S}_{impl}$ can use a component of this component type for their realization. In fact, our interest in this derived attribute is the main reason for introducing the basic attribute $\texttt{dep}$. $\square$

The derived attribute S_{impl} of the design view component type model expresses that a component of this type can be used to realize any service that provides a subset of the component's functionality and that requires a superset of the functionality needed by the component to implement that service's functionality.

Since component-based designs rely on component types only, components do not play a role in the design phase. Nevertheless, we also introduce a model for components in this view, because it simplifies definitions in subsequent views.

Model 6.6 ($\mathfrak{M}^{design}_{component}$)
In the design view, a *component* is a pair $\langle\langle\langle \mathsf{C}\rangle, \mathcal{I}_{req}, \mathcal{I}_{prv}, \texttt{dep}\,\rangle, cn\rangle$, where

1. $\langle\langle \mathsf{C}\rangle, \mathcal{I}_{req}, \mathcal{I}_{prv}, \texttt{dep}\,\rangle \in \mathfrak{M}^{design}_{component\_type}$ is a component type in the design view,
2. $\langle \mathsf{C}, cn\rangle \in \mathfrak{M}^{resource}_{component}$ is a component in the resource view.

$\square$

Indeed this model indicates that a component in the design view is nothing but a component from the resource view albeit with an extended type.

Based on the extended component type model, we define the second application model, which describes the outcome of the "allocate components" activity from Figure 2. In this model, each service has been allocated a component type whose components can implement the service, i.e., can provide the desired functionality at the appropriate service level.

Model 6.7 ($\mathfrak{M}^{design}_{cb\_application}$)
A *component-based application design* is a triple $\langle\langle S, B, \texttt{req}, \texttt{prv}\,\rangle, \mathcal{C}, \texttt{cta}\,\rangle$, where

1. $\langle S, B, \texttt{req}, \texttt{prv}\,\rangle \in \mathfrak{M}^{design}_{sb\_app\_design}$ is a service-based application design,
2. $\mathcal{C} \subseteq \mathfrak{M}^{design}_{component\_type}$ is a set of component types in the design view, such that all services in S can be implemented using components with a type from this set, i.e.,
   - $S \subseteq \bigcup_{\mathsf{C}\in\mathcal{C}} \mathsf{C}.\mathcal{S}_{impl}$,
3. $\texttt{cta} : S \to \mathcal{C}$ is a mapping that indicates the type of the component used to realize a service. So for all $s \in S$ we must have
   - $s.\mathsf{S} \in (\texttt{cta}\,(s)).\mathcal{S}_{impl}$,
   - there exists a resource mode $m \in \texttt{cta}\,(s).M$ such that deployment of a component of type $\texttt{cta}\,(s)$ in mode m yields the QoS required by the contract of s.

□

The constraint imposed on $\mathcal{C}$ in this model not only guarantees the existence of the attribute `cta`, but also limits the service-based application designs that can be considered.

The constraints imposed on `cta` itself, however, are not very restrictive. In particular, since `cta` need not be injective, the component-based application design model allows applications in which a single component realizes several services, possibly sharing some interfaces. Of course, the latter requires the existence of a resource mode that guarantees all those services their desired QoS. On the one hand, this generality complicates not only the actual deployment of a component-based application design, which is discussed in Section 7, but also complicates the formulation of deployment constraints which we will discuss next. On the other hand, the fact that `cta` is a function instead of a relation also simplifies deployment somewhat. For instance, it precludes that the required service level is obtained by parallel deployment of several components of identical type. Such deployment scenario's are conceivable, e.g., when processing a video stream on a frame-by-frame basis. To double the throughput we could demultiplex the stream into two substreams containing the odd and even numbered frames respectively, apply the processing service using two identical components, one for each substream, and multiplex the filtered streams into a single stream again. To keep our model simple, however, we assume that this type of parallelism is encapsulated in the component types, where it can be reflected by a variety of resource modes each with corresponding resource demands.

The final activity of the design phase involves prescription of deployment constraints. We consider two types of such constraints: constraints on resources and constraints on locations. Resource usage is determined by the resource modes in which components are deployed. The permissible resource modes of a component, in turn, are determined by the QoS-levels required by the services it realizes. Therefore, ideally, a resource constraint should specify for each service s a subset of resource modes from $\mathtt{cta}\,(s)$. This set, however, may depend on the existence of other services with whom s shares its component of type $\mathtt{cta}\,(s)$. Here, we will ignore this complication and assume that resource constraints can indeed be formulated as a predicate on functions of type $\prod_{s \in \mathcal{S}} \mathtt{cta}\,(s).M$, which we call *resource mode assignments.* We justify this approach by observing that it is always possible to realize each service by a separate component.

The formulation of location constraints suffers from a different problem. Whereas some constraints do not require specific knowledge of locations, such as constraints that merely specify that services have to be co-located, others, such as specifying the geographical location of camera services, do imply platform knowledge. The platform, however, is not part of a component-based application design. Hence, for proper formulation of location constraints, we must assume at least partial knowledge of the platform on which the application is to be deployed, viz. its set H of hosts. Given this set, location constraints can be formulated as a predicate on functions of type $S \to H$, which we call *host allocations.* Typical location constraints are: $\mathtt{ha}\,(s) = h$, indicating that service

s must be located on node h, or $\texttt{ha}\,(s_1) \neq \texttt{ha}\,(s_2)$, indicating that services s_1 and s_2 must not be co-allocated. Location constraints such as the latter, are a reason why services of the same type sometimes have to be distributed over several instances of the same component type.

Model 6.8 ($\mathfrak{M}^{design}_{constraint}$)
Let $\mathbb{D}_{cb} = \langle\langle S, B, \texttt{req}\,, \texttt{prv}\,\rangle, \mathcal{C}, \texttt{cta}\,\rangle$ be a component-based application design and let $\mathbb{P} = \langle H, N, \texttt{attm}\,\rangle$ be a platform. A *deployment constraint* for $\mathbb{D}_{cb}$ on $\mathbb{P}$ is a pair $\langle P_{res}, P_{loc}\rangle$, where

1. $P_{res} : (\prod_{s\in S} \texttt{cta}\,(s).M) \rightarrow \mathbb{B}$ is a predicate specifying resource constraints,

2. $P_{loc} : (S \rightarrow H) \rightarrow \mathbb{B}$ is a predicate specifying location constraints,

with $\mathbb{B}$ denoting the set of Boolean values $\{true, false\}$. $\Box$

This model does not constrain network connections. Although QoS requirements in service contracts certainly may give rise to such constraints, we have not modeled network constraints for two reasons. First, it strongly depends on the amount of control that can be exercised over the network whether constraints can be enforced upon deployment. Second, networks that do offer such control, require a more detailed model than the one given in Section 5 to express that constraints are met during deployment.

We end this section with a discussion of responsibilities of the various framework management and storage entities in the design phase. Although designing an application is mainly a human activity performed by an *application architect*, several of those entities are also involved in this activity. First, the framework repository defines, through its component descriptions, the service types from which an application can be built. Second, the resulting designs, both the service-based version and the component-based version, are maintained by an orchestrator. This orchestrator could also assist the application architect in determining the component type assignment `cta`, although the description of service contracts may be such that verifying whether a resource mode exists that fits the QoS mentioned in the contract can only be done by the application architect.

7 Deployment view

In this section we introduce the four models associated with the main activities of the deployment phase of the application life cycle, as indicated in Figure 3. For this, it is necessary to extend some of the models presented in the previous sections. For instance, in the deployment view of a platform entity, not only the resources it provides must be modeled, but also the applications deployed on it. Moreover, as indicated in Section 2, components are deployed in a certain resource mode and with their interfaces bound to those of other components. As a consequence, also the component design model, which by itself is an extension of the component resource model, has to be extended. The deployment view, however, neither extends nor introduces type models, since, by its very nature, deployment is only concerned with instances.

We begin our discussion with the extension of the component model.

Model 7.1 ($\mathfrak{M}^{deployment}_{component}$)
In the deployment view, a *component* is a quadruple $\langle\langle\langle\langle \mathsf{C}\rangle, \mathcal{I}_{req}, \mathcal{I}_{prv}, \mathtt{dep}\,\rangle, cn\rangle, S, m, \mathtt{peers}\,\rangle$, where

1. $\langle\langle\langle \mathsf{C}\rangle, \mathcal{I}_{req}, \mathcal{I}_{prv}, \mathtt{dep}\,\rangle, cn\rangle \in \mathfrak{M}^{design}_{component}$ is a component in the design view,
2. $S \subseteq \mathfrak{M}^{design}_{service}$ is the set of services that use the component and therefore must satisfy
   - $\forall_{s \in S} \quad s.\mathsf{S} \in \mathsf{C}.\mathcal{S}_{impl}$,
3. $m \in \mathsf{C}.M$ is the resource mode in which the component should operate when active,
4. $\mathtt{peers}\, : \mathsf{C}.\mathcal{I} \to 2^{\mathfrak{N}_{comp}}$ is mapping, where $\mathtt{peers}\,(\mathsf{I})$ is the set of all peers, identified by their framework unique component name, with whom the component interacts via interface I.

Moreover, we introduce the derived attribute

5. $N_{peers} \subseteq \mathfrak{N}_{comp}$ is the set of names of all peers of this component, defined by
   - $N_{peers} = \bigcup_{\mathsf{I} \in \mathsf{C}.\mathcal{I}} \mathtt{peers}\,(\mathsf{I})$.

$\Box$

In this model a deployed component neither is, nor needs to be, aware of the fact that all bonds required by the application(s) in which it participates are established. In particular, for every interface I of a component, neither does $\mathtt{peers}\,(\mathsf{I}) = \emptyset$ imply that binding of I still has to take place, nor does $\mathtt{peers}\,(\mathsf{I}) \neq \emptyset$ imply that binding of I is completed. In particular, a component c will never communicate over an interface $\mathsf{I} \in c.\mathcal{I} \setminus \left(\bigcup_{s \in c.S} s./\mathcal{I}\right)$, so for those interfaces certainly $c.\mathtt{peers}\,(\mathsf{I}) = \emptyset$.

In order for components to communicate with each other over an interface, however, a connection needs to be established. For that, the initiating component needs to indicate the other party (its peer). Because a reusable component should be oblivious of the platform upon which it is deployed, it can only do so by identifying its peer by name. The information necessary to resolve that name to the peer's host address on the platform can be found in the platform model (function `hiaps`) to be discussed later in this section, and will be maintained by the framework entities responsible for that model.

It is the responsibility of a component itself to maintain its components model. As a consequence, every component deployed by the framework must contain code that performs this task. We assume that this is achieved by instrumenting each component, as delivered by its developer, with additional services that take care of this task, in accordance with some well-defined framework standard.

Next, we consider the platform entities. Modeling the platform is complicated by the fact that at any single moment in time there may be various applications present on the platform, each of which may be engaged in a different activity of the deployment (or operation) phase. The platform entity models are oblivious of this. Only the framework management entities that run on it, carry this knowledge.

First, we extend the resource model of a host to include the components that are deployed on it. To that end, each host is equipped with a set of component types that represent the set of binaries that are installed on that host. Obviously, the component types that can be installed on a host depend on the resource types present on that host, but other than that there are no constraints. Whether component types are stand-alone executable programs or library modules is irrelevant; only that they can be instantiated into components matters. Although each component must have its component type installed on every host where it is deployed, the reverse is not required. For each component type installed on a host there can be zero or more deployed components. Thus, the model is able to capture intermediate stages of an incremental installation and instantiation procedure. Also, this model implies that when the last component of a certain type is destroyed on any host, e.g. because it is migrated to another host, there is no need to uninstall its component type. In addition to the above, the host deployment model describes a breakdown of a host's resource capacity into budgets allocated to currently deployed components and a remaining free budget for components to be deployed in the future. These budgets are constrained both by the resource demands of the deployed components and the resource capacity of the host.

Model 7.2 ($\mathfrak{M}^{deployment}_{host}$)
In the deployment view a *host* is a quadruple $\langle\langle \mathsf{H}, hn\rangle, \mathcal{C}, C, \mathtt{ra}\rangle$, where

1. $\langle \mathsf{H}, hn\rangle \in \mathfrak{M}^{resource}_{host}$ is a host in the resource view,
2. $\mathcal{C}$ is a set of component types such that
   - $\forall_{\mathsf{C}\in\mathcal{C}} \quad \mathsf{C}.\mathcal{R} \subseteq \mathsf{H}.\mathcal{R}$,
3. $C \subseteq \mathfrak{M}^{deployment}_{component}$ is a set of components in the deployment view such that
   - $\forall_{c\in C} \quad c.\mathsf{C} \in \mathcal{C}$,
4. $\mathtt{ra} : C \to \prod_{\mathsf{R}\in\mathsf{H}.\mathcal{R}} \mathfrak{V}_{\mathsf{R}}$ is a mapping from components to resource budgets such that
   - $\forall_{\mathsf{R}\in c./\mathcal{R}} \quad c./\mathtt{dmd}_{min}(c.m, \mathsf{R}) \leq \mathtt{ra}\,(c, \mathsf{R}) \leq c./\mathtt{dmd}_{max}(c.m, \mathsf{R})$,
   - $\forall_{\mathsf{R}\in\mathsf{H}.\mathcal{R}} \quad (\sum_{c\in C} \mathtt{ra}\,(c, \mathsf{R})) \leq (\mathsf{H}.\mathsf{cap}\,)(\mathsf{R})$.

To facilitate the description of other models presented in this section, we introduce the derived attribute

5. $\texttt{free} : \prod_{\mathsf{R} \in h./\mathcal{R}} \mathfrak{V}_{\mathsf{R}}$ is a resource budget, defined by

   - $\texttt{free}\,(\mathsf{R}) + \sum_{c \in C} \texttt{ra}\,(c, \mathsf{R}) = (\mathsf{H.cap}\,)(\mathsf{R})$.

□

To determine which framework entities are responsible for maintaining the host deployment models, we must consider in more detail the rôles that the various framework managers play in the activities of the deployment phase. It is a device manager's task to create processes, i.e. to instantiate components, that run executables, i.e. component types. If these executables are not yet installed on the host, it is also the device manager's responsibility to download them from the repository. From this it follows that the device managers should be responsible for maintaining the second and third attribute of the host deployment model. Another task of device managers is to provide every component with the resources it needs. The amounts that need to be allocated are specified by resource reservations, which the device manager obtains from the resource manager. So it could be argued that only the resource manager is responsible for maintaining the `ra`-attribute of the host deployment model. On the other hand, by having the device managers maintain a local copy it is possible for the framework to distinguish between the situation in which the reservation is made by the resource manager and the situation in which it is effectuated by the device manager. Since we consider this distinction to be important, e.g., because after a component crash it should be possible to determine whether there are still outstanding reservations, we stipulate that the `ra`-attribute is also maintained by the device managers. A disadvantage of this arrangement is that the values maintained by the device managers need to be consistent with the values maintained by the resource manager. Since the `free`-attribute is a derivative of the `ra`-attribute, it should be maintained in the same manner.

Next, we extend the resource model of a network with the logical connections it provides. In general, there may be multiple connections between a single pair of access points. For example, consider a deployment of four components on two hosts, where components c_1 and c_2 are located on host h_1, and components c_3 and c_4 are located on host h_2, and where two interface bindings are needed, one between the even-numbered components, and one between the odd-numbered components. In this case, two connections are needed, also when the bindings are of same interface type. As another example, consider a single pair of components located on distinct hosts that have two or more bindings in common. In this case there is a choice. Besides a separate connection per binding, also a single connection that interleaves data traffic belonging to the different bindings may be used when interface types are the same. Anyway, the model must be general enough to allow several connections between a single pair of access points.

Model 7.3 ($\mathfrak{M}^{deployment}_{network}$)
In the deployment view a *network* is a pair $\langle\langle \mathsf{N}, nn\rangle, L\rangle$, where

1. $\langle \mathsf{N}, nn\rangle \in \mathfrak{M}^{resource}_{network}$ is a network in the resource view,

2. $L \subseteq \binom{\mathsf{N}.A}{2}$ is a set of pairs of access points representing the set of connections currently maintained by N.

□

In this model a connection is oblivious of the interface binding(s) it accommodates and of the components by which it is used.

It is difficult to indicate which framework entities are responsible for maintaining the network deployment model. In case the framework has control over the internal points, i.e., the routers of the network, connections are established by suitable configuration of these routers, and the network deployment model is maintained in a distributed manner as part of the router configurations. If, in addition, network configuration also establishes the communication capabilities of a connection, then the situation becomes similar to resource reservation on hosts and maintaining network configurations becomes a responsibility of the resource manager. If, on the other hand, the framework has little to no control over the network, as is the case for hosts connected via the internet, connections are maintained as part of the communication protocols used by components to establish their interface bindings.

Finally, we extend the resource model of the platform, to incorporate the extended models of its hosts and networks.

Model 7.4 ($\mathfrak{M}^{deployment}_{platform}$)
In the deployment view a *platform* is a quadruple $\langle H, N, \texttt{attm}, \texttt{hiaps}\rangle$, where

1. $H \subseteq \mathfrak{M}^{deployment}_{host}$ is a set of hosts in the deployment view such that
   - $\forall_{h_1,h_2\in H} \quad h_1.C \cap h_2.C = \emptyset$,
2. $N \subseteq \mathfrak{M}^{deployment}_{net}$ is a set of networks in the deployment view such that
   - $\forall_{n\in N}\forall_{\{a_1,a_2\}\in n.L} \quad \texttt{attm}(n,a_1) \neq \perp \wedge \texttt{attm}(n,a_2) \neq \perp$,
3. $\texttt{attm} : \prod_{n\in N}(n./A \to H_\perp)$ is mapping from network-address pairs to hosts,
4. $\texttt{hiaps} : \prod_{h\in H}\left(\prod_{c\in h.C}\left(c./\mathcal{I} \to \left(\bigcup_{n\in N} n./A\right)_\perp\right)\right)$ where $\texttt{hiaps}(h,c,\mathsf{I})$ is the access point of h that exposes the interface of type I of component c to components on other hosts of the platform.

In order to formulate the remaining platform constraints imposed in the deployment view, we first need to introduce a number of derived attributes.

5. H_{res}, is the restriction of the host set obtained by looking only at their resources, i.e.,
   - $H_{res} = \{h.\langle \mathsf{H}, hn\rangle \mid h \in H\}$,
6. N_{res} is the restriction of the network set obtained by looking only at their resources, i.e.,
   - $N_{res} = \{n.\langle \mathsf{N}, nn\rangle \mid n \in N\}$,

7. $\mathtt{attm}_{res} : \prod_{n \in N_{res}} (n./A \to H_{res\perp})$ is defined by

   - $\forall_{h \in H} \quad \mathtt{attm}_{res}(h.\langle \mathsf{H}, hn \rangle) = \mathtt{attm}(h).\langle \mathsf{N}, nn \rangle$,

8. C_{pfm} is the set of components present on the platform, i.e.,

   - $C_{pfm} = \bigcup_{h \in H} h.C$,

9. L_{pfm} is the set of logical connections present on the platform, i.e.,

   - $L_{pfm} = \bigcup_{n \in N} n.L$,

10. N_{pfm} is the set of names of all components on the platform, i.e.,

    - $N_{pfm} = \{c.cn \mid c \in C_{pfm}\}$,

11. $\mathtt{host} : C_{pfm} \to H$ is a mapping that assigns to each component its host, i.e.,

    - $\forall_{c \in C_{pfm}} \quad c \in \mathtt{host}(c).C$,

12. $\mathtt{netw} : L_{pfm} \to N$ is a mapping that assigns to each logical connection its network, i.e.,

    - $\forall_{l \in L_{pfm}} \quad l \in \mathtt{netw}(l).L$.

In terms of these derived attributes the remaining platform constraints are now given by:

the first three attributes of a deployment view platform constitute a resource view platform, when restricted to their resources only

- $H_{res} \subseteq \mathfrak{M}^{resource}_{host} \wedge N_{res} \subseteq \mathfrak{M}^{resource}_{net} \wedge \langle H_{res}, N_{res}, \mathtt{attm}_{res} \rangle \in \mathfrak{M}^{resource}_{platform}$,

the access points by which a host exposes the interfaces of its components indeed belong to the set of access points of that host

- $\forall_{h \in H} \forall_{c \in h.C} \forall_{\mathsf{I} \in c./\mathcal{I}} \mathtt{hiaps}(h, c, \mathsf{I}) \in \mathtt{haps}_{res}(h)$,

every peer named by any component is itself present on the platform

- $\forall_{c \in C_{pfm}} \quad c.N_{peers} \subseteq N_{pfm}$,

the peer relationship is symmetric

- $\forall_{c_1, c_2 \in C_{pfm}} \quad c_1.cn \in c_2.N_{peers} \equiv c_2.cn \in c_1.N_{peers}$,

peers must have a common interface type

- $\forall_{c_1, c_2 \in C_{pfm}} \quad c_1.\mathcal{I} \cap c_2.\mathcal{I} \neq \emptyset$,

peers must be able to connect, i.e., their hosts are physically connected

- $\forall_{c_1, c_2 \in C_{pfm}} \quad c_1.cn \in c_2.N_{peers} \implies$.
  $\exists_{n \in N} \exists_{a_1, a_2 \in n./A} \quad \mathtt{attm}(n, a_1) = \mathtt{host}(c_1) \;\wedge\; \mathtt{attm}(n, a_2) = \mathtt{host}(c_2)$

□

Whereas, the first three attributes of this model are similar to those in the resource view platform model, the last attribute $\texttt{hiaps}$ is different from the derived attribute $\texttt{haps}$ of the resource view platform model. In particular, $\texttt{hiaps}$ is not a derived attribute, but carries independent information. To understand the nature of this information, recall from [27] that communication between components takes place in the transport layer between so-called transport service access points (TSAPs). In our model, these TSAPs are given by the interfaces of the components, i.e., by pairs (c, I) where $c \in C_{pfm}$ and $\mathsf{I} \in c./\mathcal{I}$. Hence, the bonds of a component-based application design indicate transport layer connections. At deployment, these connections are realized in three parts. The first part is a network layer connection, which is modelled by an element $l \in n.L$ for some network $n \in N$. This connection runs between two host access points, which are also known as network service access points (NSAPs) [27]. Hence, a second and third part is needed to describe how, on each of the two host, the TSAP is connected to the NSAP. This is what mapping $\texttt{hiaps}$ does for the entire platform (see Figure 7).

Next, we model the outcome of the various activities of the deployment phase of the application life cycle (see Figure 3). The first model defines the notion of a feasible deployment scheme. Deployment schemes are always defined for a particular component-based application design and with respect to a particular platform and consist of three mappings, as described in Section 2. The first two of these mappings have already been introduced in the previous section when the deployment constraints were defined. They are the host allocation $\texttt{ha}$ that maps services to locations, i.e., platform hosts, and the resource mode assignment $\texttt{ma}$ that assigns a resource mode to each service. The third mapping $\texttt{rsv}$ reserves for each service a resource budget that it should be granted at the next operation phase of the application. A deployment scheme for a platform is *feasible*, when the resource types and the reserved budgets are indeed available on the allocated hosts, and when all deployment constraints are met.

Model 7.5 ($\mathfrak{M}^{deployment}_{feasible\_scheme}$)

Given a platform $\mathbb{P} = \langle H, N, \texttt{attm}, \texttt{hiaps} \rangle \in \mathfrak{M}^{deployment}_{platform}$ in the deployment view, a component-based application design $\mathbb{D}_{cb} = \langle S, B, \texttt{req}, \texttt{prv}, \mathcal{C}, \texttt{cta} \rangle \in \mathfrak{M}^{design}_{cb\_app\_design}$ and a deployment constraint $\langle P_{loc}, P_{res} \rangle$ for $\mathbb{D}_{cb}$, a *feasible deployment scheme* for $\mathbb{D}_{cb}$ on $\mathbb{P}$ is a triple $\langle \texttt{ha}, \texttt{ma}, \texttt{rsv} \rangle$, where

1. $\texttt{ha} : S \to H$ is a mapping that allocates to each service a host on which it can be deployed, such that the allocated host possesses the right set of resources,i.e.,

   - $\forall_{s \in S} \quad \texttt{cta}(s).\mathcal{R} \subseteq \texttt{ha}(s)./\mathcal{R}$,

   and such that to services that share a bond either a single host is allocated, or a pair of hosts that are physically connected, i.e., that are attached to a common network

   - $\forall_{b \in B} \quad \texttt{ha}(\texttt{req}(b)) = \texttt{ha}(\texttt{prv}(b)) \vee$
     $\exists_{n \in N} \quad \texttt{haps}(\texttt{ha}(\texttt{req}(b))) \cap n./A \neq \emptyset \wedge$
     $\texttt{haps}(\texttt{ha}(\texttt{prv}(b))) \cap n./A \neq \emptyset$

and such that the location constraints are met, i.e.,

- $P_{loc}(\mathtt{ha})$ holds,

2. $\mathtt{ma} : \prod_{s \in S} \mathtt{cta}(s).M$ is a mapping that assigns to each service a resource mode in which is can be deployed, such that the resource constraints are met, i.e.,

   - $P_{res}(\mathtt{ma})$ holds

3. $\mathtt{rsv} : \prod_{s \in S} \prod_{\mathsf{R} \in \mathtt{cta}(s).\mathcal{R}} \mathfrak{V}_{\mathsf{R}}$ is a mapping that reserves for each service a resource budget such that for each resource used by that service, and considering the resource mode assigned to that service, the budget is neither too small, i.e.,

   - $\forall_{s \in S} \forall_{\mathsf{R} \in \mathtt{cta}(s).\mathcal{R}} \quad \mathtt{cta}(s).\mathtt{dmd}_{min}(\mathtt{ma}(s), \mathsf{R}) \leq \mathtt{rsv}(s, \mathsf{R})$,

   nor too large, i.e.,

   - $\forall_{s \in S} \forall_{\mathsf{R} \in \mathtt{cta}(s).\mathcal{R}} \quad \mathtt{rsv}(s, \mathsf{R}) \leq \mathtt{cta}(s).\mathtt{dmd}_{max}(\mathtt{ma}(s), \mathsf{R})$,

   and such that on every host the reserved budget is available

   - $\forall_{h \in H} \forall_{\mathsf{R} \in h.\mathcal{R}} \quad \left(\sum_{\mathtt{ha}(s) = h \,\wedge\, \mathsf{R} \in \mathtt{cta}(s).\mathcal{R}} \mathtt{rsv}(s, \mathsf{R}) \right) \leq h.\mathtt{free}(\mathsf{R})$.

$\square$

In this model the last constraint of the reservation guarantees that the reserved budgets are available on the allocated hosts even in the worst situation, in which every service is deployed on a separate component. This follows from the fact that each service that makes use of the resource has its separate term in the summation.

A feasible deployment scheme is the outcome of negotiations between the resource manager and the orchestrator of the component-based application. In general, many feasible deployment schemes exist and it is the orchestrator that has the responsibility to select one. In doing so, an orchestrator may adhere to framework specific strategies and cost criteria, to guide its selection. For the feasible deployment scheme model, however, these details are irrelevant.

At any moment in time, several negotiations between the resource manager and various orchestrators may be in progress. In that case, the resource manager has the responsibility to arbitrate between the resource reservations requested by the various orchestrators and prevent overbooking of resources. Because the resource manager is the unique framework entity involved in all negotiations, it can easily fulfill that responsibility. Disadvantages of this arrangement are that the resource manager is a single point of failure of the framework and a potential performance bottleneck. To avoid these disadvantages, a distributed resource manager is needed, but that is of no consequence for the models presented in this report.

An instantiation of a component-based application design on a platform consists of a set of components which together realize all services, and which are deployed on the hosts of the platform according to some feasible deployment.

Model 7.6 ($\mathfrak{M}^{deployment}_{installed\_app}$)
An *installed application* is a triple $\langle \mathbb{D}_{cb}, \mathbb{P}, \mathbb{F} \rangle$, where

1. $\mathbb{D}_{cb} \in \mathfrak{M}^{design}_{cb\_app}$ is a component-based application design,
2. $\mathbb{P} \in \mathfrak{M}^{deployment}_{platform}$ is a platform in the deployment view,
3. $\mathbb{F} \in \mathfrak{M}^{deployment}_{feasible\_scheme}$ is a feasible deployment scheme for $\mathbb{D}_{cb}$ on $\mathbb{P}$,

such that all component types are present on the hosts of the platform, i.e.,

- $\forall_{s \in \mathbb{D}_{cb}.S} \quad \mathbb{D}_{cb}.\mathtt{cta}\,(s) \in ((\mathbb{F}.\mathtt{ha}\,)(s)).\mathcal{C}$.

□

Model 7.7 ($\mathfrak{M}^{deployment}_{instantiated\_app}$)
An *instantiated application* is a pair $\langle \langle \mathbb{D}_{cb}, \mathbb{P}, \mathbb{F} \rangle, \mathtt{ca}\, \rangle$, where

1. $\langle \mathbb{D}_{cb}, \mathbb{P}, \mathbb{F} \rangle \in \mathfrak{M}^{deployment}_{installed\_app}$ is an installed application,
2. $\mathtt{ca}\, : \mathbb{D}_{cb}.S \rightarrow \mathbb{P}.C_{pfm}$ is an allocation of platform components to services,

such that each allocated component is of the specified type, i.e.,

- $\forall_{s \in \mathbb{D}_{cb}.S} \quad (\mathtt{ca}\,(s)).\mathsf{C} = (\mathbb{D}_{cb}.\mathtt{cta}\,)(s)$,

is on the right host i.e.,

- $\forall_{s \in \mathbb{D}_{cb}.S} \quad \mathtt{ca}\,(s) \in ((\mathbb{F}.\mathtt{ha}\,)(s)).C$,

is instantiated in the right mode, i.e.,

- $\forall_{s \in \mathbb{D}_{cb}.S} \quad \mathtt{ca}\,(s).m = (\mathbb{F}.\mathtt{ma}\,)(s)$,

and has been allocated the right amount of resources, i.e.,

- $\forall_{s \in \mathbb{D}_{cb}.S} \forall_{\mathsf{R} \in (\mathtt{cta}\,(s)).\mathcal{R}} \quad (\mathbb{F}.\mathtt{rsv}\,)(s, \mathsf{R}) = (((\mathbb{F}.\mathtt{ha}\,)(s)).\mathtt{ra}\,)(\mathtt{ca}\,(s), \mathsf{R})$.

Furthermore, we introduce the derived attribute

3. $C_{app} \subseteq \mathbb{P}.C_{pfm}$ is the set of components of the application given by
   - $C_{app} = \{\mathtt{ca}\,(s) \mid s \in \mathbb{D}_{cb}.S\}$.

□

Note that this model admits the possibility that a single component is instantiated to provide multiple services, in which case `ca` is not injective. This can only happen, however, when these services agree on the resource mode in which the component has to be deployed and on the resources reserved for it. Although not likely in practice, the model even admits that a single component is involved in multiple applications.

It is the responsibility of the application's orchestrator to instantiate the application. To this end, it instructs device managers to create the appropriate components from C on its host. For any $c \in C$ the code to be executed by c is part of its component type $c.\mathsf{C}$. If this type is not present on the host, the device manager also needs to install the type, i.e., download it from the repository. If the type contains source code instead of executable code, installation also involves compilation and linking. These activities are also the responsibility of the device managers[6].

Finally, a deployed application is an instantiated application in which all interface bindings are realized.

Model 7.8 ($\mathfrak{M}^{deployment}_{bonded\_app}$)
A *bonded application* is a pair $\langle\langle\langle\mathbb{D}_{cb}, \mathbb{P}, \mathbb{F}\rangle, \texttt{ca}\,\rangle, \texttt{la}\,\rangle$, where

1. $\langle\langle\mathbb{D}_{cb}, \mathbb{P}, \mathbb{F}\rangle, \texttt{ca}\,\rangle \in \mathfrak{M}^{deployment}_{instantiated\_app}$ is an instantiated application,
2. $\texttt{la} : \mathbb{D}_{cb}.B \to \mathbb{P}.L_{pfm}$ is an allocation of connections to bonds such that the bonded services are connected via the allocated connection, i.e.,

   - $\forall_{b \in \mathbb{D}_{cb}.B} \quad \texttt{la}\,(b) = \{\mathbb{P}.\texttt{hiaps}\,(h_r(b), c_r(b), b.\mathsf{I}), \mathbb{P}.\texttt{hiaps}\,(h_p(b), c_p(b), b.\mathsf{I})\}$,

   where the functions $h_r, h_p : B \to \mathbb{P}.H$ are given by

   - $h_r(b) = (\mathbb{F}.\texttt{ha}\,)((\mathbb{D}_{cb}.req)(b))$,
   - $h_p(b) = (\mathbb{F}.\texttt{ha}\,)((\mathbb{D}_{cb}.prv)(b))$

   and the functions $c_r, c_p : B \to \mathbb{P}.C_{pfm}$ are given by

   - $c_r(b) = \texttt{ca}\,((\mathbb{D}_{cb}.req)(b))$,
   - $c_p(b) = \texttt{ca}\,((\mathbb{D}_{cb}.prv)(b))$

Furthermore, we introduce the derived attribute

3. $L_{app} \subseteq \mathbb{P}.L_{pfm}$ is the set of logical connections of the application given by

   - $L_{app} = \{\texttt{la}\,(s) \mid s \in \mathbb{D}_{cb}.S\}$

$\square$

Recall from the discussion following the deployment platform model that a bond specifies a transport layer connection, and that such a connection is realized in three parts, two of which are described by `hiaps` and the third by the attribute `la` of the bonded application model above. The constraint in this model expresses that for each bond indeed three parts exist that can be chained together in such a manner that the indicated transport layer connection is established (see Figure 7).

It is the responsibility of the application's orchestrator to deploy the application. Part of that responsibility has been addressed by the previous model

[6]This implies that facilities for compilation and linking should be present on the host and have to be considered in the host allocation of a feasible deployment scheme, a fact that we have conveniently ignored in our models.

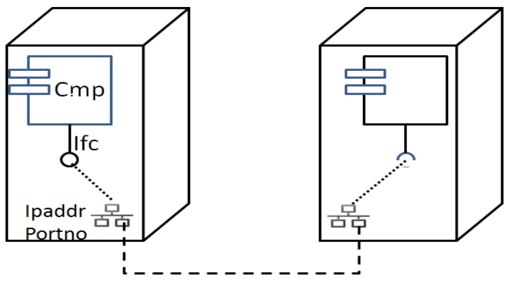


Figure 7: Transport layer connection between components consisting of 3 segments: from TSAP (Cmp, Ifc) to NSAP (Ipaddr, Portno) on each host, given by function `hiaps` of Model 7.4, and a network layer connection between the two NSAPs, given by function `la` of Model 7.8.

that ensures that components are created for all services. The other part is to ensure that transport layer connections are created for all interface bonds between services. To that end, the orchestrator informs components that are equipped with the required side of a bond interface of the address and identity of their peer and instructs them to set-up the connection.

8 Operation view

In this section we present an application model from the perspective of the operation phase. Although the activity diagram of the operation phase in Figure 4 distinguishes a large number of activities, a simple extension of the deployment model to keep track of the current operational status of its components is sufficient.

In accordance with the activity diagram of the operation phase, we model the operational status of a deployed component with a state diagram that contains three states and that is depicted in Figure 8. A deployed component can be either active or passive. All components enter the operation phase in the passive state and must be explicitly activated. Although, a component will start executing upon activation, being active does not necessarily mean that the component is executing. Since it may share its host with other components, belonging either to the same or to other applications, a component may also be blocked on access to a resource, e.g. cpu cycles. As long as QoS is not endangered, this is irrelevant for the operation of the application, which considers the operational status of the component to be normal.

Likewise, control commands that are issued to a component with status normal are irrelevant from the perspective of the application, and therefore do not result in a change of status. If, on the other hand, a resource conflict arises whose nature is distinct from the normal blocking described above, or if an other error that endangers the integrity of the application is signalled, the component will change its status to exceptional. When this happens, framework managers must start activities to resolve the error. Recall that we have adopted an extensive notion of error that does not necessarily refer to a current failure,

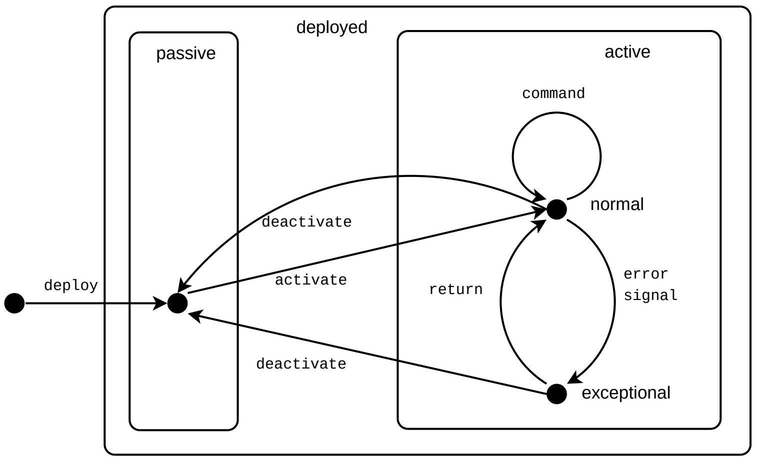


Figure 8: Operational status of a deployed component

but may also forecast a future problem. So, in the latter case, reconfiguration may take place without the application, or even the component that signalled the error, being interrupted, whereafter the component may resume its normal status. This is the best possible scenario from the perspective of the application. In the worst case, on the other hand, the entire application will fail. In all other cases, depending on the severity of the problem, at least some components are deactivated, and the operation phase is temporarily suspended for redeployment or redesign.

Model 8.1 ($\mathfrak{M}^{operation}_{runtime}$)
A *runtime application* is a pair $\langle\langle\langle\langle \mathbb{D}_{cb}, \mathbb{P}, \mathbb{F}\rangle, \texttt{ca}\,\rangle, \texttt{la}\,\rangle, \texttt{sa}\,\rangle$, where

1. $\langle\langle\langle \mathbb{D}_{cb}, \mathbb{P}, \mathbb{F}\rangle, \texttt{ca}\,\rangle, \texttt{la}\,\rangle$ is a bonded application in the deployment view,
2. $\texttt{sa} : \mathbb{P}.C_{pfm} \rightarrow \{passive, normal, exceptional\}_\perp$ is a function from components to states such that
    - $c \in C_{app} \Rightarrow \texttt{sa}\,(c) \neq \perp$.

A runtime application is called *passive*, when there exists at least one component $c \in A.C$ such that $\texttt{sa}\,(c) = passive$. Otherwise, it is called *active*.
□

It is the responsibility of an orchestrator to maintain the runtime model of its application. To fulfill this responsibility, it must be able to activate and deactivate components. For this, all components have to be equipped with an additional interface that allows an orchestrator to issue activation and deactivation commands. This is done as part of the instrumentation process to which components are subjected before they are admitted to the framework. Vice-versa, it is the responsibility of components to inform their orchestrators of a change in active state. This could, for instance, be done by a equipping each component with an interface that allows other components to subscribe to state-change events.

It makes sense to extend the range of the state function `sa` with states such as *instantiated*, *deployed*, and *crashed* that make it possible for an orchestrator to keep track of the progress of the deployment phase or fatal errors as well. For instance, $\mathtt{sa}(c) = \mathit{deployed}$ would indicate that as far as component c is concerned the constraints of Model 7.8 are satisfied. It would be the responsibility of the device managers to inform orchestrators of the occurrence of such additional states.

9 Related work

Over the past years extensions to the frameworks mentioned in Section 1 have been explored that facilitate dynamic configuration. Most of these are restricted to specific technologies and/or target specific application domains such as embedded systems. Batista and Rodriguez [2] have studied dynamic reconfiguration of CORBA-based components. Polakovic et. al. [24] use THINK an extension of Fractal and target design of dynamically reconfigurable operating systems. Rasche and Polze [25] have studied dynamic reconfiguration of mobile applications using the .Net framework. In terms of dynamic reconfiguration our work is perhaps most closely related to OpenCom [9] with which it shares the goals of both target domain independence and deployment environment independence. All these approaches, however, do not make special provisions for resource-awareness and do not focus on resource conflicts as a major cause for dynamic reconfiguration.

In terms of resource-awareness the Palladio Component Model (PCM)[3] is closest to our approach. In this model every provided service of a component is equipped with a so-called Resource Demand Service Effect Specification. These RDSEFFs abstractly model the externally visible behavior of a service with resource demands and calls to required services. Internal computations of components necessary to provide their services are clustered into actions that model only their resource demands. Thus, RDSEFFs can be used to predict runtime-performance at system design time which in the PCM precedes system deployment. Moreover, the PCM maintains a global repository of resource types which are provided by system deployers. RDSEFFs refer to these types without knowledge of resource instances. System deployers group resource types into containers, which play a similar role as the hosts in our model. Although the PCM models resources similar the models this report, they are not intended as a basis for dynamic reconfiguration. Hence the novelty of our approach lies in the combination of resource awareness and dynamic reconfiguration.

In this report we have described a restricted life cycle model that concentrates on runtime activities and focusses on applications instead of the components out of which these applications are built. Since the initial design and deployment of an application is considered an extreme case of reconfiguration, these phases are incorporated in the life cycle, but in a way that differs from more conventional life cycle models such as the ones presented in [12]. Thus our application life cycle comes closest to the reconfiguration process described by Kounev et.al. [18]. Moreover, phases which do not require separate application models, such as testing, have been ignored. Also maintenance, which is usually considered hand in hand with the operation phase, is not present. Maintenance of individual components to restore bugs is outside the scope of the report, and maintenance of an application that involves replacing components by newer versions can be treated similar to reconfiguration due to resource conflicts. In general, one would expect that a change in version also implies a change in resource demands. Although we use the life cycle and its associated models to identify responsibilities and task of the runtime platform, the design and maintenance of that platform itself is not explicitly addressed in our life cycle. However, it is possible to perceive the runtime platform as yet another, albeit special, application that provides the services needed by user applications. In this perspective, the platform can take care of its own resource management. Orchestrators and other platform entities can be reallocated and their quality of their services can be adapted to obtain the best performance of user applications.

10 Conclusions

In this report we have presented a number of models for dynamically reconfigurable applications. These applications are composed of services offered by components which either reside in a repository known to the application architect or are exposed over the WEB. The mechanism by which the components become available for application building, however, is immaterial for the models.

Because our models describe applications at various runtime stages, we have introduced an application life cycle model that contains just enough detail for that purpose. It consists of the operation phase encountered in conventional life cycle models extended with a redesign and a redeployment phase to capture dynamic reconfiguration activities. Since the extreme case of reconfiguration involves a complete redesign, initial application design and subsequent deployment becomes assimilated in this life cycle. This in contrast to more traditional life cycle models that consist of separate design and deployment phases and treat redesign in their maintenance phase. Besides this rearrangement of phases, we have kept the life cycle model as simple as possible, by omitting traditional phases that contain activities which do not require separate application models such as, e.g., testing.

The application models are especially tailored for resource management. Component models contain resource demands, platform models contain resource capacities and allocations, and deployment schemes contain resource reservations in agreement with resource usage state of the platform. Application deployment involves, amongst others, allocating resources according to reservations. The models are organized in a hierarchy with the resource models at the basis.

As a first step towards design of a resource-aware component framework, we have used the models to identify the services that the runtime environment has to provide to any application and to identify the platform entities responsible for delivery of those services. Further research must be aimed at a full specification of the architecture of such a runtime environment and its implementation. Also reconfiguration policies and strategies must be investigated and implemented in the appropriate platform management entities. Given sufficient sophistication of such policies and strategies, this can ultimately lead to a runtime environment for self-adaptive applications [11].

Finally, the models presented in this report are generic and qualitative. In practice, they have to be replaced by specific and quantitative models that describe concrete components and resources. In particular, automatic extraction of resource models from components would be an asset for any framework.

References

[1] Colin Atkinson and Dirk Muthig. Component-based product-line engineering with the uml. In Cristina Gacek, editor, *Software Reuse: Methods, Techniques, and Tools*, volume 2319 of *Lecture Notes in Computer Science*, pages 155–182. Springer Berlin / Heidelberg, 2002.

[2] Thais Batista, Ackbar Joolia, and Geoff Coulson. Managing dynamic reconfiguration in component-based systems. In Ron Morrison and Flavio Oquendo, editors, *Software Architecture*, volume 3527 of *Lecture Notes in Computer Science*, pages 1–17. Springer Berlin / Heidelberg, 2005.

[3] Steffen Becker, Heiko Koziolek, and Ralf Reussner. The palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3 – 22, 2009. Special Issue: Software Performance - Modeling and Analysis.

[4] Don Box. *Essential COM*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1997.

[5] Kris van Rens Bram Kersten and Rudolf Mak. Viframework: A framework for networked video streaming components. In Hamid R. Arabnia et al., editor, *Proceedings of the 2011 International Conference on Parallel and Distributed Processing Techniques and Applications*, PDPTA '11, pages 286–292. CSREA Press, 2011.

[6] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quema, and Jean-Bernard Stefani. An open component model and its support in java. In Ivica Crnkovic, Judith A. Stafford, Heinz W. Schmidt, and Kurt Wallnau, editors, *Component-Based Software Engineering*, volume 3054 of *Lecture Notes in Computer Science*, pages 7–22. Springer Berlin / Heidelberg, 2004.

[7] Tomas Bures, Petr Hnetynka, and Frantisek Plasil. Sofa 2.0: Balancing advanced features in a hierarchical component model. *Software Engineering Research, Management and Applications, ACIS International Conference on*, 0:40–48, 2006.

[8] Bill Burke and Richard Monson-Haefel. *Enterprise JavaBeans 3.0.* O'Reilly Media, Inc., 5th edition, May 2006.

[9] Geoff Coulson, Gordon Blair, Paul Grace, Francois Taiani, Ackbar Joolia, Kevin Lee, Jo Ueyama, and Thirunavukkarasu Sivaharan. A generic component model for building systems software. *ACM Trans. Comput. Syst.*, 26:1:1–1:42, March 2008.

[10] Ionut David, Bojan Orlic, Rudolf H. Mak, and Johan J. Lukkien. Towards resource-aware runtime reconfigurable component-based systems. In *Proceedings of the 2010 6th World Congress on Services*, SERVICES '10, pages 465–466, Washington, DC, USA, 2010. IEEE Computer Society.

[11] Betty H.C. Cheng et al. Software engineering for self-adaptive systems: A research roadmap. In Betty Cheng, Rogrio de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 1–26. Springer Berlin / Heidelberg, 2009.

[12] Syed Ahsan Fahmi and Ho-Jin Choi. Life cycles for component-based software development. In *citworkshops*, IEEE 8th International Conference on Computer and Information Technology Workshops, pages 637–642, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[13] Mohamed Fayad and Douglas C. Schmidt. Object-oriented application frameworks. *Commun. ACM*, 40:32–38, October 1997.

[14] Thomas Genssler, Alexander Christoph, Michael Winter, Oscar Nierstrasz, Stéphane Ducasse, Roel Wuyts, Gabriela Arévalo, Bastiaan Schönhage, Peter Müller, and Chris Stich. Components for embedded software: the pecos approach. In *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '02, pages 19–26, New York, NY, USA, 2002. ACM.

[15] Object Management Group. Corba Component Model V4.0. Available on: http://www.omg.org/spec/CCM/4.0/.

[16] Scott Hissam, Gabriel Moreno, Judith Stafford, and Kurt Wallnau. Packaging predictable assembly. In Judith Bishop, editor, *Component Deployment*, volume 2370 of *Lecture Notes in Computer Science*, pages 108–124. Springer Berlin / Heidelberg, 2002.

[17] Ralph E. Johnson. Frameworks = (components + patterns). *Commun. ACM*, 40(10):39–42, 1997.

[18] Samuel Kounev, Fabian Brosig, Nikolaus Huber, and Ralf Reussner. Towards self-aware performance and resource management in modern service-oriented systems. *Services Computing, IEEE International Conference on*, 0:621–624, 2010.

[19] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[20] Johan J. Lukkien. private communication.

[21] Johan Muskens, Michel R. V. Chaudron, and Johan J. Lukkien. Component-based software development for embedded systems. chapter A component framework for consumer electronics middleware, pages 164–184. Springer-Verlag, Berlin, Heidelberg, 2005.

[22] Bojan Orlic, Ionut David, Rudolf Mak, and Johan Lukkien. Dynamically reconfigurable resource-aware component framework: Architecture and concepts. In Ivica Crnkovic, Volker Gruhn, and Matthias Book, editors, *Software Architecture*, volume 6903 of *Lecture Notes in Computer Science*, pages 212–215. Springer Berlin / Heidelberg, 2011.

[23] Mike P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing: a research roadmap. *Int. J. Cooperative Inf. Syst*, 17(2):223–255, 2008.

[24] Juraj Polakovic, Sebastien Mazare, Jean-Bernard Stefani, and Pierre-Charles David. Experience with safe dynamic reconfigurations in component-based embedded systems. In Heinz Schmidt, Ivica Crnkovic, George Heineman, and Judith Stafford, editors, *Component-Based Software Engineering*, volume 4608 of *Lecture Notes in Computer Science*, pages 242–257. Springer Berlin / Heidelberg, 2007.

[25] A. Rasche and A. Polze. Dynamic reconfiguration of component-based real-time software. In *Object-Oriented Real-Time Dependable Systems, 2005. WORDS 2005. 10th IEEE International Workshop on*, pages 347 – 354, feb 2005.

[26] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[27] Andrew Tanenbaum. *Computer Networks.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 4th edition, 2003.

[28] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The koala component model for consumer electronics software. *Computer*, 33:78–85, 2000.

A UML diagrams

In this appendix we present the mathematical models from sections 5, 6 and 7 and their relationships by means of UML class diagrams. Classes that correspond to universes, prescriptions and other mathematical models are indicated by color: yellow for universes, red for prescriptions, green for type models and blue for instance models (see Figure 9).

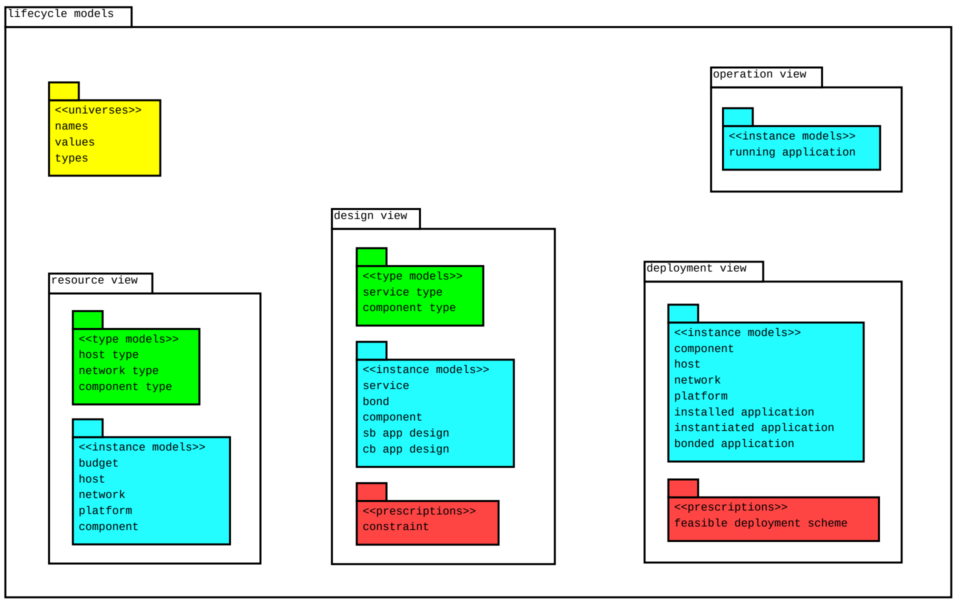


Figure 9: Model overview.

Besides these classes, there are also classes to represent model attributes like sets and functions. Note that in UML-diagrams we use currying to model the multivariate functions from the mathematical models. Wherever possible, we use annotated dotted arrows to express model constraints. In combination with function classes, this yields diagrams that can become quite complex. For one of the most complex diagrams, viz. the one in Figure 22 containing the bonded application model, we explicitly show how the mathematical constraints can be retrieved from the diagram.

In principle, we strive for a 1-1 correspondence with the names used in the mathematical models. Lack of proper fonts in UML diagrams, however, leads to ambiguities, which we resolve by prefixing names with a font indication: “ss” for sans serif, “cal” for calligraphic, and “ds” for double struck.

All diagrams in this appendix are intended as visualisations of the mathematical models, which remain authoritative.

Resource View Models

The models of the resource view are captured by two diagrams. The first diagram, displayed in Figure 10, contains the models pertaining to hardware resources.

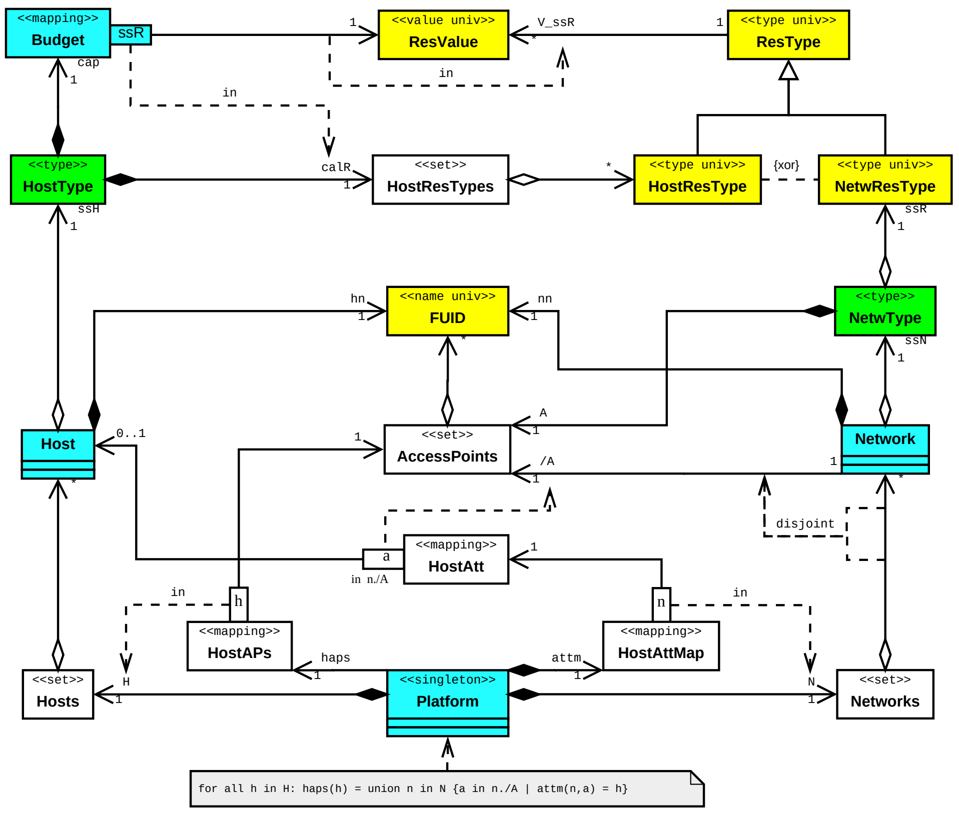


Figure 10: Hardware resource models.

The second diagram, displayed in Figure 11, contains the models pertaining to the resource usage of components.

Note that, by superimposing boxes with equally named classes, the diagrams in these figures can be combined into a single diagram. Care has to be taken with associations, however. Although the demand budgets $\mathtt{dmd}_{min}(m)$ and $\mathtt{dmd}_{max}(m)$ from Figure 11 and the capacity budget (`cap`) from Figure 10 both range over a set of resource types, indicated by a dashed arrow labeled "`in`" from the qualifier `ssR` to the resource type set `calR`, these sets are distinct, because they are represented by part-of relationships originating from distinct classes. This is a phenomenon that is more easily expressed by using separate diagrams [7].

Conversely, although possible, we have not split the diagram from Figure 10 into separate diagrams to deal with the individual platform resources in isolation, because here, in our opinion, the single diagram contributes to a clear perception of the dependencies between the various mathematical models.

[7] We adopt the convention that an "`in`" constraint on a qualifier holds only for all incoming arrows of the qualified class visible in the diagram that exhibits the constraint.

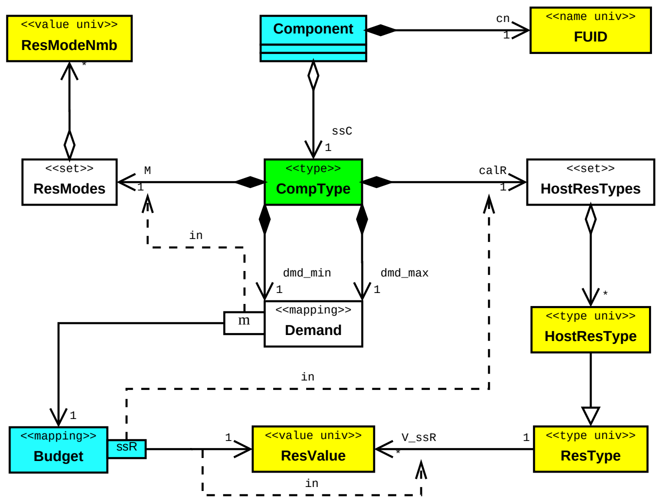


Figure 11: Component resource usage models.

Design View Models

The design view models and their relationships are captured by means of three UML class diagrams. The first diagram, displayed in Figure 12, contains the models for bonds,services, components, and their types. Notice the usage of specialization instead of composition to indicate that a component type in the design is considered an extension of a component type in the resource view. This has been expressed by the usage of an anonymous attribute in the mathematical model.

The second diagram, displayed in Figure 13 contains the application design models.

The third diagram, displayed in Figure 14, contains the constraint model. Although its constraints are defined relative to a component-based application design model and a platform model, the latter two are not part of the constraint model. In the UML diagram this is exemplified by the fact that there are only dependencies but no associations between the classes of the various models. To emphasize this fact, the relevant classes of both the component-based application design model and the platform model are put in grey boxes.

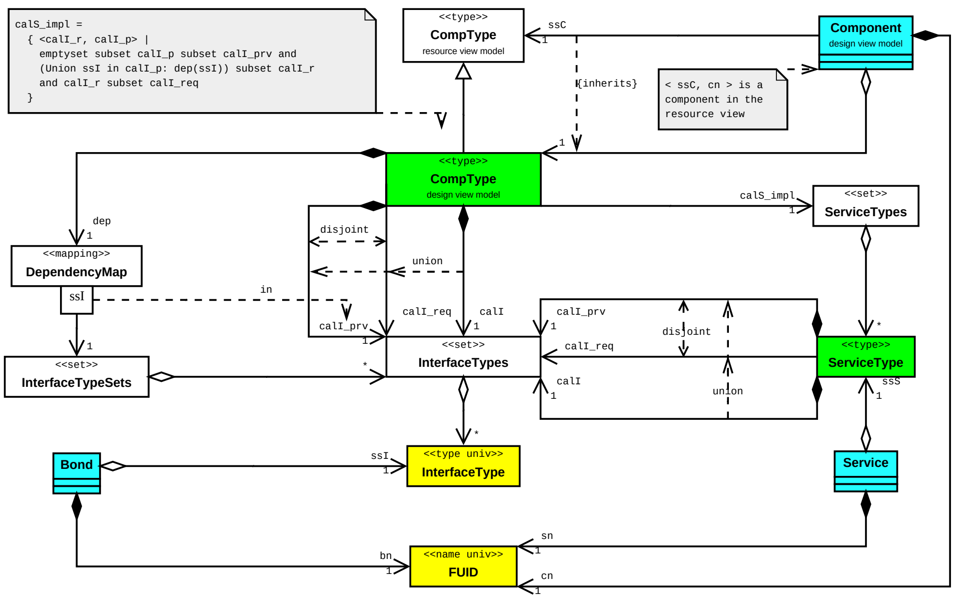

calS_impl =
{ <calI_r, calI_p> |
emptyset subset calI_p subset calI_prv and
(Union ssI in calI_p: dep(ssI)) subset calI_r
and calI_r subset calI_req
}
<<type>> CompType resource view model
Component design view model
ssC
{inherits}
< ssC, cn > is a component in the resource view
<<type>> CompType design view model
calS_impl
<<set>> ServiceTypes
dep
<<mapping>> DependencyMap
ssI
in
disjoint
union
calI_req
calI
calI_prv
<<set>> InterfaceTypes
<<set>> InterfaceTypeSets
calI_prv
calI_req
disjoint
union
calI
<<type>> ServiceType
ssS
Bond
ssI
<<type univ>> InterfaceType
Service
sn
bn
<<name univ>> FUID
cn

Figure 12: Design view: component, service and bond models.

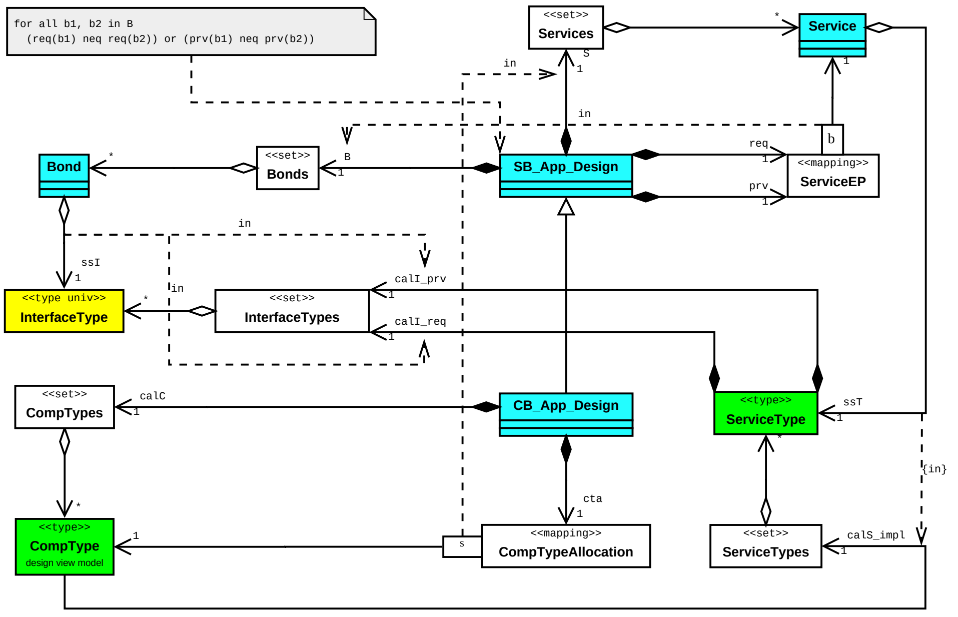


Figure 13: Design view: application models.

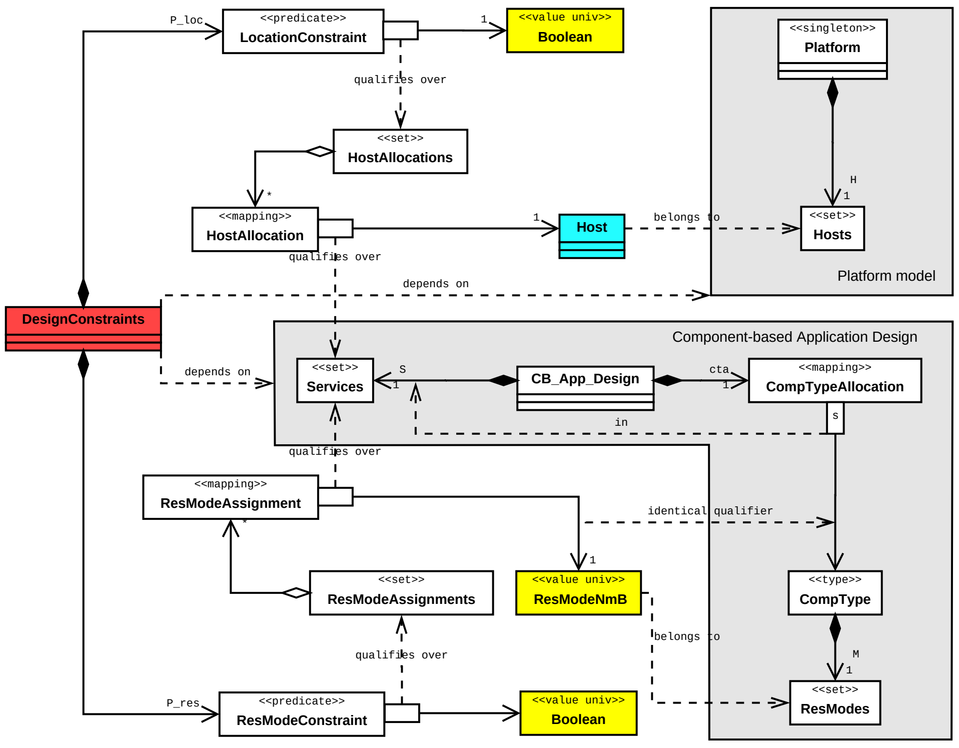


Figure 14: Design view: constraint model.

Deployment view models

The deployment view models and their relationships are captured by means of nine UML class diagrams. Figure 15 contains the component model which extends the one from the design view.

The models that describe of the platform and its composing entities and types are presented in Figure 16, Figure 17 and Figure 18. They are extensions of the corresponding resource view models. Due to the increased complexity they can no longer be presented in a single diagram althought the previous structure can still be recognized.

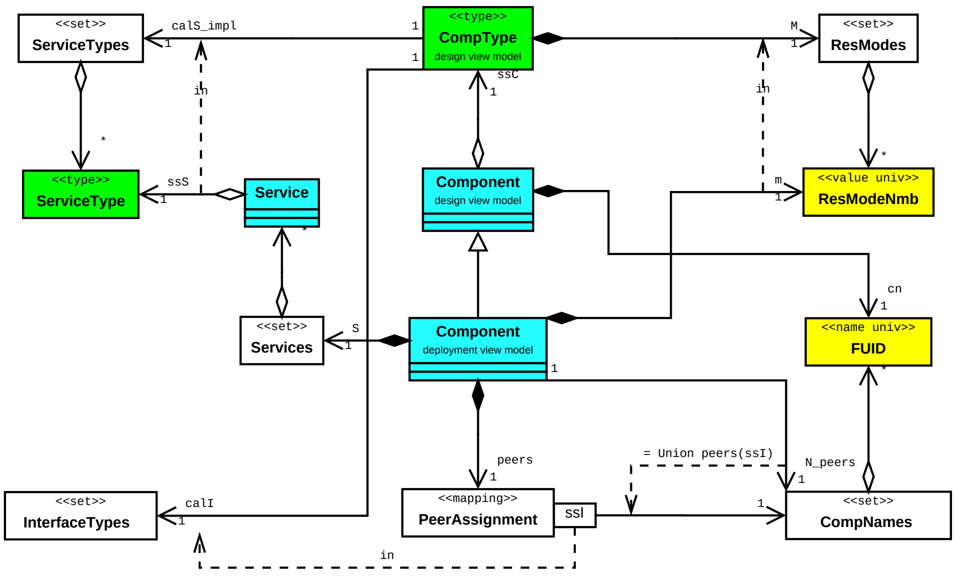


Figure 15: Deployment view: component model.

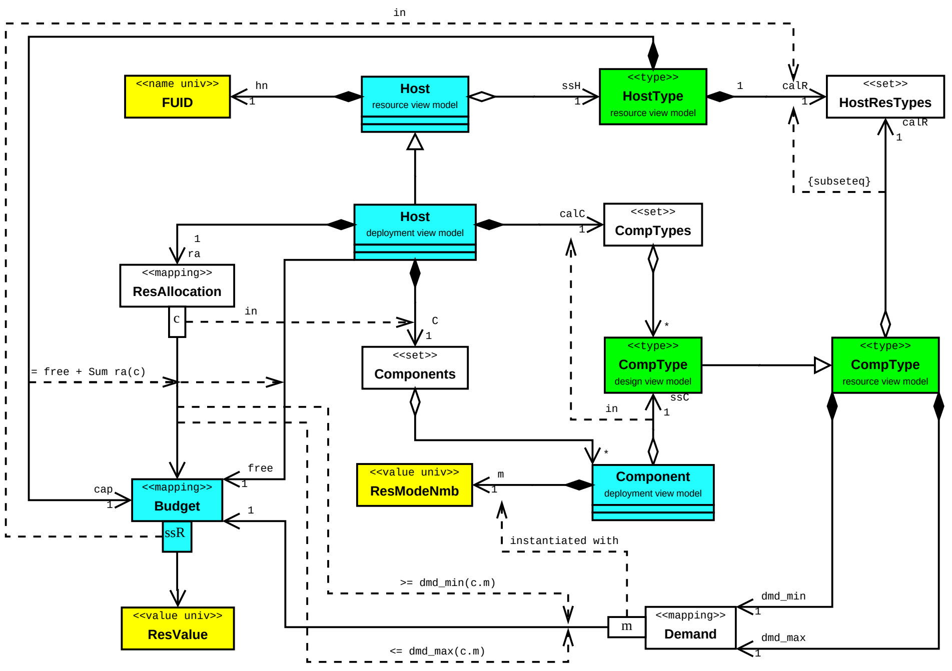


Figure 16: Deployment view: host model.

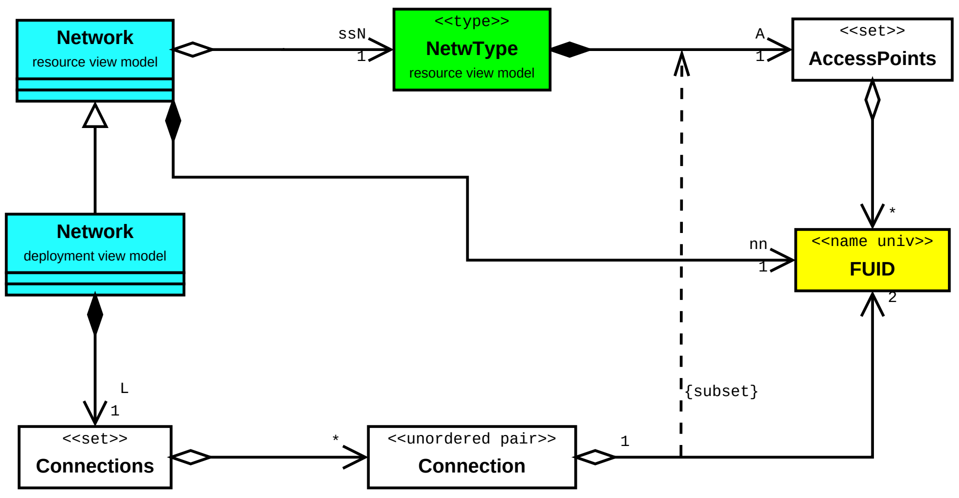
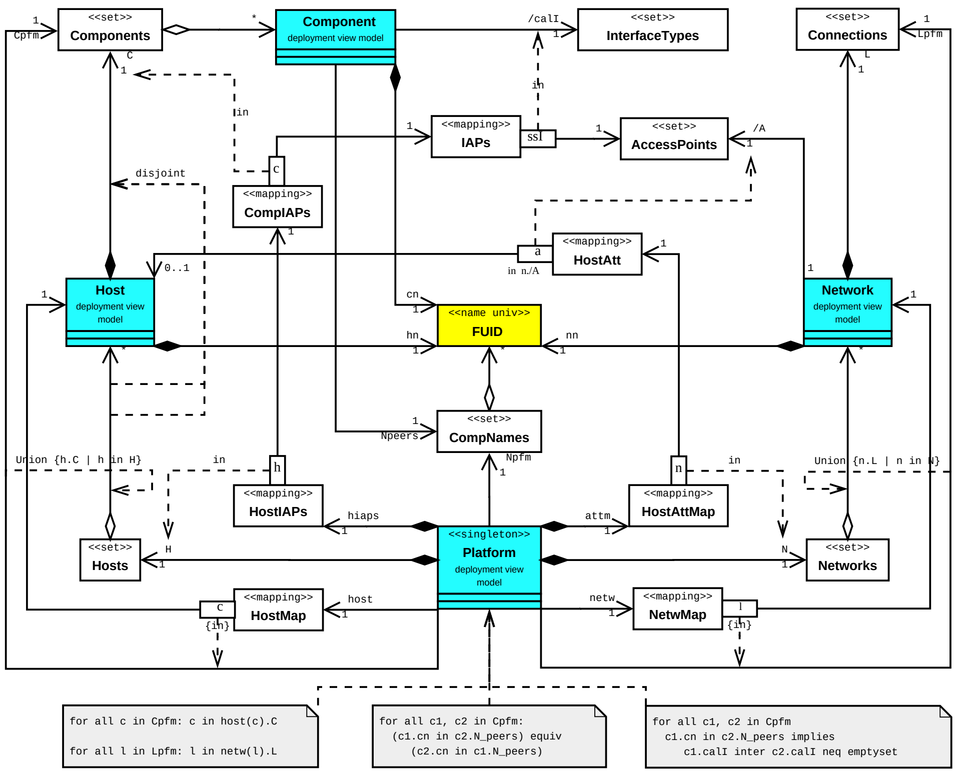


Figure 17: Deployment view: network model.



```
for all c in Cpfm: c in host(c).C

for all l in Lpfm: l in netw(l).L
```

```
for all c1, c2 in Cpfm:
  (c1.cn in c2.N_peers) equiv
     (c2.cn in c1.N_peers)
```

```
for all c1, c2 in Cpfm
  c1.cn in c2.N_peers implies
     c1.calI inter c2.calI neq emptyset
```

Figure 18: Deployment view: platform model.

Finally, we present diagrams for the models that are concerned with the deployment proper and that capture the outcome of the various activities of the deployment phase of the life cycle in Figure 3. We use two diagrams to express the feasible deployment scheme.

The first diagram, displayed in Figure 19 contains the host allocation mapping and its first constraints. To express the latter, a number of associations have been introduced that are absent in the mathematical model. First of all, we associate with each bond two specific services s_{req} and s_{prv}, which are the service endpoints of the bond as given by the corresponding mappings of the component-based application design. Next, by application of the host allocation mapping to these services, we obtain two specific hosts h_{req} and h_{prv} of the platform, to which in turn we apply the host access point set map `haps` to obtain two specific sets of host access points, viz. aps_{prv}, and aps_{req}.

The second diagram, displayed in Figure 20, contains the remaining parts of the feasible deployment scheme model, i.e., the remaining constraint of the host allocation mapping and the mode assignment and resource reservation with their constraints.

The last three diagrams contain the installed application model (Figure 21), the instantiated application model (Figure 22) and the bonded application model (Figure 23).

For the bonded application model we now show how the constraint on the connection allocation mapping can be retrieved from the diagram. We derive

$$
\begin{array}{ll}
 & \mathtt{la}\,(b) \\
= & \quad \{ \text{ see dependency } \textcircled{1} \text{in the diagram } \} \\
 & \{b.iaps_{prv},\ b.iaps_{req}\} \\
= & \quad \{ \text{ see dependency } \textcircled{2} \text{in the diagram } \} \\
 & \{g_{prv}(b.\mathsf{l}),\ g_{req}(b.\mathsf{l})\} \\
= & \quad \{ \text{ see dependency } \textcircled{3} \text{in the diagram } \} \\
 & \{(f_{prv}(c_{prv}))(b.\mathsf{l}),\ (f_{req}(c_{req}))(b.\mathsf{l})\} \\
= & \quad \{ \text{ see dependency } \textcircled{4} \text{in the diagram } \} \\
 & \{((\mathbb{P}.\mathtt{hiaps}\,(h_{prv}))(c_{prv}))(b.\mathsf{l}),\ ((\mathbb{P}.\mathtt{hiaps}\,(h_{req}))(c_{req}))(b.\mathsf{l})\} \\
= & \quad \{ \text{ remove Currying } \} \\
 & \{\mathbb{P}.\mathtt{hiaps}\,(h_{prv}, c_{prv}, b.\mathsf{l}),\ \mathbb{P}.\mathtt{hiaps}\,(h_{req}, c_{req}, b.\mathsf{l})\} \\
= & \quad \{ \text{ see dependency } \textcircled{5} \text{in the diagram } \} \\
 & \{\mathbb{P}.\mathtt{hiaps}\,(h_{prv}), \mathtt{ca}\,(s_{prv}), b.\mathsf{l}),\ \mathbb{P}.\mathtt{hiaps}\,(h_{req}), \mathtt{ca}\,(s_{req}), b.\mathsf{l})\} \\
= & \quad \{ \text{ see the diagram in Figure 19 } \} \\
 & \{\mathbb{P}.\mathtt{hiaps}\,(\mathtt{ha}\,(s_{prv}), \mathtt{ca}\,(s_{prv}), b.\mathsf{l}),\ \mathbb{P}.\mathtt{hiaps}\,(\mathtt{ha}\,(s_{req}), \mathtt{ca}\,(s_{req}), b.\mathsf{l})\}
\end{array}
$$

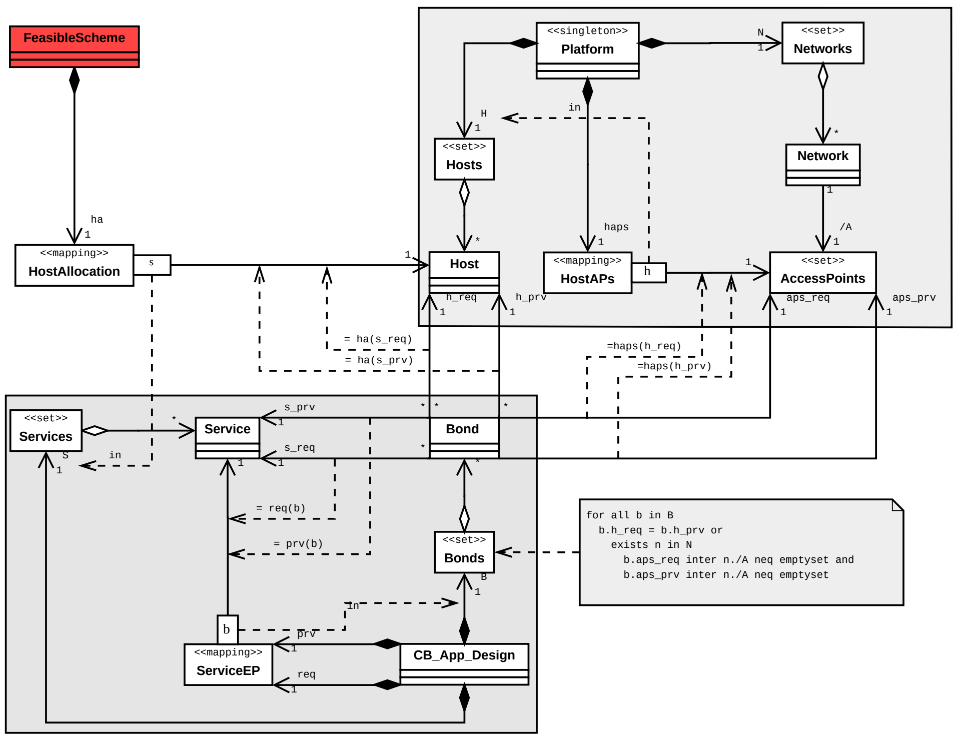


Figure 19: Feasible deployment scheme model, part 1.

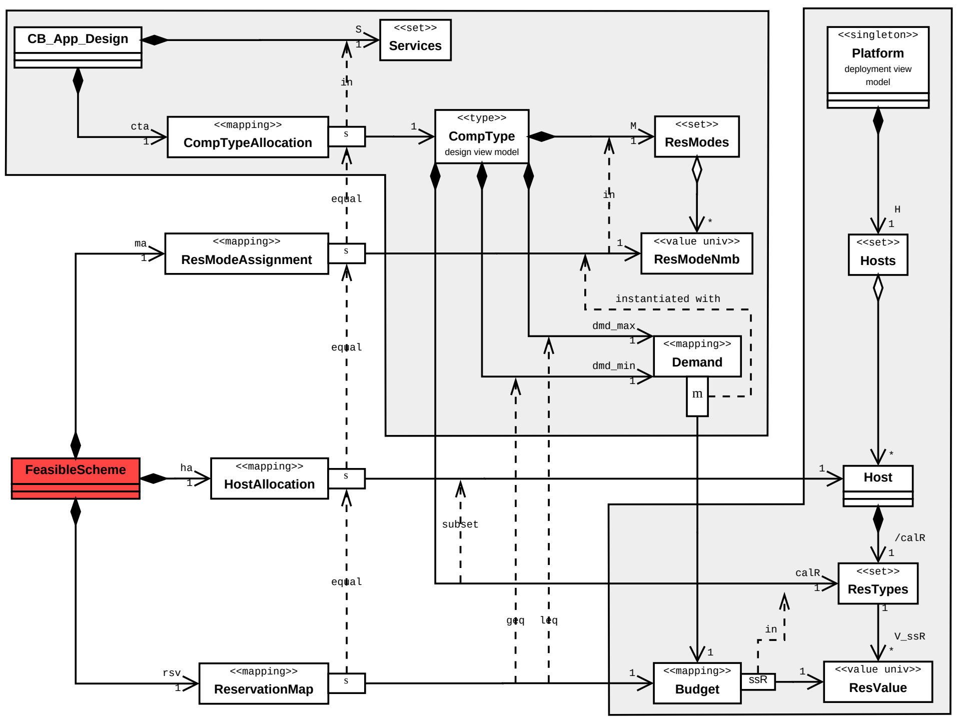


Figure 20: Feasible deployment scheme model, part 2.

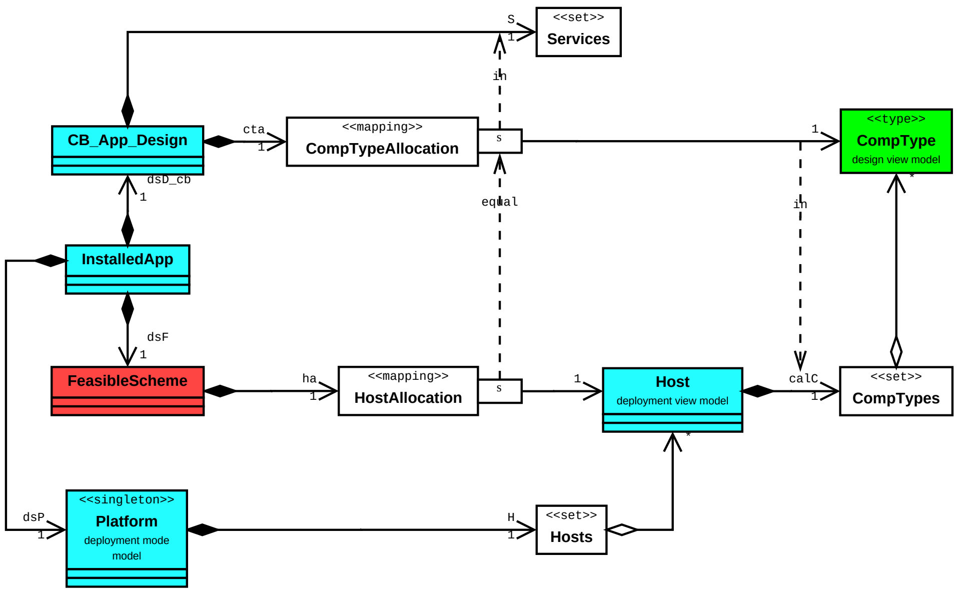


Figure 21: Deployment view: installed application model.

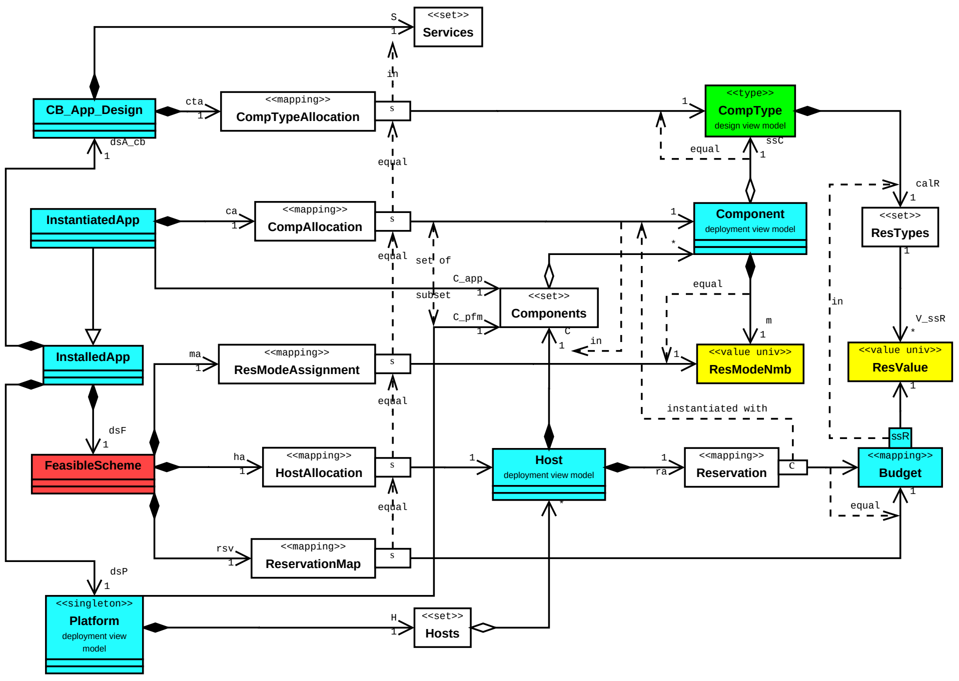


Figure 22: Deployment view: instantiated application model.

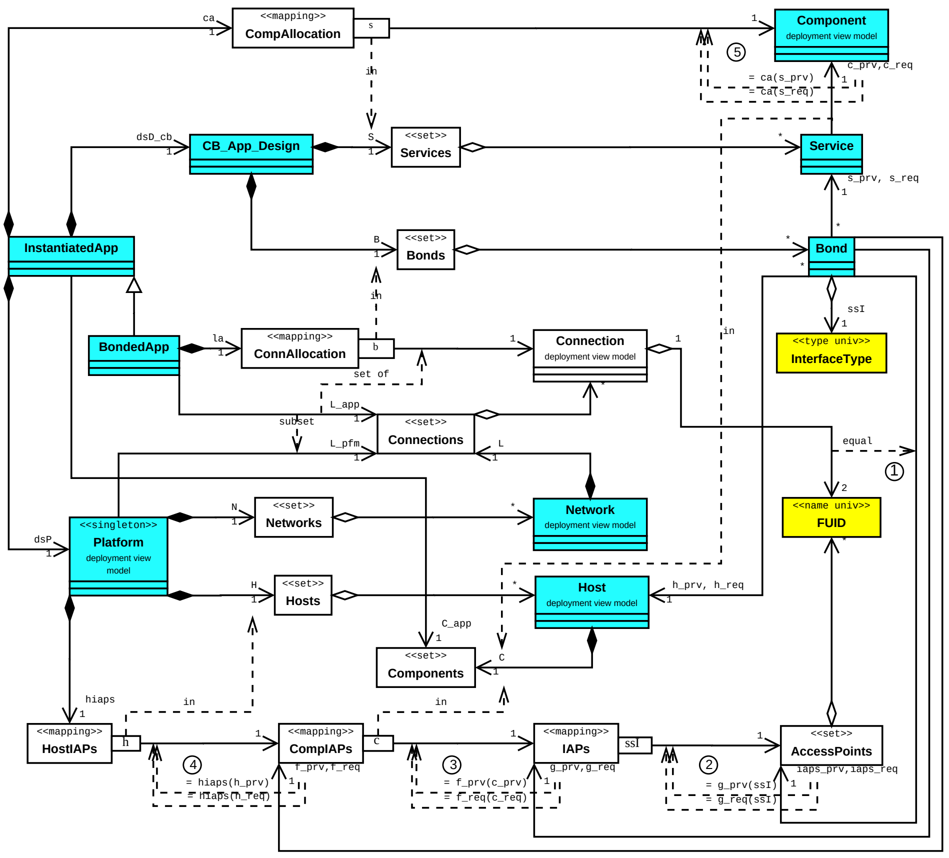


Figure 23: Deployment view: bonded application model. For the definition of associations $s_{prv}, s_{req}, h_{prv}$ and h_{req} of a *Bond*-object, see the diagram in Figure 19.

10/03	Luca Aceto, Matteo Cimini, Anna Ingolfsdottir, MohammadReza Mousavi and Michel A. Reniers	On Rule Formats for Zero and Unit Elements
10/04	Hamid Reza Asaadi, Ramtin Khosravi, MohammadReza Mousavi, Neda Noroozi	Towards Model-Based Testing of Electronic Funds Transfer Systems
10/05	Reinder J. Bril, Uğur Keskin, Moris Behnam, Thomas Nolte	Schedulability analysis of synchronization protocols based on overrun without payback for hierarchical scheduling frameworks revisited
10/06	Zvezdan Protić	Locally unique labeling of model elements for state-based model differences
10/07	C.G.U. Okwudire and R.J. Bril	Converting existing analysis to the EDP resource model
10/08	Muhammed Atif, Sjoerd Cranen, MohammadReza Mousavi	Reconstruction and verification of group membership protocols
10/09	Sjoerd Cranen, Jan Friso Groote, Michel Reniers	A linear translation from LTL to the first-order modal μ-calculus
10/10	Mike Holenderski, Wim Cools Reinder J. Bril, Johan J. Lukkien	Extending an Open-source Real-time Operating System with Hierarchical Scheduling
10/11	Eric van Wyk and Steffen Zschaler	1st Doctoral Symposium of the International Conference on Software Language Engineering (SLE)
10/12	Pre-Proceedings	3rd International Software Language Engineering Conference
10/13	Faisal Kamiran, Toon Calders and Mykola Pechenizkiy	Discrimination Aware Decision Tree Learning
10/14	J.F. Groote, T.W.D.M. Kouters and A.A.H. Osaiweran	Specification Guidelines to avoid the State Space Explosion Problem
10/15	Daniel Trivellato, Nicola Zannone and Sandro Etalle	GEM: a Distributed Goal Evaluation Algorithm for Trust Management
10/16	L. Aceto, M. Cimini, A.Ingolfsdottir, M.R. Mousavi and M. A. Reniers	Rule Formats for Distributivity
10/17	L. Aceto, A. Birgisson, A. Ingolfsdottir, and M.R. Mousavi	Decompositional Reasoning about the History of Parallel Processes
10/18	P.D. Mosses, M.R. Mousavi and M.A. Reniers	Robustness os Behavioral Equivalence on Open Terms
10/19	Harsh Beohar and Pieter Cuijpers	Desynchronisability of (partial) closed loop systems
11/01	Kees M. van Hee, Natalia Sidorova and Jan Martijn van der Werf	Refinement of Synchronizable Places with Multi-workflow Nets - Weak termination preserved!
11/02	M.F. van Amstel, M.G.J. van den Brand and L.J.P. Engelen	Using a DSL and Fine-grained Model Transformations to Explore the boundaries of Model Verification
11/03	H.R. Mahrooghi and M.R. Mousavi	Reconciling Operational and Epistemic Approaches to the Formal Analysis of Crypto-Based Security Protocols
11/04	J.F. Groote, A.A.H. Osaiweran and J.H. Wesselius	Benefits of Applying Formal Methods to Industrial Control Software
11/05	Jan Friso Groote and Jan Lanik	Semantics, bisimulation and congruence results for a general stochastic process operator
11/06	P.J.L. Cuijpers	Moore-Smith theory for Uniform Spaces through Asymptotic Equivalence
11/07	F.P.M. Stappers, M.A. Reniers and S. Weber	Transforming SOS Specifications to Linear Processes
11/08	Debjyoti Bera, Kees M. van Hee, Michiel van Osch and Jan Martijn van der Werf	A Component Framework where Port Compatibility Implies Weak Termination
11/09	Tseesuren Batsuuri, Reinder J. Bril and Johan Lukkien	Model, analysis, and improvements for inter-vehicle communication using one-hop periodic broadcasting based on the 802.11p protocol

11/10	Neda Noroozi, Ramtin Khosravi, MohammadReza Mousavi and Tim A.C. Willemse	Synchronizing Asynchronous Conformance Testing
11/11	Jeroen J.A. Keiren and Michel A. Reniers	Type checking mCRL2
11/12	Muhammad Atif, MohammadReza Mousavi and Ammar Osaiweran	Formal Verification of Unreliable Failure Detectors in Partially Synchronous Systems
11/13	J.F. Groote, A.A.H. Osaiweran and J.H. Wesselius	Experience report on developing the Front-end Client unit under the control of formal methods
11/14	J.F. Groote, A.A.H. Osaiweran and J.H. Wesselius	Ananlyzing a Controller of a Power Distribution Unit Using Formal Methods
11/15	John Businge, Alexander Serebrenik and Mark van den Brand	Eclipse API Usage: The Good and The Bad
11/16	J.F. Groote, A.A.H. Osaiweran, M.T.W. Schuts and J.H. Wesselius	Investigating the Effects of Designing Control Software using Push and Poll Strategies
11/17	M.F. van Amstel, A. Serebrenik And M.G.J. van den Brand	Visualizing Traceability in Model Transformation Compositions
11/18	F.P.M. Stappers, M.A. Reniers, J.F. Groote and S. Weber	Dogfooding the Structural Operational Semantics of mCRL2
12/01	S. Cranen	Model checking the FlexRay startup phase
12/02	U. Khadim and P.J.L. Cuijpers	Appendix C / G of the paper: Repairing Time-Determinism in the Process Algebra for Hybrid Systems ACP
12/03	M.M.H.P. van den Heuvel, P.J.L. Cuijpers, J.J. Lukkien and N.W. Fisher	Revised budget allocations for fixed-priority-scheduled periodic resources
12/04	Ammar Osaiweran, Tom Fransen, Jan Friso Groote and Bart van Rijnsoever	Experience Report on Designing and Developing Control Components using Formal Methods
12/05	Sjoerd Cranen, Jeroen J.A. Keiren and Tim A.C. Willemse	A cure for stuttering parity games
12/06	A.P. van der Meer	CIF MSOS type system
12/07	Dirk Fahland and Robert Prüfer	Data and Abstraction for Scenario-Based Modeling with Petri Nets
12/08	Luc Engelen and Anton Wijs	Checking Property Preservation of Refining Transformations for Model-Driven Development
12/09	M.M.H.P. van den Heuvel, M. Behnam, R.J. Bril, J.J. Lukkien and T. Nolte	Opaque analysis for resource-sharing components in hierarchical real-time systems - extended version –
12/10	Milosh Stolikj, Pieter J. L. Cuijpers and Johan J. Lukkien	Efficient reprogramming of sensor networks using incremental updates and data compression
12/11	John Businge, Alexander Serebrenik and Mark van den Brand	Survival of Eclipse Third-party Plug-ins
12/12	Jeroen J.A. Keiren and Martijn D. Klabbers	Modelling and verifying IEEE Std 11073-20601 session setup using mCRL2
12/13	Ammar Osaiweran, Jan Friso Groote, Mathijs Schuts, Jozef Hooman and Bart van Rijnsoever	Evaluating the Effect of Formal Techniques in Industry
12/14	Ammar Osaiweran, Mathijs Schuts, and Jozef Hooman	Incorporating Formal Techniques into Industrial Practice
13/01	S. Cranen, M.W. Gazda, J.W. Wesselink and T.A.C. Willemse	Abstraction in Parameterised Boolean Equation Systems
13/02	Neda Noroozi, Mohammad Reza Mousavi and Tim A.C. Willemse	Decomposability in Formal Conformance Testing